SUPPORTING THE USE OF ALGORITHMIC DESIGN IN ARCHITECTURE

AN EMPIRICAL STUDY OF REUSE OF DESIGN KNOWLEDGE

BY

ANASTASIA GLOBA

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the requirements for the degree of
Doctor of Philosophy

Victoria University of Wellington
(2015)

Abstract

This thesis tests the reuse of design knowledge as a method to support learning and use of algorithmic design in architecture.

The use of algorithmic design systems and programming environments offer architects immense opportunities, providing a powerful means to create geometries and allowing dynamic design exploration, but it can also impose substantial challenges. Architects often struggle with adopting algorithmic design methods (translating a design idea into an algorithm of actions), as well as with the implementation of programming languages, the latter often proving frustrating and creating barriers for both novice and advanced software users.

The proposition explored in this thesis is that the reuse of design knowledge can improve architects' ability to use algorithmic design systems, and reduce the barriers for using programming. This study explores and compares two approaches as a means of accessing and reusing existing design solutions. The first approach is the reuse of abstract algorithmic 'Design Patterns'. The second is the reuse of algorithmic solutions from specific design cases (Case-Based Design).

The research was set up as an experimental comparative study between three test groups: one group using Design Patterns, a second

group using Case-Based Design, and the control group. A total of 126 designers participated in the study providing sufficient numbers within each group to permit rigorous studies of the statistical significance of the observed differences.

Results of this study illustrate that the systematic inclusion of the Design Patterns approach to the learning strategy of programming in architecture and design, proves to be highly beneficial. The use of abstract solutions improves designers' ability to overcome programming barriers, and helps architects to adopt algorithmic design methods. The use of Design Patterns also encourages design exploration and experimentation. The use of the Case-Based Design approach seems to be more effective after designers and architects, who are novices in programming, gain more experience with the tool. It encourages more focused reasoning, oriented to the realisation of a particular (originally intended) design outcome.

The contribution of this research is to provide empirical evidence that the reuse of abstract and case-based algorithmic solutions can be very beneficial. Results of this study illustrate that both reuse methods can be strategically integrated into design education and architectural practice, supporting learning and use of algorithmic design systems in architecture. The study also identifies potential weaknesses of each approach, proposing areas which could be addressed by future studies.

Acknowledgements

Victoria University of Wellington (VUW) Graduate School supported this research. I would like to extend my most sincere thanks and appreciation to all those people who helped me accomplish this study.

Firstly, I would like to acknowledge my PhD supervisors Dr. Michael Donn, Prof. Jules Moloney and Simon Twose who provided invaluable insight, guidance and expertise. I would also like acknowledge the thesis examiners Prof. Marc Aurel Schnabel, Dr. Dermott McMeel and Prof. Karen Kensek who gave very constructive and positive feedback.

I wish to thank the teaching, administration and technical staff of the VUW School of Architecture and Design, in particular Selena Shaw, Carolyn Jowsey, Mara Dougall, Stewart Milne, Eric Camplin, Martin Hanley, Derek Kawiti and Valentina Soana, for their assistance and collaboration in my research and teaching projects. Thanks are also extended to my colleagues and fellow PhD students Nabil Allaf, Maryam Lesan, Ensiyeh Ghavampour, Maz Abadi, Remy Leblanc and Dekhani Nsaliwa. I have appreciated their encouragement and all the good times we have had together. I am also grateful to all of the VUW students and professional architects who took part in my design courses and algorithmic modelling workshops, which provided the data that made this investigation possible.

Gratitude is extended to Samantha Smith, Karen Henning-Hansen, Caitlyn Lee, Guy Marriage and Shaan Cory for their assistance with proofreading and comments that helped to improve the manuscript.

I would also like to acknowledge all those people who encouraged and inspired me to pursue my academic studies in architecture and digital design. I would like to recognize Prof. Christos Passas and Prof. Andrea Haase, my M.Arch. supervisors and extend my thanks to the teaching staff and to my fellow master students of DIA (Bauhaus) School of Architecture, Germany.

I would like to thank my home university in Russia, Magnitogorsk State Technical University (MGTU) and particularly our head of faculty, Oleg Ulchickiy, as well as the staff of the MGTU faculty of Architecture, for their support and interest in my research. Appreciation is extended to my first supervisor and mentor Ernst Zalmanovich Frenkel.

Finally, special recognition goes to my friends and family, especially my husband Sergey Maximov, for their support and encouragement during my pursuit of this Doctorate Degree in Architecture.

I thank all these people for their direct and indirect help in completing this research.

Papers published during study

Moloney, J. Globa, A.; Donn, M.; (2015) Urban Codes. Abstraction and Case-Based Approaches to Algorithmic Design and Implications for the Design of Contemporary Cities. CAAD Futures 2015 'The next city', [Proceedings of the 16th CAAD Futures Conference], Sao Paulo, Brazil *(expected to be published in July 2015)*

Globa, A.; Donn, M.; Moloney, J. (2014) Abstraction versus Case-Based: A Comparative Study of Two Approaches to Support Parametric Design, ACADIA 14: Design Agency [Proceedings of the 34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA) ISBN 9781926724478] Los Angeles, pp. 601-608

Globa, A., Ulchickiy, O., Donn, M, (2013). Metrics for measuring the effectiveness of parametric modelling in architecture. Architecture. Construction. Education. Proceedings of the Conference, "MGTU", Magnitogorsk, Russia

Globa, A., Donn, M., & Twose, S. (2012). Digital To Physical: Comparative Evaluation Of Three Main CNC Fabrication Technologies Adopted For Physical Modelling In Architecture. International Journal of Architectural Computing, 10(4), 461-480.

Globa A., Donn, M., Twose S., (2012) Digital to Physical. CAADRIA 2012. Beyond codes and pixels. Proceedings of the 17th International Conference on Computer-Aided Architectural Design Research in Asia, Chennai, India, pp. 327-337

Globa, A., Ulchickiy, O., (2012) CNC fabrication within Design process, Architecture. Construction. Education. Proceedings of the Conference dedicated to the 70th anniversary of the Architectural faculty "MGTU", Magnitogorsk, Russia, pp. 72 – 82. Архитектура. Строительство. Образование. Материалы конференции, ФГБОУ ВПО «МГТУ», 2012, - С. 72 - 82

Table of Contents

List of Exhibits

Exhibit 3.19. Key words used to describe parametric designs; Indexing in Case-Based Design

Exhibit 3.20. Indexing programming solutions/algorithmic modelling (all groups) day 1/day 2 key words count.

Exhibit 3.21. Typology and distribution of design objectives.

Exhibit 3.22. Examples of sketches (original design ideas) and corresponding output models, designed by participants using Design Patterns. Typical cases where designers have significantly changed their original idea and still reported that they were able to find a Design Pattern(s) that fit and were able to accomplish what they wanted.

Exhibit 3.23. Examples of sketches (original design ideas) and corresponding output models, designed by participants using Case-Based Design approach. Typical cases where designer managed to develop an output model that was close to their original idea and reported that they were able to find a Design Pattern(s) that fit and were able to accomplish what they wanted.

Exhibit 3.24. Examples of models, designed by participants, who used Design Patterns and were able to accomplish what they wanted; explored alternative design options; significantly changed the original idea; and developed more complex programming algorithms and output models.

Exhibit 3.25. Examples of models, designed by participants, who used Case-Based Design approach and were able to accomplish what they wanted; managed to model the original idea; and developed more simple programming algorithms and output models.

Exhibit 3.26: Overall amount of difficulties. Typology and distribution of programming difficulties.

Exhibit 3.27. Results of comparative study (all criteria).

Exhibit 4.1. Comparison between the male and female participants

Please note that illustrations showing the results of the comparative study are duplicated in the Report of Results (Appendix B) in section 'Diagrams and illustrations'.

Definitions

CAD – Computer Aided Design. Sometimes in the field of architecture expanded to CAAD: Computer Aided Architectural Design

Computation –refers to the use of mathematical or logic methods (Terzidis, 2006)

Computerisation – the mode of using computers in design practice (Menges, Ahlquist, 2011)

Algorithm –textually or diagrammatically represented set of instructions and rules. A procedure of solving a problem in a series of steps using the logic of if-then-else operations (Terzidis, 2006)

Algorithmic Design refers to the use of rule-based procedural logic and computation (Terzidis, 2006). It is typically performed through computer programming languages (Leach, 2010)

Parametric Design refers broadly to the use of parametric modeling programs. (Leach, 2010). It is based on the use of parameters (variables) and rules. Such terms as parametric and algorithmic have a large overlap (and are closely related), in some cases they can be interchangeable or can be seen as a synonyms (Davis, 2013).

Visual Programming – a type of algorithmic design method, which uses diagrammatic (e.g. box-and-wire) representation. In visual programming, program-elements (boxes) containing specific instructions and are used to

represent and manipulate the outcome model (Celani, Vaz, 2012). Such programs as: Grasshopper plugin for Rhinoceros (Grasshopper3d, 2012), (Rhino3d, 2012), Generative Components (GC) (Bentley, 2012), etc.

Script - list of commands written in a textual programming language, such as: Rhino Script (Rhinoscript, 2012), Mel (Autodesk, 2010), MaxScript (Autodesk, 2012), Python (Python, 2012).

Plugin – software component of a larger application that has specific abilities and functions within the main software framework

0. Introduction

The architectural profession could benefit from knowing more about knowledge reuse methods that can help architects and designers to overcome programming barriers and make the use of algorithmic modelling systems less problematic and more effective.

This thesis explores the reuse of design solutions as a support method for learning and using algorithmic design in architecture. The focus of the study is to test whether the reuse of design knowledge can improve architects' ability to understand and effectively use algorithmic modelling systems, and to help users to overcome barriers associated with the implementation of programming languages.

In the context of architecture the term algorithmic design refers to the use of rule-based procedural logic and computation (Terzidis, 2006), which typically operate through computer programming languages (Leach, 2010). The word 'algorithm' has Persian roots, and means a procedure of solving a problem in a series of steps using the logic of if-then-else operations (Terzidis, 2006). Algorithms are the soul of the computational design systems. They can be seen as an automated formula (a recipe) specifying procedural operations of the system, such as

calculating mathematical functions, searching, selecting objects, modifying them and generating output geometry (Menges, Ahlquist, 2011).

An increasing number of designers and architects choose to learn and use algorithmic modelling methods in their designs. One of the reasons for this is that algorithmic modelling tools incorporate both computational complexity and the creative use of computers (Terzidis, 2006). Algorithmic design combines the complexity and the creativity of CAD (Menges, Ahlquist, 2011); and enables designers to shift their role from 'architecture programming' to 'programming architecture' (Terzidis, 2006). It has been argued that computation allows architects to create original and complex design solutions that are difficult, or impossible, to achieve using other methods (McCormack, Dorin and Innocent, 2004).

The mathematical nature of scripts and visual programming definitions gives architects the ability to explore multiple output models, simply by changing the rules and the values of parameters. The use of algorithmic design enables architecture to go beyond 'a static creature' state, and become a fluid sequence of parametrically generated forms and patterns. Through computation, architecture 'transcends itself beyond the common and predictable' (Terzidis, 2006).

The other reason for growing interest in algorithmic modelling techniques is that the use of programming provides a means to overcome limitations of predefined commands and interfaces of CAD software. It allows CAD users more freedom and flexibility in the face of software constraints. By using programming languages architects can overcome 'the factory-set limitations' of CAD software (Ibid).

0.1 Problems with algorithmic design

While the use of algorithmic modelling systems provides architects and designers with tremendous opportunities, it can also impose considerable challenges (Menges, Ahlquist, 2011). These challenges are often associated with the acquisition of programming skills that traditionally are outside the architect's repertoire and design education. However the main challenge may not reside in mastering computation techniques, but rather in assimilating 'a mode of computational design thinking' (Menges, Ahlquist, 2011). Because the initial principles of human and computer reasoning do not follow the same patterns, it is not easy for some people to use programming algorithms when translating their idea into form. The algorithmic logic of idea-to-form translation introduces novel principles of design thinking (Matcha, 2007). Many designers find it difficult to integrate algorithmic thinking and programming techniques into the design process (Woodbury, 2010).

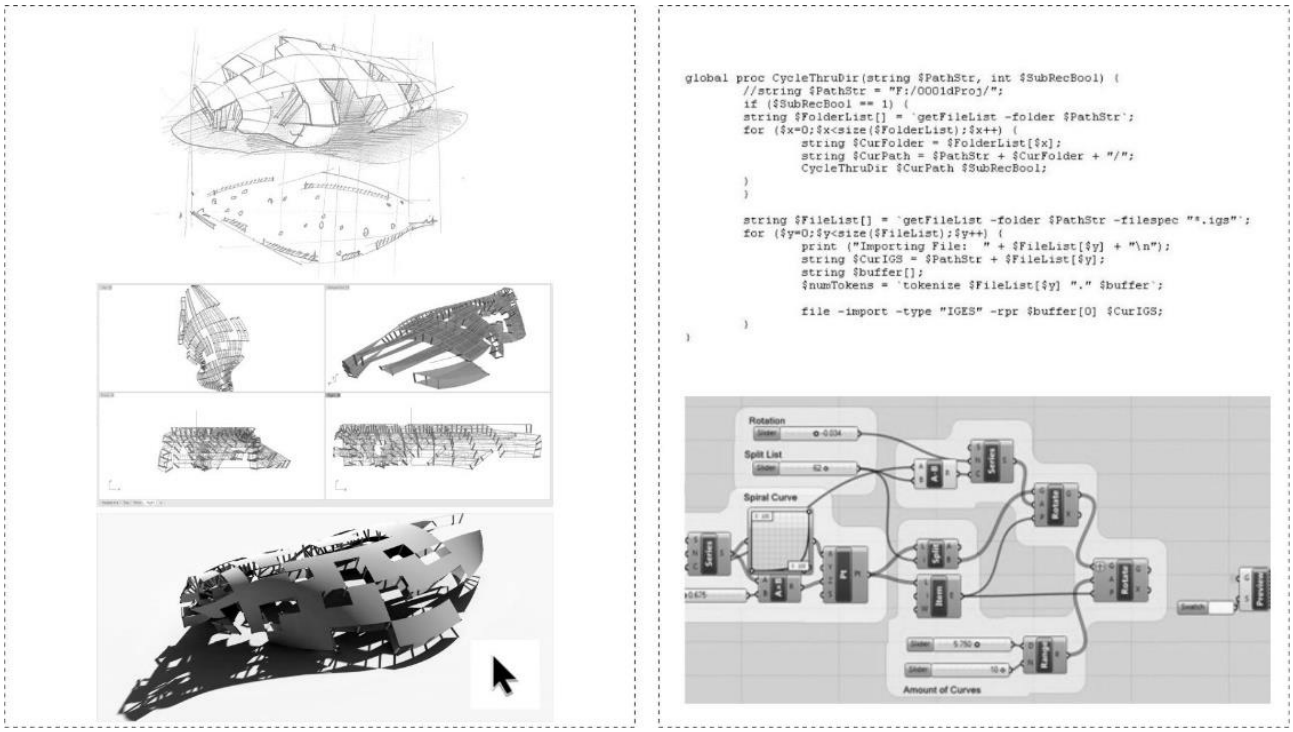


Exhibit 0.1 Traditional design languages and programming design languages

One of the reasons behind these problems is that there is a distinct gap between traditional design principles and algorithmic modelling methods and rules. Most architects and architectural students find it difficult to shift from conventional freehand drawing and modelling (including manual CAD modelling) to describing their ideas through the language of algorithms and codes (Exhibit 0.1)

Algorithmic modelling systems are operated through symbolic (scripting) or analogue (visual) programming languages (Exhibit 0.1), which are used as the means to actualise an idea-to-form translation (Mitchell 1975). The implementation of these programming languages can be frustrating and cause many difficulties for both novice and advanced users. Many architects face difficulties with adopting their logic and syntax (Celani and Vaz, 2012; Woodbury 2010). Understanding and learning the programing framework syntax rules can be especially frustrating to novice users (Celani, Vaz, 2012).

As a result, adopting algorithmic design principles and mastering programming techniques often requires additional effort from designers and architects, many of whom face substantial barriers with understanding and using algorithmic design methods. While software developers work towards improving the characteristics of design systems (making more intuitive and flexible programming languages and interfaces), this thesis proposes to explore this issue from the perspective of design process itself.

The reuse of programs, algorithms and codes (software artefacts) is an important part of programming practice (Krueger, 1992). Software engineers and architects using algorithmic modelling systems share similar challenges (Davis, 2013). However, the systematic reuse of design solutions is not a part of algorithmic design practice in architecture, and we can learn from programming practices (Woodbury, 2010) (Davis, 2013).

0.2 Research hypothesis: reuse of knowledge as a design support method

This thesis proposes to test the reuse of algorithmic solutions as a design support method with the aim of to reducing barriers to the use of programming and improving architect's ability to use algorithmic design systems. Conceptually, this thesis asks how might one test experimentally the reuse idea? The primary research strategy is to work with two alternative radically different reuse methods that are well established and discussed in literature. Two different approaches are proposed to test the idea of design knowledge-sharing and the reuse of the solutions: the first is to learn and reuse abstract solutions (Design Patterns), the second is to reuse case-based solutions using a database system.

The proposed methods of reusing abstract and case-based knowledge are not new. Over the past few decades the pattern and case-based design approaches have been adopted by educators and practitioners in various fields of design, architecture and software development. This thesis aims to test these approaches as a means of accessing and reusing existing knowledge in the context of algorithmic design in architecture. Neither of these approaches is a research target in itself, but they are a vehicle through which this research investigates the impact of each method on the design process. It wants to know whether re use of knowledge may be of help. It selects two radically different approaches to knowledge re use to test this bigger idea

The abstraction reuse approach is tested using Design Patterns developed by Robert Woodbury (2010). These Design Patterns focus on generalised methods of structuring programming solutions, and address both problems with programming (code) itself, as well as with solving

problems specific to architecture (Davis, 2013). According to Woodbury Design Patterns are a theory, which is yet to be tested (Woodbury, 2010)

Research hypothesis

The research objective of this thesis is to test the hypothesis that the reuse of abstract programming solutions (Design Patterns) can help designers to overcome programming barriers and improve their algorithmic modelling performance (Woodbury, 2010), and to compare it with the alternative Case-Based Design approach (the reuse of specific programming solutions).

Reuse methods: abstract solutions versus solutions from specific design cases

Typically, every reuse technique (abstract or case-based) involves selection, specialisation and integration of artefacts, though the degree of involvement may vary depending on the reuse approach. The purpose for the reuse of programming artefacts is usually to reduce time and effort required to design systems (Krueger, 1992). This thesis investigates how each approach influences designers' ability to overcome barriers (reduce effort) and their ability to use algorithmic design methods (improve performance). The study tests whether the reuse of abstract and case-based solutions can reduce programming difficulties, increase the explored space of programming solutions, improve designers' ability to realise original design concepts, and accomplish all design objectives (See 'Detailed criteria for comparing the DP and CBD approaches' section).

The first approach is the reuse of abstract solutions to a design problem - Design Patterns (Woodbury, 2010). These patterns were developed to assist designers with structuring their own programming solutions on an abstract level. In his book 'Elements of Parametric Design' Woodbury states that, in architecture, designers tend to create algorithms anew, rather than reuse them (Woodbury, 2010). The development of an algorithmic structure is an 'act of high-level abstraction' (Menges, Ahlquist, 2011). Woodbury argues that designers can make their designs much more effective by employing reusable abstract parts (Design Patterns). The key concept of Design Patterns lies in the reuse of design knowledge (Alexander, Ishikawa, Silverstein, 1977). Instead of solving each new problem individually, architects can reuse the patterns successfully implemented in the past (Gamma, Helm, Johnson, Vlissides, 1994). The pattern methods have been adapted and tested in various disciplines including the field of object-oriented design (software development). This is particularly relevant, because both software design and algorithmic design operate using programming languages.

It has been suggested that an effective reuse technology implies the use of a high level of abstraction (Woodbury, 2010), (Gamma, Helm, Johnson, Vlissides, 1994), (Krueger, 1992). The idea is that a designer should know 'what' the reusable artefacts do rather than 'how' they do it. However, there are difficulties associated with the reuse of abstractions, because in order to use abstract solutions a designer must be familiar with the abstractions prior to the design process, which requires time to study and understand these abstractions (Krueger, 1992). This suggests that for a reuse technique to be effective it must be easier to reuse an existing artefact (solution) than it is to develop a new system from scratch (Ibid).

The works of both Alexander et al. (Alexander, Ishikawa, Silverstein, 1977) and Gamma et al. (Gamma, Helm, Johnson, Vlissides, 1994) helped Robert Woodbury to identify the following structure of patterns: each design pattern has to be explained using the 'Name', 'Intent', 'Use When', 'Why' and 'How' and it should be illustrated by a set of samples (examples) (Design Patterns, 2014) (Woodbury, 2010).

In summary, Design Patterns are generalised reusable solutions, described with a high level of abstraction, and documented in such a way as to be broad enough to apply to a range of different design contexts. Woodbury has outlined the following principles of patterns for parametric design (Woodbury, 2010):

- Explicit. Others should be able to read (understand) your patterns in your absence.
- Partial: separate solutions to problem parts;
- Problem focused: a pattern should solve a shared problem;
- Abstract. Patterns are abstract and represent a general concept.

The second approach is the reuse of specific programming solutions, employing case-based reasoning principles (Kolodner, 1993). Case–Based Reasoning (CBR) is a problem solving approach which utilises specific knowledge from previous cases, instead of making assumptions based on generalised relationships between a description of a problem and conclusions (Aamodt, Plaza, 1994). In CBR a new problem is solved by finding and reusing an existing solution from a similar case from the past. In other words, in order to solve a new problem one finds a previous situation and reuses the knowledge of its solution in a new context. Case-based reasoning is a cognitive model proposing that thinking by analogy is consistent with natural patterns of problem solving. (Kolodner, 1993). It is argued that CBR is used by people as a primary mechanism for common

reasoning on a daily basis; there is evidence that when humans solve new problems they predominantly rely on specific, previous encountered situations (Ibid) (Riesbeck, Schank, 2013). Research on human cognition shows that people tend to use previous cases as models both when they are novices (Anderson, 2013) and when they are experts (Rouse, Hurt, 1982).

Studies on the use of case-based design in architecture indicate that designers can benefit from past cases, by adapting similar design solutions (Heylighen, Verstijnen, 2000). One of the fundamental strategies in acquiring knowledge is to learn by example. In architecture examples are design cases, however there is a fundamental difference between learning by example and case-based reasoning. In case-based reasoning cases 'are generalised with respect to the context of a specific problem during each problem solving process' (Hua, Fairings, Smith, 1996). Traditionally, in the field of design, knowledge has been recorded and formalised as examples of successful design outcomes, rather than generalised in the form of principles (Ibid).The approaches using case-based reasoning incorporate the following principles (Aamodt, Plaza, 1994):

- Identification of a new problem (new case);
- Finding a similar past case (existing solution in a case-base);
- Use of this past case to solve (suggest a solution for) a new problem
- Evaluation of your solution and update the case base by learning from your new experience (new solution).

In this thesis, the CBD (Case-Based Design) approach was tested through an online case-base of visually represented algorithmic models and corresponding downloadable programming algorithms. These cases, and their illustrations, were developed specifically for this research and

were labelled according to the design concept, shape and programming logic.

0.3 Research methodology

This research was designed as an experimental comparative study between three test groups: 1) a control group, 2) a group reusing abstract solutions (Design Patterns (DP)), and 3) a group reusing solutions from specific design cases (Case-Based Design (CBD)). The approaches are tested in a series of algorithmic modelling workshops for architects, and landscape and interior architects. Participants recruited to participate in the experimental part of the study are a diverse group of students, and practicing architects, with no age restriction, but a minimum of one year experience in design. A total of 126 people participated in the study providing sufficient numbers within each group to permit rigorous studies of the statistical significance of the observed differences. Continuous variables were compared using the t-test (for two test groups), and ANOVA (for three test groups); binary data was compared with the chi-square test (all statistical testing was done using SPSS) (IBM SPSS, 2014) (See Statistical methods section).

The study was organised in the form of two-day algorithmic modelling workshops. Each workshop offered an introduction to algorithmic design using Grasshopper (Grasshopper3D, 2014) for Rhinoceros (Rhino3D, 2014) (See Methodology and Experiment set-up sections). Grasshopper 3D is often referred to as a parametric or an algorithmic modelling system, which is why in this study, Grasshopper algorithms (definitions) are referred to as parametric/algorithmic solutions. On each day participants were given one design assignment, which they

were to develop on their own. This was preceded by an introductory series of exercises focused on familiarisation with the software and the DP and CBD groups were additionally taught how to use the respective reuse approach. Participants modelled and submitted their designs within a two-hour period. The collected data consisted of submitted 3D models, programming definitions and survey results. The 3D Rhino models were used to calculate the level of complexity of each model. The Grasshopper definitions were used to measure the complexity of each programming algorithm and to determine the explored solution space of each algorithm.

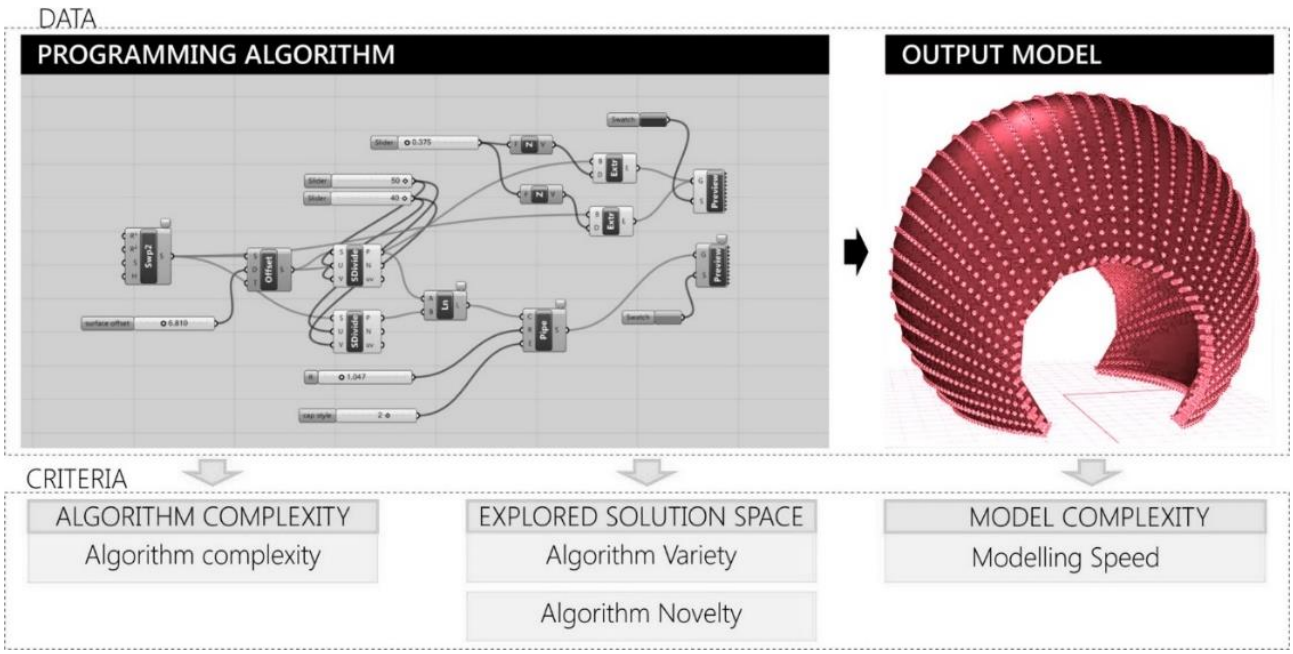


Exhibit 0.2 Algorithmic Modelling Performance: Data/Criteria, illustrating: Programming Algorithm which generates an Output Model; and shows the sources of the data, informing the corresponding algorithmic Modelling Criteria.

In Exhibit 0.2 the image on the left (labelled 'Data') illustrates an example of a programming algorithm (box-and-wire diagram made in Grasshopper for Rhino3D). The image on the right illustrates the output 3D model that is generated by the programming algorithm. The bottom row shows the respective 'Criteria' groups which were used to evaluate this programming algorithm, for example the 'Algorithm Complexity' and the

'Explored Solution Space'. The explored solution space is determined by the variety and novelty of a programming solution (Shah, Smith and Vargas-Hernandez, 2003). Variety refers to how many different programming components each algorithm has. Novelty evaluates how unusual (less frequently used at the group level) each programming component is (Ibid). The 'Model Complexity' criteria are derived from the output model. The methodology for measuring the complexity of the output models, was informed by geometrical, combinatory and dimensional complexity criteria for model classification– Shape Grammars (Forrest, 1974).

Questionnaires helped to determine the quantity and type of programming difficulties and the number of reused algorithms. They sought feedback from workshop participants on the levels of satisfaction with the design outcome, and their motivation to use algorithmic modelling systems in the future. The participants also provided data regarding their design objectives, their ability to model the original design idea and the degree of change made in the design due to programming difficulties.

The comparative study addressed the following criteria of algorithmic modelling performance, which outlines designers' ability to use algorithmic design systems (See Detailed Research Methodology section):

- Number of programming difficulties/type of programming barriers;
- Explored space of programming solutions (Novelty and Variety);
- Learning precedents;
- Degree of algorithm and output model complexity (modelling speed);

The aesthetic and design qualities of the models were not judged directly. However, these issues were addressed indirectly. Each participant was asked to indicate their design intentions and, reflecting on the design outcome, evaluate the degree of satisfaction with their produced model. This strategy also provided insight into what each person intended relative to what was actually achieved. To examine how each approach of reusing programming solutions (abstract or case-based) influences designers' ability to realise an idea-to-form translation within the algorithmic modelling environments, the following design performance criteria were identified:

- Ability to realise original idea
- Ability to accomplish all design objectives/typology of design objectives
- Change in design idea due to programming difficulties
- Change in design idea due to discovery of more interesting reusable solutions
- Participants' satisfaction with the design outcome
- Motivation to use algorithmic design in future

To investigate further designers' experience of the use of the DP and CBD approaches the following criteria were used of:

- How easy-to-use
- How intuitive
- How helpful

The outlined criteria formed the evaluation metrics by which this study measured the effect (empirical evidence) of the reuse of abstract and case-based algorithmic solutions in algorithmic design architecture. This evidence was used as a means to answer the research questions.

Research scope

The overall principles of both abstract and case-based reuse approaches can potentially be used with any algorithmic design software, and used with both textual and visual programming languages. In theory, regardless of the type of software programs that are currently used by architects or will be used in future, the principles of reusing abstract and specific algorithmic solutions will remain the same. However, in the context of this study the Design Patterns (DP) and Case-Based Design (CBD) approaches are tested using visual programming with Grasshopper/Rhino (Grasshopper3D, 2014) (Rhino3D, 2014). Grasshopper uses visual programming language. The section 'Expanding beyond the scope of this research' discusses the boundaries of the study in more detail.

The target group of this study is students and professional designers and architects, both novice and experienced programming users. However, even though the recruited participants were a diverse group of both students and practitioners, their experience with algorithmic modelling tools, and particularly with the use of Grasshopper, was minimal. Thus, it is acknowledged, that this study tests the DP and CBD approaches using test population who are novices in programing. (See 'Expanding beyond the Scope of This Research' section for more detail)

Note on language

Throughout this thesis such terms as *computation* and *algorithmic* design/modelling are used frequently. As outlined in the introduction, within the field of digital design the term *algorithmic design* refers to the use of programming languages and procedural techniques to solve a design problem (Leach, 2010). Computation is a term which refers to the

use of mathematical or logical methods (the procedure of calculating) in the design process (Terzidis, 2006).

Algorithmic design is closely related to the concepts of parametric design, in many ways parametric and algorithmic can be seen as a synonyms (Davis, 2013). The term parametric is used in a variety of disciplines and it means working with parameters within a defined range (Leach, 2010). Parametric design is based on the use of parameters (variables) and form-making rules as a driving force for the design process. Robert Woodbury states that parametric design enables the 'parts of design' to relate to each other in a coordinated way (Woodbury, 2010). As Daniel Davis (2013) notes in his thesis, in the book 'Elements of Parametric Design' Woodbury does not actually give a definition for parametric design. Currently, within the field of architecture the term parametric has a range of meanings and there are 'battles and misgivings' surrounding this term (Davis, 2013). To avoid controversy, such terms as parametric design and parametric modelling are used throughout this thesis mostly when discussing Design Patterns (or as Robert Woodbury (2010) describes them 'Patterns for Parametric Design'). However in order to have a consistent set of terms, this thesis predominantly uses the word algorithmic. (See 'Definitions' and 'Computation, parametric and algorithmic design in architecture' for more details regarding the terminology).

0.4 Aim

The central research question addresses this aim, and asks to what extent, and in which particular way does the reuse of abstract and case-based algorithmic solutions improve and support a designer's ability to learn and use algorithmic design systems, and help users to overcome

barriers associated with programming? As a part of this investigation this thesis tests Design Patterns, developed by Robert Woodbury (2010).

The objective of this research is to investigate the effect of each reuse method on the design process and the design outcomes. Does the reuse of abstract/case-based solutions help to overcome some particular types of difficulties more than other types? Do the approaches improve designers' performance in terms of their ability to use computation as a means to translate design concepts into algorithmic models? Do the abstract concepts or examples of particular design cases help learners to understand and adopt the principles of algorithmic thinking? Do these approaches support design exploration or supress it? Do they save time and effort in solving design problems? And does their integration into the design process lead to a better design performance and higher satisfaction with the results? Ultimately, the aim is to determine whether the reuse approaches are worth using or not.

The secondary objective of this study is to understand the strengths and weaknesses of each approach, and to investigate in what way each approach can potentially be improved.

After answering these questions, the thesis aims to suggest ways in which re use of knowledge can be integrated into design education and practice and whether it is likely to beneficial.

0.5 Thesis structure

This thesis is divided into seven chapters: the introduction (current chapter 0.); the background (1.), the methodology (2.), the results (3.), the expansion beyond the scope of this research (4.), and the recommendations (5.) and conclusion (6.) chapters.

Chapter one - Background (1.) is split into three main sections, and also contains a list of set definitions. The first section is 'Context of this study'. It discusses the opportunities and challenges of using computation and algorithmic design systems in architecture, expanding on the literature regarding the research problem, stated in this thesis. It discusses the types of barriers that designers face when they use algorithmic modelling systems and programming languages. This section also discusses different reuse strategies (reuse of programming artefacts) employed in programming practice. The second section chapter 'Abstract solutions' explains the patterns approach in design and computation. The third section discusses the theory behind the 'Case-Based Design' approach.

Chapter two – Methodology (2.) is split into three main sections: 'Methodology for comparing approaches', 'Evaluation of the approaches', and 'Statistical methods'. The first section explains in detail the research problem, aims, objectives, focus of the study, and the overall experimental set-up. It also outlines the adaptation of the Design Patterns and Case-Based Design to the experimental framework of this study. The second methodology section presents the detailed metrics (criteria) for evaluating the approaches. The third methodology section explains the statistical methods used in this study, including hypothesis testing and correlation analysis.

Chapter three presents the Results (3.) of this experimental study. It contains four sections. The first section presents the overall results of the study, focusing on the identified advantages and barriers that designers face when using algorithmic design systems in architecture. This section outlines the benefits of integrating the reuse of algorithmic solutions into the learning narrative and design process. The second section presents results focusing on the reuse of abstract solutions in algorithmic design, comparing the performance of the Design Patterns (DP) group with the control group. The third section presents results of the Case-Based Design (CBD) approach and compares them with the control group. The fourth section compares the performance of the DP group against the performance of the CBD group, and contains the summary of key findings

Chapter four talks about future research (4.) and contains the discussion of an expansion beyond the scope of this study. It outlines the strategies for testing the DP and CBD approaches on a group of architects who are more advanced in algorithmic design, and the potential of testing these approaches using textual programming languages. It is suggested that to improve some of the issues identified for the DP and CBD approaches a hybrid approach could be developed. This hybrid approach would incorporate the methods of both abstract and case-based solution reuse.

Chapter five is a recommendations (5.) chapter, and includes a proposal for setting up a course to teaching programming in design based on the lessons learned from this study, outlining the lessons as a bullet point list.

The final chapter (6.) is a Conclusion chapter. It concludes that the reuse of knowledge (abstract or case-based algorithmic solutions) can be integrated as a design support method and can significantly reduce barriers to using programming improving the ability of architects to use algorithmic design systems.

1. Background

1.1 Context of this study

Algorithmic design provides architects with vast opportunities but it also requires them to adopt a particular set of design principles and techniques (such as programming), which some find to be very challenging. These issues relate to the overall research problem of this thesis. Firstly, this chapter discusses the shift in design practice caused by the use of computer technologies. It expands on the opportunities and challenges associated with the use of computer-aided design and computation in architecture. As algorithmic design progresses using programming languages, it belongs to the fields of both architecture and programming. In many ways architects, who create algorithmic design models share similar challenges as software engineers who create computer programs (Davis, 2013). We can learn from programming research and practices. Secondly this chapter discusses typical barriers associated with learning and using programming methods and expands on the knowledge reuse approaches that software developers use in their design practice.

Background: CAD in architecture

The conception of the twenty first century's architectural design is strongly linked with computer technology (Martens, Koutamanis, Brown, 2007), and our current 'architectural design culture is being explored through new

digital techniques' (Leach, 2009). CAD tools have settled themselves as a primary design platform in the field of architecture. Computation appears to be one of the most rapidly developing technologies of architectural design. It provides a unique means for architects to translate an idea into a form through the implementation of a simple set of operations and parameters which can link the form to wider social, aesthetic, political and environmental relationships. Inevitably, CAD technology expands beyond being only an aid of the design process and affects the process itself (Shih, Williams, Gu, 2011). This new logic of translating an idea into form can facilitate the emergence of novel principles of design thinking (Matcha, 2007).

Existing research in this area explores the future possibility of CAD tools that are able to learn; tools that have the ability to recognise, improve and apply appropriate knowledge to relevant problems (Gero, 1996). According to Professor Kalay, the primary use of computers in the building industry had already shifted two decades ago from the evaluation of proposed design solutions to visualization and collaboration among the various professional disciplines that operate within this industry, for example: architects, engineers, quantity surveyors et al (Kalay, 1999). Other studies envisage that future users of CAD for architectural design will require tools that allow them to work collaboratively and synchronously (Reffat 2006). Reffat suggests that CAAD (Computer-Aided Architectural Design) processes will be performed within smart and real-time 3D virtual environments and that the computer can be used as a 'metaphoric machine' adopting the role of the generator of chances (Ibid). Huang's 2009 paper 'Technology in Computer Aided Architectural Design' discusses the relationship of 3D modelling, BIM, IFC and CAAD network technologies. Huang states that 3D geometric modelling, BIM and CAAD

networks have an additive relationship between each other and that the future potential for CAAD lies within these relationships (Huang, 2009).

Polar opinions coexist within architectural society regarding the relationship between the use of computers and creativity (Bonnardel, Zenasni, 2010), (Jonson, 2005), (Musta'amal, 2010). On one hand, there is a commonly expressed opinion that the shift from conventional manual drafting to CAD modelling has improved design creativity (Chen, 2007). It is also believed that CAD tools are able to accommodate a wide range of users: from those developing quite simple product design to more sophisticated and complex designs solutions (Zeid, 2005). However, some architects suggest digital tools can limit or even suppress a designers' ability (Shih, Williams, Gu, 2011). Some research recognises that, while CAD inflicts certain limitations on architects, it also offers powerful opportunities. To the inexperienced CAD user these opportunities can also present a danger. It has been suggested that CAD models may be more readily accepted as finished designs without an appropriate level of critical development. (Walther, Robertson, Radcliffe, 2007).

There is a belief expressed by some that CAD is less effective, particularly during the initial ideation stage (Mora, Bédard, Rivard, 2008) (Mallasi, 2007), when an architectural concept does not have a certain form (Cao, Protzen, 1999). Similar opinions suggest that CAD is only appropriate for the post-development stages and should be used for refining a final proposal. "Its value as a development tool is extremely limited" (Charlesworth, 2007). According to Dorta, CAD tools still cannot support ideation in the way they should. He suggests that computer technology fails to compete with hand sketching and modelling during conceptual design stages (Pérez, Dorta, 2011) (Dorta, Perez, Lesage, 2008). The experimental set-up of this study addresses these issues. This research was

organised in such a way that throughout the course of the workshops each participant of this study had to produce at least two conceptual design models.

Design models in architecture

Digital design in architecture progresses as the architectural model progresses. Architects use models as a thinking and defining mechanism for understanding and presenting architectural ideas (Smith, 2004). Virtual models are basically the sets of coded information that exist within the virtual realm, and operate through computer media. Digital files and data can be exported from one program to the other, thus providing direct exchange of often complex and precise information. Constant dialogue between software and virtual models, provided by computer-aided technology, creates an effective and powerful multifunctional digital design platform. Digital fabrication has triggered a design revolution, in particular promoting innovative and inventive work in the field of architecture (Iwamoto, 2009). With rapid technological development in the field of CNC fabrication, computer-aided design has evolved from pure virtuality to a more complex tool, which blurs the boundary between matter and space (Andia, 2001). In this context, digital fabrication appears to be a logical extension of computer-aided technology to the material world and therefore to the field of computational design in architecture.

In the work 'Material Computation: Higher Integration in Morphogenetic Design' Achim Menges (2012) states that the production of architecture is on the verge of a significant change. The author predicts that in the near future we will witness a new degree of integration between computational design and the physical realisation of architecture. Material

characteristics and behaviour provide means to inform the design process. The reality of the physical constraints of the material world, its self-organisation, and structuring mean that there are limits to what is actually possible (Ball, 2011). However, architecture, being a material practice, is still broadly based on the design approaches that are not primarily focused on the characteristics and performance of the materials (Menges, 2012). Architectural design, especially during the early conceptual stages, is usually materially abstract. It often progresses through geometrical form-finding, the results of which have passive material properties automatically assigned. Yet the characteristic of such material as wood for example, can suggest amazing design opportunities and structural solutions (Ibid). Algorithmic modelling has been proposed as an enabler of parametric form-finding approaches, which also consider functional aspects, and structural properties and behaviour (Baerlecken et al, 2010).

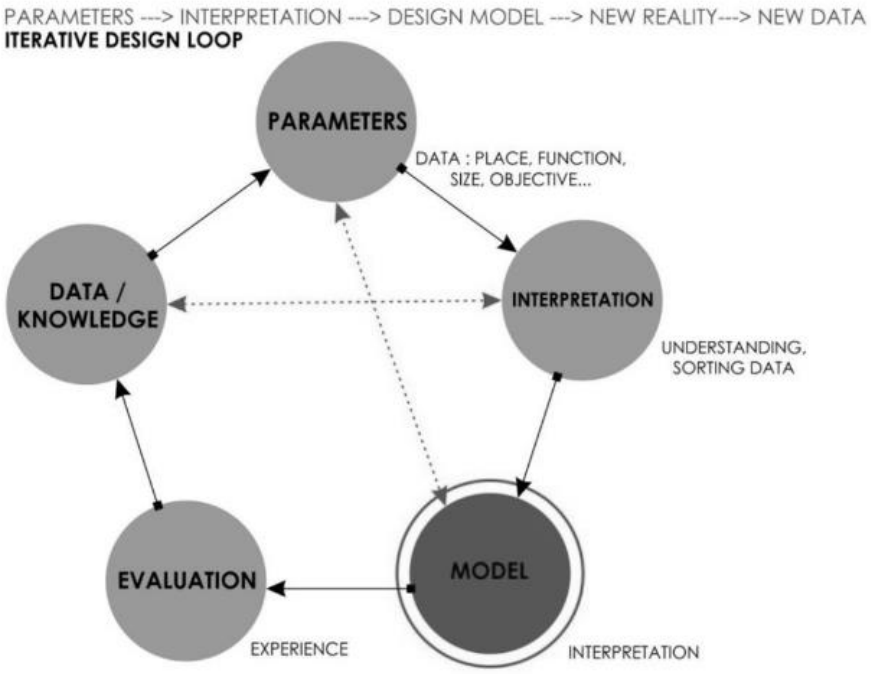


Exhibit 1.1 Iterative Design Loop.

Architectural design is not a linear process; it often involves repeating loops and iterations (Berkel, Bos, 2006). In order to evaluate their work, architects have to have a 'reality check', which design models often provide. Design models contain the very core ingredients of architecture, functioning as a set of compositional, organisational, and structural principles and parameters (Ibid). The architectural design process could be described as a series of loops (Exhibit 1.1). Each loop involves interpretation of relevant objectives and parameters, and further translation of these instances into architectural models. The model, being a physical or virtual representation of an idea, becomes a reality itself, and serves as a source for experience and exploration. Perceived information, interpreted into a new, updated set of parameters, triggers further model development. This design loop can iterate an infinite amount of times, in a never ending search for the most fitting solution. The quality of design outcome along with other conditions highly depends on the diversity of parameters.

It is important to understand the difference between computation and computerisation. Computerisation and computer-aided approaches refer to utilising computers for organising information (containing and representing information) (Menges, Ahlquist, 2011). Computational approaches (including parametric and algorithmic design methods) allow production of new data, by deducing results from values and actions using programming algorithms (Ibid).

Computation in architectural design can have a profound influence on how the form is perceived and how the output form and structure are envisioned and produced (Menges, Ahlquist, 2011). Computational design techniques, such as the use of programming algorithms as a model making (form-generating) method, have an immense effect on the way designers

and architects think and act (Benton, 2007). The design process is no longer a straight transition from an abstract idea to a design model through a direct manipulation with the form. In form-making morphology, there is a turning away point from end products composed of simple fixed structures towards dynamic, ever-changing processes (Kwinter, Davidson, 2008). The use of computational design systems in architecture triggered the shift in representation and design thinking from object-oriented models to 'dynamic system' models (Menges, Ahlquist, 2011).

Computation introduces a different level of design construct, which operates through the use of form-generating programs (algorithms). The emergence of algorithmic form-generating design tools led to a fundamental change in architectural morphologies, increasing the opportunity to create 'innovative smart geometries' (Abdelsalam, 2009). Computational design approaches make it possible to generate specific design outputs from the 'initial abstraction' through the use of a programming algorithm which contains parameters and actions (Coates, 2010). These approaches re-define the role of a design model in architecture. In computational design a model is no longer an object, but it is an integral part of a dynamic design process, fluid and responsive to changes in the input parameters and programming logic. In computational design architects design process instead of designing objects. There is a profound shift in design thinking and methods caused by the current transition from Computer-Aided Design to computation (algorithmic design), from crafting of objects using design software towards the development of dynamic algorithmic systems (Menges, Ahlquist, 2011).

Parametric and algorithmic design in architecture

Algorithmic design systems have become the subject matter of much research recently. Architectural design rapidly and readily shifts from the concept of static (fixed) forms to dynamic forms, defined by interdependencies of forces and geometrical constraints (Menges, Ahlquist, 2011). Some researchers suggest that algorithmic modelling tools allow the creation of new and original design solutions that are difficult or impossible to achieve via other methods (McCormack, Dorin and Innocent, 2004).

In design and architecture such terms as parametric design and algorithmic design are closely related. They refer to the computation driven design processes that progress through the use of programming algorithms, defined by rules and parameters (variables). In computation, the organisation of an architectural form (object) can be perceived as an assembly of parts, which are defined by constraints of form-making rules and negotiations between architectural primitives and the external forces (Menges, Ahlquist, 2011). To use algorithmic modelling it is fundamental to understand how the system operates and how the form and programming constructs work together (Ibid). To understand these operations it is essential to be able to predict the behaviour of a computer model, which represents the system. The success of this process depends on the architects' ability to define and organise the system and its parameters; and their ability to inform and further improve systems' behaviour, using prediction and feedback from the model (Ibid).

Parametric design uses parameters and rules to express and define the relationship between the design idea, constraints, form-making logic; and the resulting design behaviour. Parametric design can be defined as a series of questions, which establish the variables (parameters) of a design

and a computational algorithm that can be used to produce a variety of outcomes (Karle, Kelly, 2011). The deeper understanding of parametrics allows a designer to establish a method connecting the behaviour of forces and forms and representing them as mathematical algorithms and geometric rules (Woodbury, 2010). Parametric thinking requires a designer to establish clear relationships by which the design parts connect, rather than creating the design solution directly. To achieve that, one has to step back from direct manipulations with forms and concentrate on building the logic of the design (Woodbury, 2010) (See 'Problems with algorithmic design' section).

Robert Woodbury, (2010) in his book 'Elements of Parametric Design' states: "Parametrics is more about an attitude of mind than any particular software application". He notes that parametric design requires a very specific way of thinking, which some designers may find alien. He argues that parametric modelling systems simply combine basic ideas from geometry and computer programming. It turns out that these basic ideas do not appear so easy to grasp for people with typical design backgrounds. In order to master parametric design techniques one has to be part-designer, part-computer scientist and part-mathematician. Woodbury argues that all CAD models are sets of mathematical propositions. Therefore, in some sense, designers 'do' mathematics. Designers seldom look at CAD modelling from this perspective, and they more 'use' mathematics than actually 'do' mathematics (Woodbury, 2010). It can also be argued that learning algorithmic design in architecture can enhance education, as it allows students to better understand how to de-code complex structures and concepts (Howe, 2011).

When defining algorithmic and parametric design Neal Lech states that within the field of digital design, the term parametric design refers

broadly to the use of parametric modeling software (2010). According to him, algorithmic design refers to the use of programming languages that allow to design through the direct manipulation not of form but of programming algorithm (Ibid). However most of parametric modelling software progress using programming languages; and parameters (variables) are often utilised in scripts and programming definitions, which makes the resulting solutions both algorithmic and parametric.

In the context of this study 'algorithmic design' is identified as the most fitting term, because the objective of this study is to investigate the ways to support 'idea-to-algorithm' translation and to assist the use of programming algorithms in architecture. Therefore, the focus is on designing through the use of programming algorithms. In this respect algorithmic thinking and algorithmic design describe the topic of this research most accurately.

Barriers in end-user programming systems

Algorithmic modelling progresses using visual or textual programming languages. Architects and designers often face substantial difficulties with adopting programming logic and syntax (Celani and Vaz, 2012); (Woodbury 2010). The initial principles of human and computer reasoning are often alien to each other. Many designers, who are novice to programming, struggle to overcome barriers associated with the use codes and algorithms. They often find it difficult to use algorithmic design thinking and programming techniques as a part of their design process (Woodbury, 2010). It is also problematic for architects to master algorithmic design logic, because the practice of architecture is associated with 'artistic sensibility and intuitive playfulness', whereas a programing

algorithm is perceived as 'non-human creations' (Menges, Ahlquist, 2011). There could be numerous reasons why architects and designers struggle to acquire computational thinking mode and to master programming languages. The fact remains: end-user programmers have to overcome substantial barriers in learning and using programming systems (Ko, Myers, Aung, 2004). The aim of this study is to investigate whether the re-use of algorithmic solutions can help designers reduce these barriers.

Research on learning barriers in programming systems has identified six types of most re-occurring barriers: design, selection, coordination, use, understanding, and information (Ko, Myers, Aung, 2004). Ko et al. define learning barriers as programming problems that lead to invalid assumptions, preventing the end-user from achieving the progress. In programming languages the common causes, which often lead to invalid assumptions, include the use of: conditions, loops, data structures and language constructs (Pane, Ratanamahatana, Myers, 2001) (Engebretson, Wiedenbeck, 2002) (Ko, Myers, Aung, 2004). The experimental study that Ko et al. conducted observed 40 participants who learned programming with Visual Basic. NET (VB) during the five week 'Programming Usable Interfaces' course. To understand learning barriers their study focused on the behaviour and progress of the learner.

The focus was on 'insurmountable' barriers, which learners could not overcome (understand and fix) despite considerable effort (Ko, Myers, Aung, 2004). The first type of programming barriers was identified as design barriers: 'I do not know what I want the computer to do' (Ibid). Design barriers refer to the cognitive difficulties and represent user's inability to realise the idea-to-programming algorithm translation. The second type of barrier was selection barriers, articulated as: 'I think I know what I want the computer to do, but I do not know what to use' (Ibid). It

proved to be difficult for some users to locate those programming artefacts (commands/programming components) that performed a particular action. Ko et al. indicate that the majority of users eventually managed to overcome these selection barriers by using the code examples of their peers. The third type of programming barrier is coordination barriers: 'I know what to use, but I do not know how to make them work together' (Ibid). These difficulties were also labelled as 'invisible rules' and covered such problems as knowing how to organise, structure and coordinate a set of programming artefacts. Use barriers were identified as the fourth type of barriers. They can be explained as: 'I think I know what to use, but I do not know how to use it' (Ibid). The fifth type is understanding barriers: 'I think I knew how to use it, but it did not do what I expected'. The understanding barriers occurred when there was a mismatch between expectations and the program's actual behaviour, or when a program returned an error message and learners could not figure out why. The last type of barrier associated with learning programming environments was identified as information barriers: *'I think I know why it did not do what I expected, but I do not know how to check'*. The authors state that information barriers occur due to the fact that it is often difficult to acquire information about the internal behaviour of a program. When learners came across information barriers their typical strategy was to try and guess what statement caused the problem.

Ko et al. argue that while experienced programming users do face certain types of difficulties, they are able to easily overcome barriers associated with selection, coordination and use (Ko, Myers, Aung, 2004). However, according to Ko et al., experts often face significant difficulties caused by understanding and information barriers. Learners could easily understand data and principles of programming logic. However they had major difficulties in trying to act on it (the actual implementation of

programming). In the conclusion section of 'Six Learning Barriers in End-User Programming Systems' Ko et al. state that the use of examples (case-based reasoning) can potentially improve user's ability to overcome some of the barriers including, design, coordination and use barriers. One of the research objectives of this thesis is to investigate whether these claims (stating that the case-based reasoning helps to overcome programming barriers) are valid in the context of algorithmic design in architecture.

Software reuse

Methods of knowledge re-use are often used in programming practice and education as a way to help software engineers to overcome programming barriers make the design process more efficient. The reuse of programs and codes (software artefacts) is an important part of programming practice and research in the field of software design (Krueger, 1992).

The paper 'Software reuse' by Charles Krueger discusses different types of software reuse techniques, which are employed in software design (Ibid). Krueger quotes Biggerstaff and Richter (1989) and states that all reuse approaches involve four instances: abstraction, selection, specialisation and integration of software artefacts. According to Krueger Abstraction plays an essential role in any reuse technique, because without it software developers would be most likely lost in the vast collections of reusable artefacts. There is a strong relationship between abstraction and reusability; they are in fact 'two sides of the same coin' (Krueger quotes Wegner, 1930). In software reuse Abstraction helps to determine what each artefact does and when and how it can be applied (Krueger, 1992). There are strong parallels to the pattern approach to architectural design

proposed by Alexander (Alexander, 1975), who identified the key principles of design patterns structure as: what to use, when to use and how to use (Alexander, 1975). Selection plays an important in any reuse approach as it helps to locate, compare and select reusable items (Krueger, 1992).

Classification of reusable artefacts is used as an example that can help to organise a library and guide the search and selection process (Ibid). It should be noted that classification or grouping of reusable objects can often require abstraction. Many reuse approaches merge similar solutions (artefacts) into one generic reusable solution, as for example in the TRIZ method (Stratton, Mann, Otterson, 2000). To reuse a generic solution, software designers need to specialise it by changing its parameters and constrains to suit a new design context (Krueger, 1992). Specialisation of a reusable solution is almost inevitable as only in rare cases is it possible to find an artefact that can be reused directly, without any modifications and alterations. Specialisation applies to the reuse of abstract solutions such as generic schemes and design patterns; and it also applies to specific solutions such as codes (scripts) and visual programming algorithms. The final instance, which is involved in almost all reuse approaches in programming, is Integration. Integration is a framework, which helps to combine a number of located and specialised reusable artefacts together (Ibid). This is very similar to the idea of using design patterns as building blocks in order to create more complex design solutions (Alexander, 1975) (Gamma, Helm, Johnson, Vlissides, 1994) (Woodbury, 2010).

In his 'Software Reuse' survey Krueger describes and compares eight different reuse techniques. The list is sorted according to how well each technique minimises the intellectual effort required to use them (cognitive distance) (Krueger, 1992):

- High level languages (Programming languages with strong abstraction from the details. The reusable artefacts in a high-level language are the assembly patterns.)
- Very high-level languages (Goal-oriented programming languages with a very high level of abstraction)
- Application generators (High-level systems, with often have a high level of abstraction, that generate application programs, by reusing software system designs.)
- Software architectures (High level reusable structures that capture a software system design, focusing on subsystems and their interactions. Analogue to the large-scale software schemas)
- Transformational system (Transformational systems often have a very high level of abstraction. It takes one program and through a series of transformations generates from it another program)
- Software schemas (The goal of schema is to capture and reuse abstract algorithms and structures rather than reusing the code itself.)
- Source code components (The reusable artefacts are the 'off-the-shelf source code components', which are organised and categorised in a catalogues or libraries of components)
- Design and Code scavenging (The reusable artefacts in scavenging are code fragments (scripts and algorithms), copied from existing systems.)

  (Krueger, 1992)

The outlined types of the software reuse techniques are developed specially for designing software systems. However there are strong similarities between the categories, proposed by Krueger for software reuse and the reuse approaches identified for algorithmic design. The 'Design and code scavenging' and 'Source code components' refer to the

reuse of specific programming solutions. The rest of software reuse techniques, such as 'Software schemas' and 'Software architectures' refer to the reuse of solutions with a high level of abstraction, i.e. abstract programming solutions.

Krueger defines software reuse as a process of using existing programming artefacts instead of building them from scratch (Ibid) He emphasises that typically every reuse technique involves selection, specialisation and integration of artefacts, though the degree of involvement may vary depending on the reuse technique. The objective of the reuse of programming artefacts is to reduce time and effort required to design software systems. According to Krueger, an effective reuse technology implies the use of high level of abstraction (Ibid). Meaning that a designer should know 'what' the reusable artefacts do rather than 'how' they do it. However, the author points out that there are difficulties associated with the reuse of abstractions. As in order to use abstract solutions a designer must be familiar with the abstractions prior the design process, which requires time to study and understand these abstractions. The study concludes that for a reuse technique to be effective:

- It must reduce an intellectual effort required to reuse artefacts (abstract or specific programming solutions);
- it must be easier to reuse an existing artefact (solution) than it is to develop a new system from scratch;
- a designer must know 'what' a solution does, to be able to select it for reuse
- a designer must be able to find it faster than he/she can build it; (Krueger, 1992)

All mentioned above aspects of the reuse methods apply to both the Design Patterns (DP) and Case-Based Design (CBD) approaches. Both

of these approaches can be described as methods reusing programming artefacts. The difference is the degree of abstraction of these artefacts (solutions). The DP approaches is at one end of the spectrum, representing the reuse of an abstract generalised idea (construct), while the CBD approach is at the other end of the spectrum, representing the reuse of a very specific solution (existing within a particular design context).

1.2 Abstract solutions in design and computation

This thesis uses patterns as a means to test the reuse of algorithmic solutions with a high level of abstraction in the field of architecture and design. The Design Patterns method was adapted and tested in various other disciplines including the architecture, design, human-computer interaction, software design, object-oriented design and participatory design.

Design Patterns

The idea of Design Patterns was introduced by the architect Christopher Alexander. His work "A Pattern Language: Towns, Buildings, Construction" (Alexander, Ishikawa, Murray Silverstein, 1977) has greatly influenced the subsequent studies of the subject and was adapted for various disciplines, such as: landscape design, product design and computer science. According to Christopher Alexander each Design Pattern describes a problem which occurs over and over again (Alexander, Ishikawa, Murray Silverstein, 1977). The pattern describes the core of the solution to the problem, so this solution can be used a million times over, without ever doing the same thing twice (Ibid).

The systematic approach proposed by Christopher Alexander is widely referenced and used. This approach outlines the following principles of writing a Design Pattern:

- Decomposition of the problem into sub-problems;
- Generating an abstract solution to a global problem by synthesising the individual solutions;
- Giving a name and a reference number to the pattern;
- Providing an image and a description of the context and problem of the pattern;

  Including a diagram which illustrates the solution. (Ibid).

An architecture example of a design pattern developed by Alexander et al. (Ibid) as a part of the Pattern Language can be the 'Main Entrance' pattern:

Name: Main Entrance

Context: You need to fix the entrance of the building

Consider these patterns first: Circulation Realms, Family of Entrances

Problem: 'Placing the main entrance is perhaps the single important step you take during the evolution of a building' (Ibid)

Solution: 'The entrance must be placed in such a way that the people who approach the building see the entrance as soon as they see the building itself' (Ibid). The two steps the solution are: 1) position the main entrance correctly, so it can be seen immediately from the street; 2) make it clearly visible (a shape that stands out in front of the building).

Consider next: Entrance Room, Entrance Transition, Shield Parking, Car connection

Terms related to the reuse of abstract solutions

The following list of terms and term explanations refer to the core concept of the first approach: the reuse of abstract solutions in design and architecture, tested by this study as a means to aid algorithmic modelling. It is important to point out that all these terms were originally used in the contexts where the reuse of an abstraction (the core principles of a certain type of solution) plays an important role in the design process or inventive problem solving. Some of the terms may seem rather distinct, for example 'Design Patterns', 'Abstract Solutions' and 'Generic Solutions'.

The term 'Abstract Solutions' has frequently been used to describe the idea of Design Patterns (Alexander, 1975) (Gamma, Helm, Johnson, Vlissides, 1994) (Woodbury, 2010) and the term 'Generic Solutions' describe the TRIZ solution system, based on the principle of abstraction (Altshuller, 1988) (Terninko, Zusman, Zlotin, 1998). Both of those terms articulated the idea that recurring types of designs (solutions) can be reused effectively through the abstraction of the core of this design (solution) and applying it in the new context. Hence 'Abstract Solutions' and 'Generic Solutions' describe the same underlying principles and ideas, even though they were originally utilised by authors who worked in different fields of knowledge.

The differences in these terms, may have also occurred due to translation issues, as the TRIZ theory (Altshuller, 1988), was originally written in Russian. When talking about the reuse of abstract design solutions for a problem, Altshuller frequently used such terms as 'Standard' (Standard for solving inventive problems) and 'Standard 'Formula'' (the terms are translated from Altshuller's manuscript, written

in 1975 (Altshuller, 1975). In my own reading (as a native speaker of Russian) of the original Russian Alshuller text, the meaning of the term 'Standard' is very similar (and almost identical) to the meaning of a term 'Design Pattern' used by Alexander, Gamma and Woodbury (Alexander, 1975), (Gamma, Helm, Johnson, Vlissides, 1994), (Woodbury, 2010) (See more details in Theory of Inventive Problem Solving (TRIZ) Section).

The terms, listed below, vary depending on the particular aspects of the context, intake and interpretation of the authors, but all refer to the same fundamental concept: the reusable abstracted design solution.

Abstract Solutions (Abstraction) (Alexander, 1975), (Gamma, Helm, Johnson, Vlissides, 1994), (Woodbury, 2010)

Typical Solution, Category/Class of the solutions (Gamma, Helm, Johnson, Vlissides, 1994)

Standard, Generic Solution, Standard Solution, 'Formula' (Altshuller, 1988) (Terninko, Zusman, Zlotin, 1998), (Woodbury, 2010)

Design Patterns, Patterns (Patterns for Parametric Design) (Woodbury, 2010), (Alexander, 1975), (Gamma, Helm, Johnson, Vlissides, 1994)

- Design Pattern is an abstract solution, which can be applied to a shared problem (Woodbury, 2010).
- Interpretation of the design idea/concept (Woodbury, 2010)
- Pattern is a 'pre-formal construct', which describes the forces in the world and relationship between them (Lea, 1994);
- Patterns emerge from repetitions of human behaviour (Coad, 1992);
- Pattern is a recurrent phenomenon or structure, 'didactic medium for human readers' (Borchers, 2001);
- Pattern describes a problem and then describes the core of the solution (Gamma, 1994 quote Alexander (1977)).

- Pattern is a structured description of invariant solution. Invariant refers to a set of shared characteristics of the recommended solution (Winn, Calder 2002)
- Patterns should capture 'big ideas' (Winn, Calder 2002) instead of covering every possible design decision.

Pattern is an abstraction, which describes not some specific example, but it rather refers to a general concept or idea, which is often associated with vagueness. In computer science, an abstraction characterizes a class of instances which omits inessential details (Woodbury, 2010) (Gamma, Helm, Johnson, Vlissides, 1994).

Design Patterns are the medium to understand and express the practice craft of parametric modelling (Woodbury, 2010)

Studies based on the Design Patterns approach

This thesis tests the Design Patterns approach in the context of algorithmic design, which relates equally to the fields of architecture, design and programming. While originally pattern study was developed in the field of architecture (Alexander, 1975), the idea of design patterns and pattern languages was widely adopted in the computer sciences, such as programming, software design and human-computer interactions (HCI) (Gamma, Helm, Johnson, Vlissides, 1993), (Dearden, Finlay, Allgar, Mcmanus, 2002). Patterns research has been very successful and has many 'practical applications and benefits' in the field of software engineering (Lano, 2014)

In the early 1990s, software engineering researchers started to explore the means to reuse design knowledge (Coplien, Alexander, 1996); (Garlan, Delisle, 1990); (Gamma, Helm, Johnson, Vlissides, 1993). In 1994 the first conference on 'Pattern Languages of Programming' was organised. It was

followed by further conference series investigating pattern languages in software engineering. One of the important publications in this field was the book 'Design Patterns: Elements of Reusable Object-Oriented Software' by Gamma et al. (1994).

Here are some of the various pattern definitions, given by different authors, discussed in Gamma's paper:

- Pattern is a 'pre-formal construct' (Lea, 1994);
- Patterns emerge from repetitions of human behaviour (Coad, 1992);
- Pattern is a recurrent phenomenon or structure, 'didactic medium for human readers' (Borchers, 2001);
- Pattern describes a problem and then describes the core of the solution (Gamma, 1994 quote Alexander (1977)).

Design Patterns: abstraction and reuse of object-oriented design

A theoretical study inspired by Alexander's work 'Design patterns, Elements of Reusable Object-Oriented Software' uses design patterns as a mechanism for the analysis, systemisation and reuse of knowledge in the field of computer science and software development (Gamma, Helm, Johnson, Vlissides, 1994). Object-oriented design is the approach to solving a software problem by treating it as a system of interacting objects. The authors use design patterns as a medium to express the design solution by identifying the 'objects' (data and procedures) and establishing their collaborations and responsibilities. The role of the patterns in this case is to reduce the complexity of a system by identifying abstractions and to

act as the reusable building blocks from which the compound software solutions can be composed (Ibid).

Gamma et al. (Gamma, Helm, Johnson, Vlissides, 1994) establish the principles of design patterns, and develop a pattern catalogue which composes the major part of their book (Ibid). The authors argue that the key identifier of an experienced designers' success is that they do not try to solve every problem from first principles; rather they reuse solutions that have worked for them in the past. This way they can apply existing patterns again and again without rediscovering them. Their study (Ibid) identifies four essential elements of a design pattern: the pattern name, which describes a problem at a high level of abstraction; the problem, which describes when to apply the pattern; the solution, which is an algorithm of actions; and the consequences, the results and trade-offs. The design patterns discussed in their book are descriptions of objects that solve a general design problem in a particular context.

In the earlier paper 'Design Patterns: Abstraction and reuse of Object-Oriented Design' Gamma et al. (Gamma, Helm, Johnson, Vlissides, 1993) describe the use of design patterns as a mechanism to capture design intent in the field of object-oriented software design. The authors stress the importance of abstract design (as opposed to a particular design) and state that it is the essential part of any design pattern. Though Design patterns may specify potential implementation details, they are supposed to have an adequate level of abstraction to ensure their wide applicability.

Gamma et al. tested the use of design patterns in the context of object-oriented software design using two tools: 'ET++SwapsManager' (Eggenschwiler, Gamma, 1992) and 'QOCA: A Constraint Solving Toolkit' (Marriott, Chok, 2002). They have observed a number of positive effects induced by the reuse of abstract solutions (design patterns):

- reduce the effort required to learn new software;

- help during design development and code review stages;
- help explore alternative design solutions;
- motivate 'to go beyond concrete objects'
- when patterns are introduced together with examples, it works out as an effective way to teach object oriented design by example (case-based design strategy)
  (Gamma, Helm, Johnson, Vlissides, 1993)

Design Patterns in participatory design

One of the advantages of the abstractions is that design patterns provide 'reusable models that can be instantiated across different domains' (Ramirez, Cheng, 2010). A number of studies discuss the benefits and challenges of the reuse of abstract solutions, through the use of design patterns and pattern languages in interdisciplinary and cooperative design projects (Woodward, 2010), (Dearden, Finlay, Allgar, Mcmanus, 2002), (Dearden, Finlay, 2006). Even though these works were done outside the context of algorithmic design in architecture, landscape and industrial design the findings and discussions raised by these studies are relevant to this thesis, as they outline the potential strength and weaknesses of the method. The reviews also reflect on: What is a design pattern? How patterns can be used? And how pattern-based approach influences design process? (Ibid)

The paper 'An Interpretation Design Pattern Language: A Propositional Conceptual Tool for Interdisciplinary Team Members Working on Interpretation Design Projects' (Woodward, 2010) introduces a 'pattern language' methodology, which is based on Alexander's pattern language. It proposes a new, shared language for interdisciplinary teams working on interpretation design projects. This designer-led Interpretation

Design Pattern language aims to improve the collaboration between designers and professionals from the other fields of research and practice. The author states that the pattern finding methodology is an appropriate and suitable method to group and sort data. Woodward draws parallels between pattern-finding and design research and practice, which often focus on 'problem finding' and 'problem solving' approaches. The research concludes that the Interpretation Design Pattern language does not provide ready-made solutions or answers, but it may trigger new strategies of interpretation, which is suggested by the insights from an extended range of disciplines (Ibid).

The work 'Using pattern languages in participatory design' (Dearden, Finlay, Allgar, Mcmanus, 2002) explores the potential of using pattern languages as tools within design processes in the field of Human Computer Interaction ((HCI) interaction between people and computers). Participatory or cooperative design is a design approach which involves active work of multiple types of participants, such as designers, developers, employees, customers, users and so on. The authors mention that Alexander originally developed the philosophy and concept of pattern languages in the radical scope of cooperative (participatory) design. In the Oregon Experiment Alexander and his colleagues state that all the decisions of what and how to design and build should be in the hands of the users (Alexander, 1975). They also point out that every part of a good environment should be highly adapted to its particularities. And that this adaptation can only be successful if people do it themselves.

It is recognised that in participatory design within Human Computer Interaction, studying a human and a machine in conjunction, it is vital to write patterns in such a way that users will be able to comprehend design patterns (Dearden, Finlay, Allgar, Mcmanus, 2002). Nevertheless, there is

an opinion that it becomes more evident that the main goal of pattern languages has shifted towards sharing knowledge between professionals, while allowing users only to critique and participate in discussions (Borches, 2001).

Issues related to the use of design patterns

In software engineering, patterns tend to be interpreted as the preliminary abstract relationships between context, problem and solution. The actual examples (physical presentations) of the pattern are usually seen as elements of secondary value. Dearden et al. (2002) argue that, in the context of participatory design, this viewpoint is not valid and cannot be sustained. The observations indicate that users often search for specific remembered patterns, while browsing the language (Dearden, Finlay, Allgar, Mcmanus, 2002).

Other findings indicate that users subconsciously 'trusted the patterns' and considered them to be 'correct' by default (Dearden, Finlay, Allgar, McManus, 2002). The authors, who actually developed these patterns, on the other hand, state that they cannot really claim that they (themselves) trust their patterns in their present form (Ibid).

In a Pattern Language critical review, Dearden and Finlay (2006) examine the history of patterns and pattern languages in HCI. The work aims to locate design patterns in relation to other interactive design approaches. This research states that recently patterns and pattern languages are getting more and more attention in HCI for their potential in supporting the design process and recording and communicating design knowledge. This study identifies the following established and emerging techniques adopted by interactive systems:

- Guidelines for Designing and heuristics (Mosier, Smith, 1986) (Nielsen, 1994);
- Style-guides (Gnome project, 2003);
- Participatory design (Schuler, Namioka, 1993) (Muller, Haslwanter, Dayton, 1997);
- Claims analysis (Sutcliffe 2000);
- Design rationale (MacLean et al., 1991);

Talking about the history of patterns, the authors report that the work of Christopher Alexander and his colleagues provoked controversy within the architectural profession. Though it was criticised (Dovey, 1990); (Saunders, 2002) this work has been very influential in the field of architecture and several other domains (King, 1993); (Gabriel, 1996); (Saunders, 2002).

The authors state that in HCI and software engineering, the term 'pattern' stands for a structured description of an invariant solution. Invariant here refers to a set of shared characteristics of the recommended solution. One of the distinguishing characteristics of patterns is that they are rooted in practice, rather than theory. Patterns should capture 'big ideas' (Winn, Calder 2002) instead of covering every possible design decision. Patterns also should have a timeless quality, thus be applicable, regardless of a particular platform or technology. The authors (Ibid) argue that this is probably the weakest spot in many interaction design patterns. It is only possible for a pattern to be timeless when it is written in a high level of abstraction (Bayle et al, 1998); the more detailed a pattern is the more it is necessary to reflect on a particular technology and platform characteristic. That is why, when writing a pattern, it is important to find an appropriate degree of abstraction. If a pattern is too abstract it will be not

efficient in real design practice and if it is too specific it will be hard to reuse it in new scenarios.

It is 'difficult to formalise' and describe patterns, as well as organise their selection and application methods (Lano, 2014). Ramirez and Cheng (2010) state that in practice it is difficult to harvest design patterns, because a) there is no 'standard methodology' for creating patterns; and b) there is no metrics for the evaluation of the resulting patterns.

Theory of inventive problem solving (TRIZ)

The use of abstractions and reusable items (solutions) in design and problem solving is common for mathematics, programming, engineering, design, architecture and other disciplines. This thesis tests the reuse of abstract solutions through an example: the Thirteen Patterns for Parametric Design (Woodbury, 2010). Nevertheless it is important to consider that the design patterns approach is not the only method that combines the reuse of design solutions and the use of abstractions. The investigation has revealed more examples of relevant works regarding the principles which lie behind the design methods based on the reuse of knowledge.

One of the works, which incorporates the principle of abstraction, patterns and the reuse of design solutions, is the TRIZ method (Altshuller, 1988) (Terninko, Zusman, Zlotin, 1998). This method is closely related to the concept of the design pattern approach and can be seen as its an alternative approach to use abstraction. TRIZ is a Russian acronym for 'Theory of Inventive Problem Solving'. It was developed by Genrich Altshuller et al. (1926 to 1998) as a methodology of problem solving and inventive thinking in engineering. It started as a study investigating whether there were any systematic patterns to inventive thinking (Stratton, Mann,

Otterson, 2000). Altshuller analysed over 200,000 documented inventions (patents) trying to identify the common sets of inventive principles and repetitive patterns, which afterwards were used to form the 40 principles of TRIZ. According to the TRIZ methodology 'an inventive problem can be classified and methodically solved, as any other engineering problem' (Ibid). In architectural design we usually think in terms of idea development, rather than problem solving (which refers mainly to the field of applied sciences), but the core concepts and logic behind these two processes have strong similarities. A number of design studies have implemented Altshuller's TRIZ methodology. It was adapted for product design, for example in 'A TRIZ approach to design for environment' (Serban, Man, Ionescu and Roche, 2004). In the context of this thesis, TRIZ method can be viewed as an alternative knowledge reuse approach.

In one his early works Altshuller introduces the idea of 'Standards', which he describes as a 'high method' to solve inventive problems (Altshuller, 1975). According to his Algorithm of Inventive Problem Solving, it is possible to identify a generic method (solution) to solve a certain type of inventive problem, by analysing the large masses of existing solutions. The identified generic method can be then translated into a 'Standard'. The idea of 'Standards' is very similar to the idea of Alexander's Design Patterns, which was introduced at the same time period (Alexander, 1975). Alexander states that a pattern describes a problem and then describes the core of its solution (Alexander, 1977). Altshuller describes a 'Standard' as an algorithm (method) solving a wide class of inventive problems on a 'high' (abstract) level (Altshuller, 1975). According to ARIZ each 'Standard' should contain:

- A 'Standard Formula', describing the core of its idea
- An Explanation and examples

- An Application of a 'Standard' (Ibid)

The principles and methodology for the use of Design Patterns and TRIZ have a large number of parallels and similarities. Firstly, both systems operate through the reuse of knowledge. Secondly, Design Patterns use the principle of abstraction to provide a generic solution for a problem. One of the organisation principles in the Theory of Inventive Problem Solving is the principle of generalised solutions (Altshuller, 1999). Thirdly, Design Patterns has the separation (segmentation) principle: in cases when the initial idea has a high degree of complexity, the project is divided into independent parts (Gamma, Helm, Johnson, Vlissides, 1994). The first principle of TRIZ is the 'Segmentation' principle. It is used to divide an object into independent parts, to make an object easy to assemble/disassemble or to increase the degree of fragmentation. Additionally, the 'Extraction' principle of TRIZ, used to extract (identify) a part or a property of an object, employs the abstractions and metaphors, which can be directly related to the idea of Design Patterns.

TRIZ methodology is heavily based on the use of knowledge bases and computer systems, which manage the knowledge. In TRIZ the search for the ideal design solution is associated with the reuse of available existing resources (solutions) (Bakar, 2014), (Stratton, Mann, Otterson, 2000). To provide a framework for the ever growing knowledge TRIZ tools employ organised knowledge bases. The data (solutions) in the knowledge bases is classified and sorted into various groups. Similar to the concept of the Design Patterns, TRIZ uses the principle of abstraction (Kaplan, 1996). According to TRIZ methodology, abstraction is needed to identify and classify the generic problem and solution. After that, the relationships (correlations) can be identified between the established groups of problems and solutions.

The use of TRIZ solution system for the inventive problems can be described as:

- Classify a specific problem, so it can be sorted into a generic problem category
- Use the established correlations (relationships) to find a generic solution category
- Use original thinking (specialisation) and a generic solution to develop a specific solution

  (Stratton, Mann, Otterson, 2000)

The idea of TRIZ is closely related to the idea of Design Patterns (DP), as both those methods are based on the reuse of generalised solutions. The TRIZ method progresses through the use of abstraction, which directly relates to the first approach, tested in this thesis. At the same time TRIZ employs the use of computer systems and databases. This method also heavily relies on the use of cases, as a driving force for the identification of a typical solution (Standard). In relation to this thesis, the TRIZ method was investigated as a potential third approach to reuse knowledge. However, it was identified that has a major overlap with the pattern method (and the reuse of abstraction) to be considered a radically different knowledge reuse approach.

Abstract solutions as a tool to support algorithmic design in architecture

To test the abstract approach aiming to support algorithmic design, this thesis uses Thirteen Design Patterns developed by Robert Woodbury (2010) as a method representing the reuse of abstract solutions in design and architecture. In his book 'Elements of Parametric Design' Woodbury

discusses the theory and proposes a practical methodology for the use of Design Patterns (Woodbury, 2010). The author points out that the method is a theory, which is yet to be tested.

The reuse of programming solutions is popular in Computer Science. To this extent Woodbury is proposing that the architectural design profession learns from the computer science profession. To help designers master the new complexity inflicted by parametric design systems, Woodbury proposes the use of design patterns as thinking and working tools. According to Woodbury, patterns, being themselves an old idea, are abstract solutions, which can be applied to shared problems. It is essential to think with abstraction in order to use design patterns successfully. In design, an abstraction describes not some specific example, but it rather refers to a general concept or idea, which is often associated with vagueness. In computer science, an abstraction characterizes a class of instances which omits inessential details (Ibid).

In chapter 3.3.2, 'Throw code away' Woodbury (2010) points out that designers tend to rebuild codes rather than reuse them. He says that programmers would most definitely be horrified by such wasteful acts. Surprisingly, while abandoning their own parametric models, designers are eager to invest time in finding existing models (developed by others) and utilising them for their own purposes (copy and modify approach) (Ibid).

A complex model usually consists of parts, which are mostly reusable. That is why Woodbury (2010) argues that reusable abstract parts are a keystone of professional practice in parametric design. The author describes Thirteen Design Patterns as a medium to understand and express the practice craft of parametric modelling. The Thirteen Design Patterns for Parametric Design are: Controller, Goal Seeker, Increment, Jig, Mapping, Organised Collection of Points, Place Holder, Projection, Reactor,

Recursion, Reporter, Selector, and Transformer. The author has outlined the following principles of patterns for parametric design (Ibid):

- Explicit. The others should be able to read (understand) your patterns in your absence. Writing a pattern may aid reflection on reuse of design ideas (reflection in action (Schön,, 1983));
- Partial: separate solutions to problem parts;
- Problem focused: a pattern should solve a shared problem;
- Abstract. Patterns are abstract and represent a general concept (divide-and-conquer). Some particular examples can be given to illustrate this concept.

The works of Alexander (1979), Gamma et al. (1994) and Tidwell (2005) helped Woodbury to identify the following structure of design patterns: Name, Diagram, What, When, Why, How, Samples, Related Patterns. The following methodology describes the steps for designers who want to create a Design Pattern:

- Identify: Name, What, When, How;
- Collect a set of sample files;
- Look at samples together and discover what they share;
- Refine patterns for clarity and simplicity;
- Share it (online) and make it easy to find.

Design Patterns developed by Robert Woodbury were used as a method to test the reuse of abstract algorithmic solutions in architecture. Participants of the DP (Design Patterns) group were introduced to the concept of patterns during the course of algorithmic modelling workshops. They learned the idea and reasoning behind each pattern and went through a step-by-step tutorials illustrating how patterns can be practically implemented (See more details on how Design Patterns were integrated in the algorithmic modelling course in Methodology section)

1.3 Case-Based Design methods in architecture and computation

Case-Based Reasoning

Design methods using case-based reasoning constitutes the core of the second approach tested in this thesis as a means to reuse design knowledge in algorithmic design. It is an example of the Case-Based Design approach. The aim of this approach (similar to the DP approach) is to work as a design support method, helping designers to better understand and use algorithmic modelling tools, using case-based reasoning (as opposed to generalised pattern-based reasoning of the DP approach).

Recently, the idea to use case-based reasoning to complement or replace other approaches supporting design has been explored by researchers in various fields of design (Maher, Pu, 2014). Case–based reasoning (CBR) is a problem solving approach, which utilises specific knowledge from previous cases, instead of making assumptions based on generalised relationships between a description of a problem and conclusions (Aamodt, Plaza, 1994). In CBR a new problem is solved by finding and reusing an existing solution from a similar case from the past (Riesbeck, Schank, 2013). In other words, in order to solve a new problem one has to remember (find) a previous similar situation and by making an analogy reuse the knowledge (solution) of this situation in a new context. In a paper discussing the principles and methods of case-based reasoning and problem solving Aamodt and Plaza (1994) claim that 'reasoning by reusing past cases is a powerful and frequently applied way to solve problems for humans'. This statement is also supported by studies on cognitive psychology of human problem solving and case-based reasoning (Ross, 1989), (Schank 1982), (Anderson 1983). There is evidence that when

humans solve new problems they predominantly rely on specific, previous encountered situations (Ross, 1989). Research on problem solving by analogy indicates that it is natural for people to use experiences from their past when solving new problems (Carbonell, 1986) (Riesbeck, Schank, 2013). Studies on human cognition show that people tend to use previous cases as models both when they are novices (Anderson, 2013) and when they are experts (Rouse, Hurt, 1982). In a recent paper, Riesbeck and Schank suggest that 'case-based reasoning is the essence of how human reasoning works' (Riesbeck, Schank, 2013).

Case-based reasoning provides a cognitive model for people, because thinking by analogy is consistent with natural patterns of problem solving for humans (Kolodner, 1991) (Riesbeck, Schank, 2013). As a matter of fact, CBR is used by humans as a primary mechanism for common reasoning on a daily basis. As a general rule, it is always easier to solve a problem second time, than first time, because people can reuse previous solutions and experiences (Kolodner, 1993).

One of the fundamental strategies to acquire knowledge is to learn from examples: in architecture these examples are design cases. However there is a fundamental difference between learning from examples and case-based reasoning. While acquiring knowledge similar cases (examples) are generalised into an abstract solution. In case-based reasoning the cases 'are generalised with respect to the context of a specific problem during each problem solving processes' (Hua, Fairings, Smith, 1996).

Terms related to the reuse of case-based solutions

The following list of terms and term explanations refers to the core concept of the second approach: the reuse of solutions from specific design cases. This approach to accessing and reusing algorithmic design knowledge follows case-based reasoning principles (Kolodner, 1993). Case–Based Reasoning (CBR) is a problem solving approach, which utilises specific knowledge from previous cases (Riesbeck, Schank, 2013), instead of making assumptions based on generalised relationships between a description of a problem and conclusions (Aamodt, Plaza, 1994). In CBR a new problem is solved by finding and reusing an existing solution from a similar case from the past (Riesbeck, Schank, 2013) (Heylighen, Neuckermans, 2001). There is evidence that when humans solve new problems they predominantly rely on specific, previous encountered situations (Ross 1989). Recently, the idea to use case-based reasoning to complement or replace approaches supporting design has been explored by researches in various fields including such disciplines as architecture and software design (Maher, Pu, 2014). In this research, the CBD (Case-Based Design) design approach was tested through an online case-base of visually represented parametric models and corresponding downloadable programming algorithms. These cases, and their illustrations were developed specifically for this research.

'Cases play a central role in architectural design education' (Zimring, 1995). Design cases are useful in solving problems for both novices and experts (Maher, Pu, 2014)

- In CBR a case can be considered as a story (experience) or a lesson; it can be vied as information about resulting solution; or it can be seen as a record of a method of how to solve a problem. Whichever way one defines it, the ultimate purpose of a case in

CBR is to help to solve a similar problem in future (Maher, de Silva Garza, 1997).

- Traditionally, in the field of design, knowledge has been recorded and formalised in a form of examples of successful designs, rather than generalised in the form of principles (Hua, Fairings, Smith, 1996).
- Cases are stories that capture past experiences, documenting 'real-world situations and analysing their outcomes' (Maher, Pu, 2014).

Case Adaptation implies that a new solution is created through the modification of a past case in order to meet the requirements (constrains) of a new design problem (Hua, Fairings, Smith, 1996). Design adaptation involves 1) mapping the differences between the new problem and the existing case to identify potential modification; 2) evaluation and execution of modifications (Maher, Pu, 2014).

Case-Based Reasoning is a paradigm for problem solving based on the reuse of specific past experiences (Maher, de Silva Garza, 1997) (Riesbeck, Schank, 2013).

- Case–Based Reasoning (CBR) is a problem solving approach, which utilises specific knowledge from previous cases, instead of making assumptions based on generalised relationships between a description of a problem and conclusions (Aamodt, Plaza, 1994).
- The Case-Based Reasoning mode involves more focused reasoning, applied to a very specific (narrow) context of a design problem (Pearce, 1992).
- In Case-Based Reasoning the cases 'are generalised with respect to the context of a specific problem during each problem solving processes'. While acquiring knowledge similar cases (examples)

are generalised into an abstract solution (Hua, Fairings, Smith, 1996).

- Case-Based Reasoning uses an abstraction from a specific experience (design solution) as a method to interpret and transfer this knowledge, in order to learn how to solve a new problem (Maher, de Silva Garza, 1997).
- Case-Based Reasoning is a cyclic process of solving a problem, learning from it and reusing this experience (knowledge) to solve a new problem (Aamodt, Plaza, 1994).

'Case-Based Methodology provides a way to easily generate answers' (Kolodner, 1991).

Problem Solving By Analogy. It is natural for people to use experiences from their past when solving new problems (Carbonell, 1986) (Gentner, 1983) (Riesbeck, Schank, 2013).

- Case-based reasoning provides a cognitive model for people, because thinking by analogy is consistent with natural patterns of problem solving for humans (Kolodner, 1991)

In Case-Based Design a new problem is solved by finding and reusing an existing solution from a similar case from the past (Aamodt, Plaza, 1994).

- As a general rule, it is always easier to solve a problem second time, then first time, because people can reuse previous solutions and experiences (Kolodner, 1991).

Dynamic Knowledge Repository - is a dynamic information space, it refers to a collective knowledge base, which operates within a particular domain of knowledge (Engelbart, 2003).

Database - a logically coherent collection of meaningful data (Robbins, 1994)

Knowledge-Based Or Expert Systems - the systems which use artificial intelligence (AI) techniques to solve expert level problems in specific domains of knowledge (Akerkar, Rajendra, 2010)

Role of design cases

Cases (or examples) can be viewed as stories that capture past experiences, 'recording real-world situations and analysing their outcomes' (Maher, Pu, 2014). Traditionally, in the field of design, knowledge has been recorded and formalised in a form of examples of successful designs, rather than generalised in the form of principles (Hua, Fairings, Smith, 1996). In practice, it is extremely difficult to find out the 'general principles which hold over all abstractions'. Alexander's design pattern language attempted to formulate knowledge in an integrated and abstracted way. However, the rules that he describes in his work have little generalisation, his patterns actually refer to particular buildings within particular environments (Hua, Fairings, Smith, 1996).

In CBR (Case-Based Reasoning) a 'case' refers to a previously experienced situation, which is interpreted and recorded in such a way that it can be reused in future (Aamodt, Plaza, 1994). According to Aamodt and Plaza case-based reasoning is a cyclic process of solving a problem, learning from it and reusing this experience (knowledge) to solve a new problem. That is why CBR is closely related to learning (Ibid). In fact, learning is a natural product of CBR problem solving, because when a solution is successful it is saved and recorded in a case base, so that in future people can learn from it to solve similar problems. Aamodt and Plaza also state that it is usually easier to learn by following a specific problem solving algorithm, than to 'generalise from it' (Ibid). According to Maher et al. a case in CBR can be considered as a story (experience) or a lesson; it

can be viewed as information about a resulting solution; or it can be seen as a record of a method of how to solve a problem (Maher, Balachandran, Zhang, 1995). Whichever way one defines it, the ultimate purpose of a case in CBR is to help to solve a similar problem in future (Maher, de Silva Garza, 1997).

'Cases play a central role in architectural design education' (Zimring, 1995). Past cases help students to identify a design problem, to inspire a potential design solution; to critically evaluate a completed design and to suggest alternative design strategies (Ibid). In design, case-based reasoning can be used for various purposes. For example it can be: adapting an old solution to a new design context; using past cases to explain and interpret new problems; and to critically evaluate and refine new design solutions (Kolodner, 1993). It is often argued that while case-based reasoning is an effective learning method, design cases are as useful in solving problems for both novices and experts (Maher, Pu, 2014)

In CBD (Case-Based Design) prototypes can also be referred as design cases (solutions). As one of the methods for reusing the knowledge in engineering and computational design some of the most successful solutions are used as prototypes. Prototypes are complete, fully developed design solutions able to be modified and integrated into a new problem (Hua, Fairings, Smith, 1996).

Case-Based Reasoning in design

Case-based reasoning is a paradigm for problem solving based on the reuse of *specific* past experiences (Maher, de Silva Garza, 1997). This problem solving paradigm was adopted by AI practitioners as a tool for design support. Maher et al. carried out a survey investigating the issues

raised by the use of CBR for design (Ibid). The study focuses on two contrasting types of case-based design: design assistance and design automation; and comments on the issues and difficulties related to the implementation of these approaches.

Maher at al. points out that when designing a CBR system three major aspects should be taken into consideration (1997):

- How the design cases are going to be represented;
- What is the process for recalling cases;
- What is the process for adapting design solutions;

The representation of a design case requires an abstraction of this case, as a means to translate this particular experience into a symbolic form that a designer or a computer system can understand and manipulate (Maher, de Silva Garza, 1997). Practitioners often employ abstractions, based on a design model, design method or philosophical approach to make sense out of a particular design experience/design solution (Ibid). To define the best way to represent a case, it is also important to consider what kind of information facilitates the reuse of a design solution (Maher, Pu, 2014).

The process of recalling/finding relevant design solutions involves several steps: *indexing*: to identify the features to search for in the past cases, relevant to finding a solution for a new problem; *retrieval:* to identify the cases with matching search features (indexes); *selection*: to evaluate the retrieved cases and choose the most fitting (Maher, de Silva Garza, 1997).

The design case adaptation is a process of reuse of a selected case in a context of a new design problem. The adaptation of a case usually involves: suggesting a selected case as a hypothetical solution for a new

design problem; evaluation of how well this this proposed solution will work; and the modification of parts and parameters to meet the requirements of a current design problem (Ibid). Maher and Pu state that the process of design adaptation involves two basic steps. The first step is to map the differences between a new design problem and the existing case (solution); this step is needed so a designer can identify the scope of potential modifications. The second step is the evaluation and execution of those modifications (Maher, Pu, 2014).

Principles of CBR methods

Case-based reasoning uses an abstraction from a specific experience (design solution) as a method to interpret and transfer this knowledge, in order to learn how to solve a new problem (Maher, de Silva Garza, 1997). CBR's problem-solving approaches often employ analogical thinking, especially in cases when the reused solutions (experience) are outside of current problem's context or domain. Instead of a direct adaptation or reuse of a design solution, analogy can indirectly provide valuable insight and assistance (Ibid). The basic idea of case-based reasoning in design can be expressed as: solving new problems by adapting solutions that were used to solve old problems (Riesbeck, Schank, 2013).

The approaches using case-based reasoning incorporate:

- Identification of a new problem (characterising the appropriate features)
- Retrieving the cases with those features (from the case-base memory);

- Evaluate the cases and find the best match for current design problem

(Riesbeck, Schank, 2013).

The CBR problem solving methods usually can be split into four major task groups:

- RETRIEVE: identify features (interpret a new problem and define its relevant descriptors)/search/initial match (a set of plausible candidates: past cases in the case-base)/select (best matching case);
- REUSE: copy (a solution or a method)/adapt;
- REVISE: evaluate solution/repair fault (detecting and fixing errors of a current solution);
- RETAIN: integrate/index/extract (solution, method or relevant descriptors).

(Aamodt, Plaza, 1994)

The representation of cases in CBD, whether visual or textual, can typically be split into three major groups:

- Problem-situation description
- Solution description
- Outcome description

(Kolodner, 1991)

The outlined principles of the representation and organisation of the design cases have informed the methodology of the CBD system development (repository of the algorithmic design solutions) which was used as a platform testing the reuse of specific cases (See the 'Adaptation of the CBD approach to the framework of this study' section)

Case-Based Design tools

One of the first works of CBR in the field of computer science and artificial intelligence was done by Roger Schank, who investigated the role of previous cases (including specific scripts and situation patterns) in learning and problem solving (Schank 1982). One of Schank's colleagues Janet Kolodner (1983) (1988) developed one of the first case-based reasoned systems CYRUS, which, basically, was a question-answering system with access to the database containing information about meetings and travels of Cyrus Vance, former US Secretary of State (Aamodt, Plaza, 1994). Later on a research group led by Kolodner and Domeshek developed and tested a case-based design aid system called ARCHIE, which worked in the domain of architecture (Domeshek, Kolodner, 1992).

Another exemplar-based knowledge system called PROTOS (Bareiss, 1989) was developed by Porter and Bareiss (1986). This research was pushed forward to create a new CBR system GREBE, which operated in the field of law (Branting, 1991). Currently numerous applications and systems, which use using case-based reasoning, operate in various domains of knowledge and practice, such as law, medicine, engineering and artificial intelligence. CBR tools are based on reasoning from old cases in order to solve new problems, evaluate proposed solutions or interpret situations. The core idea of aiding decision making through a CBD approach, is that a case-based system provides relevant past cases, which designers can utilise to solve a new design problem. Ultimately, it is always designers who do the actual decision making (Kolodner, 1991). We, as architects do not have a pre-defined algorithm for our designs and this fact could be taken either as a constraint or as a challenge. (Domeshek, Kolodner, 1992).

In the CBD study conducted by Hua et al., authors report that the creative adaptation (reuse) of design cases can lead to innovative designs, especially when two or more cases are combined (Hua, Fairings, Smith, 1996). Innovative ideas often occur through the adaptation and combination of existing design solutions (Sun, Faltings, 1994). Pearce et al. found that the use of CBD approach (Archie system) helps architects with getting new design ideas and inspirations by providing an opportunity to explore past cases. The case-based reasoning mode involves more focused reasoning, applied to a very specific (narrow) context of a design problem (Pearce, 1992).

Any case-based problem solving system is often composed of two main processes:

- Indexing, which refers to storing and retrieving of the reusable items (design cases)
- Adaptation, which is the reuse of a solution(s) within the new design context (problem) (Riesbeck, Schank, 2013)

Case adaptation implies that a new solution is created through the modification of a past case in order to meet the requirements (constrains) of a new design problem (Hua, Fairings, Smith, 1996).

Pearce et al (1992) investigated whether a large case-base (library) can support design in architecture by improving human decision making. The authors state that in order for their computer-based library of architectural designs to work, it was decided that:

- The system should support the design and problem solving process but all the decisions should be made only by the user.
- The system should have a specified narrow domain

- The system should focus on supporting the conceptual design stages, because a) often the decisions made on early stages have a major impact on how a design will progress further; and b) it is often more challenging to innovate conceptual design.

  (Pearce, 1992).

Case-Based Reasoning in Design Education

Case-based reasoning supports design. It helps designers with finding solutions for new situations by reminding them of experiences from the past. CBR is the way Architecture is often taught: in design education students learn how to be designers through experiencing design situations (Maher, de Silva Garza, 1997). In order to create new designs, people need to have previous experience or at least to have access to similar design experiences of others. Practice shows that 'designers rely heavily on specific design experiences' (Maher, de Silva Garza, 1997).

In architecture the support of design computation is hindered because it is necessary to control both the design generation process and the search process. Case-based design systems can be used as a solution to overcome the issues associated with the complexity of design generation and the search process (Dave, 1994). Architectural design is a domain which exist somewhere in between the sciences and the art. It is expected to simultaneously express both 'universals and particulars' (Dave, 1994). Architectural education heavily relies on the use of design cases as a communication medium to exchange experience and knowledge between teachers and students (Ibid). That is why example-based learning and teaching are commonly used approaches in the field of design and architecture. (Dave, 1994). The process of using past knowledge in order

to solve new design problems continues to be utilised in professional architectural practice as well as in education (Dave, 1994).

During the early stages of design, designers almost never work in a vacuum, instead they invest their time analysing existing designs and reviewing relevant information about earlier works. This mode of learning from past cases is common not only for the field of architecture, but also for fields where 'where designers work on something radically new 'such as engineering and physics. (Domeshek, Kolodner, 1992). Domeshek and Kolodner (1992) argue that if research in case-based design aims to support and improve design in architecture, conceptual design is likely to be the area with the potentially high payoff.

Case-Based Design Systems in Architectural Practice

Case-based reasoning and case-based aiding systems are equally useful for both novices and professionals. Case-based design (CBD) approaches can provide novices with the variety of knowledge and experience that they have not yet had. That is why novices are expected to improve their design performance using case-based systems. Case-based reasoning is especially helpful when 'knowledge is incomplete', or when there is a large number of unknown variables (parameters/evidence). The 'Case-based methodology provides a way to easily generate answers' (solutions) (Kolodner, 1991). CBD is a promising method for the design fields, which deal with geometry, such as: architecture, engineering and construction (Hua, Fairings, Smith, 1996).

Solutions from past design cases often help architects to solve their current design problems, refine solutions, improve proposed designs and justify particular design strategies and choices (Pearce, 1992). In

architecture many designs are created through the process of creative combination and adaptation and of past design cases in the new design context (Dave, 1994). Despite the fact that architects extensively use past designs in their decision making process, it is often very problematic for them to have access to appropriate cases (Pearce, 1992).

Heylighen et al. conducted a practical study testing six CBD systems for architecture: Archie-II, CADRE, FABEL, IDIOM, PRECEDENTS and SEED. The study states that CBD approach seems to be a promising method to develop 'intelligent design support' (Heylighen, Neuckermans, 2001). The authors define the case-based design systems as vehicles to 'find new design solutions by abating similar experiences from the past' (Ibid). Though all tested systems were developed for the domain of architectural design, each of them takes a different direction in terms of CBD methodology and 'ingredients' such as: case base content, organisation and representation; retrieval of cases; and reuse approaches (Ibid). The study states that the research on CBD tools has not reached its full potential and is yet to make the convincing breakthrough. However the authors indicate that recent experiments with the use of case-based design approaches in architecture show that students 'benefit from exposure to cases during the design process' (Heylighen, Verstijnen, 2000).

Among the possible weaknesses of the CBD approach is that the chosen case might be not the most suitable solution. Therefore, the major disadvantages of case-based design is that 'the solution space is not fully explored' (Kolodner, 1991).

Issues related to the implementation of CBD tools

The main issues in developing the CBD systems is representation and control issues. Representation refers to how a design solution is represented (how information is documented and presented to the users). Control issues relate to how a database (repository of cases) is organised and how the indexing works (Maher, Pu, 2014). Indexing: how to retrieve the best matching solutions from the case-base, is one of the big issues in the design of a large CBD system (Kolodner, 1991). Kolodner developed the guidelines for indexing a case-based memory. They propose that indexes should be:

- Predictive (to be illustrative of the solution/outcome features)
- Predictions should be helpful (useful in later reasoning, for example indexing design goals, constrains and solution features)
- Abstract (to be applicable to a variety of future problems)
- Concrete (to be recognisable/identifiable)

(Kolodner, 1991)

In CBD systems indexing and retrieval of cases can be done *informally* or *formally*. The informal method refers to the technique, when the users browse the repository and select cases themselves. The formal method is when the system uses the definition of a new problem as input and automatically retrieves solutions as output (Maher, Pu, 2014).

Some of the recurring issues related to practical implementation of case-based design identified by Maher et al. include:

- How to represent complex design cases
- How to link the specific design experiences with the generalised design knowledge
- How to formalise design experiences

(Maher, de Silva Garza, 1997)

In their study of CBR applications in design Maher et al. state that at the moment there is a universal way to resolve the major issues in the development of case-based design systems, such as the representation of individual design cases, the organisation of case-memory and case recall and adaptation. Authors point out that each CBD system addresses these issues in its own way based on the context and objectives. The bigger case-memories require the more efficient indexing/organisational principles of the system. This could be done through hierarchical indexing trees with multiple sub-branches and narrow specification of features. However it is a challenging task to predefine the set of features, which would be most helpful and relevant for future reuse (Maher, de Silva Garza, 1997). It also should be noted that, one of the main difficulties of using a CBD approach is to find the appropriate cases, which are scattered across various sources. (Zimring, 1995).

Uniform representation, including documentation, classification and indexing, of all of the design cases in a CBD system is an important issue. A systematic representation approach is needed, because in practice the way a project (case) is documented can vary greatly, depending on the individual background and preferences of each designer. When defining the system for the case representation, the most important consideration should be the facilitation of future design reuse. Case representation is a part of the design process in CBR, which is why CBD tools should provide case information in a format that will be most helpful for future retrieval and adaptation of a solution within a new design context (Maher, de Silva Garza, 1997).

Essentially, design case adaptation in CBD is the process of generation of a new design solution (Maher, de Silva Garza, 1997). The

adaptation process can be done either by a human designer or by a computer program. In option one a CBD system serves as a case library and provides relevant information about the cases, which can be used (reused) by a designer. This way a designer makes all the decisions. The second option means that the adaptation process is automated and performed by a computer program through a design algorithm, which finds a solution satisfying all the constraints. The role of designer in this case is to define these constrains and choose the design algorithm (Maher, de Silva Garza, 1997). Maher et al. state that the main issue of a CBD system development is not its degree of automation: both human and computer case adaptation methods can be successfully implemented in a case-based design system. The authors conclude that the major issue in the design of a CBD system is the need to develop a formal representation of the design experiences (Maher, de Silva Garza, 1997).

Pearce et al. report the following practical lessons learned from testing a large case library supporting design in architecture (Pearce, 1992):

- Design cases are often incomplete (not well documented), which makes it complicated or impossible to reuse;
- Design cases are often too large and complex, therefore it is often too hard to extract the useful information.
- The system should be able to cover multiple types of knowledge (reusable items): models, design methods and reasoning; which should be cross-indexed (labelled) so that user can find what is needed.
- The system should provide (present) relevant information to users. Cases can be usable only when the system interface presents the information in an intuitive, associable and easily understood format. (Pearce, 1992).

Complex design cases

In many domains the development of a feasible design solution often implicates the development of a complex system. The adequate representation of a complex design case is essential for the CBD approach, however it can often be a challenging task. There is a concept of reusable cases in the paradigm of case-based reasoning. In practice a design case is not 'one case', but it is a collection of various experiences and decisions that form a complex output system (Maher, de Silva Garza, 1997).

One of the ways to deal with case complexity is to decompose it into a set of subcases. This decomposition strategy allows designers to focus on the particular parts of a design solution, the parts which are most relevant to a current design problem (Maher, de Silva Garza, 1997). Subdivision of a complex solution into specialised sub-cases makes the analytical and reasoning process more efficient. For example, a design case can be decomposed according to its: function (design intentions/purposes); behaviour (interactions and respond to the environment); structure (physical and geometrical properties); and context (design's environments) (Ibid). Maher et al. concludes that designers' tend to handle complexity by dividing a case into smaller and simpler abstractions (Maher, de Silva Garza, 1997).

The investigation of structure and organisation of knowledge claims that (due to the specifics of the human cognitive model) knowledge in our memory exist both as generalisation and as a set of specific cases (events and experiences) (Heylighen, Neuckermans, 2001). According to this CBD cognitive model both generalised and specific knowledge follow the same organisational principles and vary mainly in the level of abstraction (generalisation). The study states that the central ingredient of the cognitive model in case-based design is the ability of the CBD system memory to

dynamically improve its performance. It implies that the CBD tools should be able to constantly update by: adding new cases to the memory base, re-organising the old cases (re-indexing) or establish new generalisations (abstractions) (Ibid). The study concludes that the structure and organisation of knowledge in current architectural CBD systems lack one of the most essential principles of the CBD approach: 'learning from experience'. This means that future research on the CBD systems should investigate the ways to dynamically change (update) the structure of a case base system, so that the system becomes responsive to users' interactions and inputs.

Indexing and case retrieval

In theory, recalling a case in case-based reasoning suggests that designers know what they are looking for in a case-base. This assumption implies that every design problem is fully defined. However in practice defining the problem is an integral part of a design process. That is why it is often difficult for designers to clearly identify the relevant search indexes, simply because they do not know yet what they are looking for. In many CBD systems the indexing and case retrieval is done by the user through informal case-base browsing and individual selection of relevant design cases (Maher, de Silva Garza, 1997). Design, especially conceptual design, is a task without a clearly defined specification (algorithm for design), because 'part of the problem to be solved is identifying the problem' (Domeshek, Kolodner, 1992).

Other research in the field of CBD also suggests that, in case-based design the classification and indexing of cases is regarded as one of the main challenges of developing (designing) a CBD system. (Dave, 1994). It

is hard to identify the features and characteristics, which will represent a design case universally, because individual designers 'see' different features in a design solution, due to the differences in their personal experiences and associations (Dave, 1994). Therefore, it is essential that a case representation method allows individual users to specify their own classifications, features and characteristics, which when inputted in the database system will re-organise the structure and representation of cases (Ibid). It is also important that the information (reusable items) in the case base is easily accessible and is presented in an adequate, easily applicable format (Dave, 1994).

Database and knowledge based systems

Case-based design principles can be used by designers and architects themselves but computer programs can also reuse knowledge, reason and make decisions based on processed information. In theory, computer knowledge based systems can perform some of the current designer's functions, for example solving some of the design problems by reusing/adapting existing solutions.

In this study the Case-Base Design approach was tested through the use of an online repository of algorithmic solutions, which is a database system. It is essential for this research to draw a clear distinction between the concepts of the 'Database system' and 'Knowledge-based system'. Both of those notions refer to the computer programs (software) which deal with data (including knowledge), but they manage and draw conclusions from this data in quite a different manner. Both database and knowledgebase systems were initially considered as possible methods to test the CBD approach.

In order to proceed with the comparison of the knowledge and database systems, it is necessary to give the definitions of following key terms, which will be used within the context of the research. Data is basically a collection of facts or information, which can be digitally extracted, interpreted, processed and displayed on a computer. In other words it is an organised set of related, structured and indexed information, which may exist in a form of physical files (folders, documents, etc.) or system data files. Data in the CBD system (testing the reuse of case-based solutions) consist of images representing the cases and attached files containing programming algorithms (Grasshopper definitions and corresponding Rhino files).

Current database systems are capable of operating, storing and managing a large amount of resources, which contain all sorts of information. Google, which is a hyper-textual web search engine (Brin, Page, 1998), has one of the top ten largest databases in the world. It is a very powerful and widely used tool for sharing information and knowledge all around the World. But it is not a 'knowledge based system', because it does not give an answer for a question or produce a new information, but it rather gives a list of relevant resources (existing data) when issued with a query.

'All Knowledge is information, but not all information is knowledge' (Siemens, 2006). From one perspective, knowledge is a human understanding of a subject matter that has been obtained through a study or experience (Akerkar, Sajja, 2010). But from another viewpoint knowledge can be processes not only by humans but also by other agents, such as computer programs (Wigg, 1999). George Siemens in his book 'Knowing Knowledge' (2006) states that people are only able to describe, not define knowledge. According to Siemens (2006), there are two main

characteristics of knowledge. First: knowledge describes or explains something, and second: knowledge can be applied in some type of action. Both of those characteristics are more than relevant towards the concept of a case-base system (CBD repository) and the reuse of knowledge (algorithmic solutions).

The notion of a knowledge based system is closely linked to the concept of an artificial intelligence. According to Akerkar and Rajendra (2010) a machine is intelligent if it exhibits such human characteristics as: respond to situations flexibly, make sense of ambiguous messages, assign relative importance to elements, find similarities and draw distinction between situations. Hence Artificial Intelligence (AI) attempts to solve problems by mimicking human thinking patterns, through symbolic and non-algorithmic problem solving approach. The systems which use AI techniques to solve expert level problems in specific domains of knowledge are called Knowledge-based or Expert systems (Ibid).

Knowledge-based systems (KBS) are much more ambitious then the database systems. KBS use existing data, information and knowledge to generate new knowledge. These computer programs can understand information, reason and make decisions based on processed information (Ibid). KBS are currently used in medicine to interpret symptoms and produce diagnoses, in business and banking to interpret input data and offer a prediction, in design industry to propose a configuration of product components etc. Tuthill and Levy (1991) have identified five types of Knowledge based systems: Expert systems (problem solving), Linked Systems, Case-based systems, Database in conjunction with an intelligent user interface, Intelligent tutoring systems.

However not all knowledge based systems aim to solve complex tasks. Some of them have a rather simple set of 'if-then' rules such as: for

example, to determine whether an applicant is eligible for a certain program or not. As Sargent (1991) points out, in practice only a tenth part of a typical knowledge based system consists of the actual knowledge manipulation, the rest of the system is mostly conventional software. More than that, such software techniques as: abstraction, inheritance, tree-navigation etc., which were originally developed for artificial intelligence, are now adopted and routinely used in database management and control systems. That is why in some cases it is difficult to distinguish between data-based and knowledge based systems (Ibid).

There are a number of reasons why it was decided that (as a means of testing the CBD approach) a database system suits the framework of this study better that a knowledgebase system. A database system does not solve a design problem (or any aspect of a design problem), instead it leaves all the reasoning to a designer. This way both the Design Patterns (DP) and Case-Based Design (CBD) approaches give the actual decision-making to users, which ensures a more equal set-up for this experimental study. Even though the online CBD system performed certain actions, such as sorting and retrieving cases (based on their indexes), it did not produce any new data or solve any problems by itself. All reasoning and decision making towards what features to search for, which solutions to selects and how exactly algorithms can be reused (applied in the new design context) was the hands of designers and architects who used this system. (See more details on how the Case-Based Design approach was used in the context of this study in the Methodology section).

The literature, discussed in this chapter, has informed various aspects of research methodology. Firstly, it has helped to formulate and clearly articulate the research problem (See the Research Problem Description section in Methodology chapter). This was done to identify

what are the current set of issues and how we can test (measure) whether the reuse of solutions can improve designer's performance (ability to overcome these issues)? This helped to **formulate the focus of the study and identify the aims and objectives** of the approaches in more detail and clarity (See Focus of the Study, Shared aims and objectives in Methodology chapter). Secondly, the theory behind the reuse of abstract and case-based solution provided a formative set of **principles for practical application of the DP and CBD approaches in context of this study** (See Adaptation of the DP/CBD Approaches in Methodology chapter). Lastly, issues discussed in this chapter informed the measures (evaluation criteria) that constitute the research metrics evaluating the reuse approaches. These measures are used by this thesis as evidence testing the research hypothesis: that the reuse of design knowledge can be an effective design support method in the context of algorithmic design in architecture (See Evaluation of the Approaches in Methodology chapter.

2. Methodology

2.1 Methodology for testing and comparing approaches

The Background chapter outlines the challenges that architects face when adopting algorithmic methods and using programming languages in design; and explains the principles of the reuse approaches (identified as a means to overcome these challenges). This methodology chapter explains and illustrates the core of the problem (specific to the context of algorithmic design in architecture: what the problem is and why it occurs); and relates it back to the objectives of the approaches (how the reuse of abstract and case-based algorithmic solutions can help to solve the problem). The chapter explains what this experimental study is testing, the effect of the approaches; and which particular criteria are being measured and why.

Research problem

Algorithmic modelling tools allow designers to create design models via programming. Instead of direct manipulation with the form, an architect creates a programming logic (either by textual script or visual programming) (Leitão, Santos, 2011) which generates a model as an output. This process is fundamentally different from conventional form-making approaches in design and architecture where a model is created by manipulation with the geometry itself.

To use algorithmic modelling tools an architect has to think like a programmer and build a step-by-step algorithm of actions which are to be executed by a computer program. The following example illustrates the basic principles of creating a model through building a simple five step programming algorithm.

Step One: Create a Point, with coordinates: X=0, Y=0, Z=0;

Step Two: Create a circle at the centre of this point with a radius of 20 (mm);

Step Three: Divide the circle (curve) into 20 equal segments;

Step Four: Create lines between each division point of the circle and the centre point of the circle;

Step Five: Extrude lines along the Z vector, with vector value 10 (Exhibit 2.1)

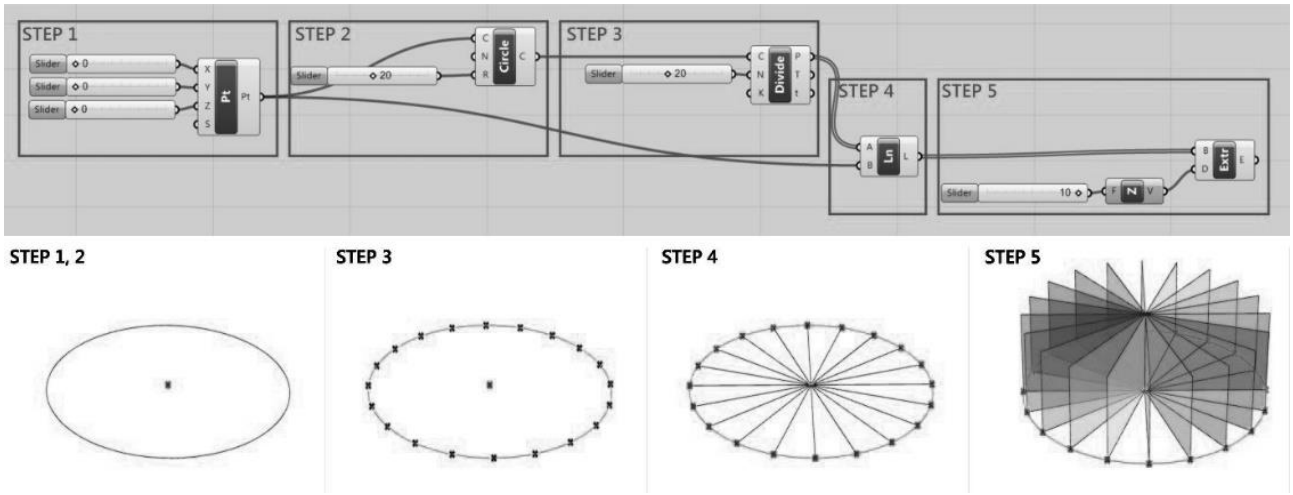


Exhibit 2.1. Example of a step-by-step algorithm of actions and corresponding output geometry. The output model and programming definition was created using Grasshopper (Grasshopper 3D, 2014), a graphical algorithm editor integrated with Rhino (Rhino3D, 2014).

Algorithmic modelling tools allow architects and designers to generate complex and mathematically precise models. They can also be used to produce simulations, such as particle motions, surface transformations and structural element movements. By a simple change

of parameters or a change in the form-making logic of a programming algorithm, a designer can obtain varying iterations or modifications of an output model without necessarily re-building the form manually. However, along with all the opportunities and advantages, the use of algorithmic design has its disadvantages.

A large number of people who are currently learning and implementing programming in their designs face major difficulties in mastering and applying, in practice, the programming principles and grammar (Celani and Vaz, 2012) (See introduction section). This applies to both textual (scripting) and visual (box-and-wire) programming methods. In order to use programming, one not only has to know which commands or programming components to use in each particular case, but also has to be able to build the correct sequence of these commands. When one of these conditions is not satisfied, a flawed programming algorithm will return an error, generate an un-intended output model, or in a worst case scenario result in a software crash.

The use of computational modelling tools requires an algorithmic, 'step-by-step program' way of design thinking. Fundamentally, programming logic does not relate to conventional design approaches in architecture, such as hand sketching, building physical models or manual CAD modelling. Traditionally, programming has not been a part of the architectural syllabus (Burry, 2011). Both advanced and novice users of algorithmic modelling techniques often face difficulties with the implementation of programming languages. Many designers and architects struggle to integrate algorithmic thinking into design process (Woodbury, 2010) and it can be especially frustrating for beginners (Celani, Vaz, 2012). The current shift in architectural education and practice towards new computational technologies and design approaches

is still an on-going process, as much as the development of the computational technology itself. Algorithmic modelling tools are constantly being updated, adapting to the demands of the design field and becoming more powerful and intuitive.

This thesis aims to contribute to that on-going process by investigating ways to support the use of programming in architecture. While developers of software and programming languages work towards improving various aspects of the software platforms that designers use, this study looks at the problem from the designer's perspective and investigates ways to support learning and use of algorithmic modelling through integrating new approaches into the design process itself. This research explores methods to reduce the barriers of using programming in architecture and potentially improve modelling performance through utilising existing algorithmic design solutions. Algorithmic design belongs equally to the fields of design and programming, and the reuse of solutions as a method to support design is an important part of programming practice (Krueger, 1992).

Therefore it is reasonable to suggest that the knowledge reuse approaches can potentially be as useful when applied in the field of algorithmic design in architecture.

Focus of the Study

Two approaches have been proposed as a means of accessing and reusing existing algorithmic design knowledge. The first approach is the reuse of abstract solutions to a design problem. An example of this approach: Robert Woodbury's patterns for parametric design (Design

Patterns) (Woodbury, 2010) was used to test the first reuse method. Design Patterns are abstractions. They are generic reusable solutions which are documented in such a way that is broad enough to apply to a range of different design contexts (Alexander, 1977). Thirteen Design Patterns, identified by Robert Woodbury (2010), aim to help designers learn and use algorithmic modelling systems. Woodbury states that patterns are useful because they promote communication, and can be used as a vehicle for sharing design ideas. Although, as the author states, writing a design pattern can take a considerable amount of time and effort, it aids reflection on and reuse of design ideas. According to Woodbury, design patterns are especially effective when a designer is doing the same thing again and again in variations (Ibid). Originally, patterns for parametric design were developed to assist designers and architects with structuring their programming solutions on an abstract level by reusing one or several of the Woodbury's Thirteen Design Patterns. Woodbury states that the proposed Design Patterns can be an effective medium to understanding the essence of algorithmic modelling (Woodbury, 2010). The author claims that patterns can help to overcome complexity inflicted by parametric design systems, but also states that it is a theory that is yet to be tested (Ibid). This study aims to test Design Patterns as a learning and design support approach.

These claims are supported by research conducted by Gamma et al in the field of software design, who observed a number of positive effects associated with the reuse of abstract solutions (patterns) (Gamma, Helm, Johnson, Vlissides, 1993).

Authors state that patterns reduce the effort required to learn software, helped with the design development, helped to explore alternative solutions and motivated users to 'go beyond' specific objects (Ibid).

Therefore an objective of this research was also to find out whether the use of Design Patterns in the context of algorithmic design in architecture would have similar positive effects.

The second proposed approach is based on case-based reasoning and the reuse of specific programming solutions: Case-Based Design (CBD). In Case-Based Design, instead of creating a new solution for each individual problem, a new problem is solved by adapting an existing solution from a similar case from the past (Riesbeck, Schank, 2013). Research in the field of human reasoning indicates that case-based reasoning is a natural way for people to solve any problem (Aamodt, Plaza, 1994) (Riesbeck, Schank, 2013), because when humans solve new problems they primarily rely on experience from previously encountered situations (Ross, 1989). Learning by following a specific problem solving algorithm is usually easier than to learn by generalising from it (Aamodt, Plaza 1994). This implies that, potentially, the reuse of case-based solutions can be expected to be easier and more intuitive for architects and designers compared to the reuse of abstract solutions.

Some authors claim that case-based reasoning is an effective design support method because it helps designers with solving solutions for new situations by reusing experiences from the past (Heylighen, Verstijnen, 2000) (Maher, de Silva Garza, 1997). The CBD approach is also claimed to help designers with overcoming problems associated with the complexity of design generation (Dave, 1994) and deems to be an especially promising method for design fields dealing with geometry (Hua, Fairings, Smith, 1996). There is, however, controversy regarding the effect of the CBD approach to design innovation. According to one opinion, the reuse of case-based solutions can lead to innovative design (Hua, Fairings, Smith, 1996) (Sun, Faltings, 1994). According to the other, Case-Based

Design actually limits the explored space of solutions (Kolodner, 1991), which can potentially supress design innovation.

Therefore, one of the main objectives of this study is to test whether these claims and suggestions regarding the reuse of case-based solutions are valid when applied in the field of architectural algorithmic design.

The secondary set of research objectives is to investigate the ways to overcome some of the challenges that the use of the CBD approach is likely to impose on designers (as well as the CBD systems developers). One of these issues being that it is often hard to find the appropriate reusable cases (Zimring, 1995). Even when cases are located in a single organised repository, finding them might be challenging. The problem is that, it is often assumed that when designers are searching for cases to reuse, they already know what they are looking for. In practice, defining the problem and, therefore, knowing which search features (indexes) to use, is an integral part of a design process (Maher, de Silva Garza, 1997) (Domeshek, Kolodner, 1992). Moreover, designers often 'see' different features in the same design solutions as they have different backgrounds and associations, which makes it very challenging to find universal indexes which would work effectively for all designers (Dave, 1994).

Therefore the secondary aim of this research is to investigate the ways in which designers and architects tend to think about their algorithmic designs. This is planned to be done through the investigation of how architects describe their design concepts, models, and algorithms; and try to identify the types of indexes (key words) that could be more effective.

In this study the Case-Based Design approach was tested using an online repository of visually represented models and corresponding downloadable programming algorithms. This approach provided a means to share programming solutions, allowing direct reuse (copy/modify) of existing algorithms. The idea of the effective reuse of existing algorithmic solutions appears to be relevant, because the computer technologies and the Internet have already become an integral part of everyday life, as well as a part of the architectural design practice and education. Access to online databases and the ability to obtain relevant information is likely to continue being a part of most design practice.

Therefore it seems sensible to investigate how architects and designers can utilise this opportunity of having constant access to online resources containing existing design knowledge and how this access to reusable solutions in return can influence design process.

In theory, the reuse methods of abstract and case-based algorithmic solutions are applicable to any type of textual and visual programming. Therefore these approaches are likely to be relevant even when all the current versions of the modelling software and programming languages become outdated.

Shared Aims and Objectives of the DP and CBD Approaches

One of the shared objectives of the DP and CBD approaches, stated in this thesis, is to reduce the number of barriers related to the use of programming languages. The goal is to increase users' ability to overcome these barriers on their own by reusing existing solutions (abstract or case-based).

The next objective, common for both the DP and CBD approaches, is to increase designers' knowledge and awareness of the existing algorithmic solution space. Hypothetically both DP and CBD approaches can produce original design ideas and programming strategies. Thus, they can both contribute to the decrease of design limitations as (by default) each user is no longer limited by his or her own individual knowledge and understanding of the subject. Regular interaction with the DP and CBD solution can potentially lead to the expansion of the explored solution space.

The other set of objectives is associated with designers' capability to enable computational design thinking, and their ability to employ algorithmic reasoning to translate a design idea into a step-by-step programming algorithm, generating an intended geometry. The objective is to help users structure their programming logic thereby increasing productivity of algorithmic modelling by offering examples which they can reuse in the context of their current design problems. The outlined above arguments and hypotheses informed the evaluation criteria used in this study. (See Appendix B, page B55).

Research Aims and Objectives

Through comparison of the Design Patterns (DP) and Case-Based Design (CBD) approaches this research investigated ways:

- to overcome the barriers, which users face when adopting the principles and grammar of programming in architecture; and
- to make the use of algorithmic design tools more effective.

In order to evaluate and compare how each approach influences various aspects of algorithmic design, the study has identified five groups

of criteria which formed the research evaluation metrics. The metrics include these criteria groups:

- algorithmic modelling performance (ability to effectively use algorithmic modelling systems);
- programming criteria (ability to overcome barriers associated with programming),
- design ideation (ability to realise an idea-to-form translation using algorithmic modelling environments);
- motivation criteria (the level of satisfaction with the design output and motivation to use algorithmic modelling in future) and;
- approach characteristics (the level of how easy to use, intuitive and helpful each approach is);

These metrics provided a means to identify to what extent and in which particular aspect each approach improved and supported designers' ability to use algorithmic modelling tools in architecture and design. Three test groups were compared: the control group, which used No Approach (NA), the group which used the DP approach (reuse of abstract algorithmic solutions), and the group which used the CBD approach (reuse of case-based solutions). To test the effect of each approach, comparisons between the control group and approach groups were conducted. This gave a means to answer the main research question, which was: whether the reuse of abstract and case-based algorithmic solutions could help architects to overcome programming barriers and improve their algorithmic modelling performance. Ultimately, this study aims to test whether it is worth using the DP and CBD approaches in the context of algorithmic design in architecture or not.

The comparison between the Design Pattern and Case-Based Design groups allowed the investigation of the strengths and the

weaknesses of each approach. Through this comparison the study aims to explore how each approach can potentially be improved.

Designer Population

The target group for this study was established as architects, landscape and interior designers who were learning or already using algorithmic modelling tools in their designs. Woodbury (2010) states that Design Patterns were developed for both designers who were still learning and who were already using parametric modelling. The second approach, Case-Based Design, also applies to a wide range of designer population. People tend to reuse previous cases both when they are novices and when they are experts (Anderson, 2013) (Rouse, Hurt, 1982), and by adapting these existing solutions designers were expected to benefit from past cases (Heylighen, Verstijnen, 2000). In order to carry out the proposed experimental study (See Experimental set-up section), and to test and compare the two approaches, a list of criteria was identified for selecting participants. The following participant selection criteria were established:

- people who were doing/learning architectural, landscape, or interior design;
- those with design experience of at least one year (to ensure certain fluency and confidence in design);
- those who were interested in learning how to use algorithmic modelling systems/or who were already using algorithmic modelling systems;
- open (flexible) towards new design methods and ideas;
- keen on mastering and experimenting with computational design technologies;
- available in terms of time;

The 126 participants who were recruited to participate in the experimental part of the study were a diverse group of architecture and design students, and practicing architects. Their design experience varied from 2 to 33 years (including the years of studying of architecture and design) with an average of 4 years' experience. When indicating experience with any computational design tools (including the use of visual or textual programming languages) the range was from 0 to 3 years with an average of 4 months' experience in computational design. When specifying their experience using Grasshopper, participants reported an average of only 1.5 months, with the majority of participants having no experience with the software. These results indicate that the recruited test designer population were mostly novice programmers with an average of 4 years design experience.

The test groups of at least thirty test subjects per approach (See Statistical Analysis Section for more details) had both male (55%) and female (45%) participants and were balanced in terms of design experience.

Software Platform

Algorithmic modelling methods are implemented through the use of textual and visual programming languages. The key difference between these methods of representation is the level of language abstraction (Mitchell, 1975). Visual or diagrammatic (analogue) programming languages are represented by a so called 'box-and-wire' modelling environments, while scripting or textual programming languages use sequences of text: words, punctuation, and numbers (Exhibit 2.2).

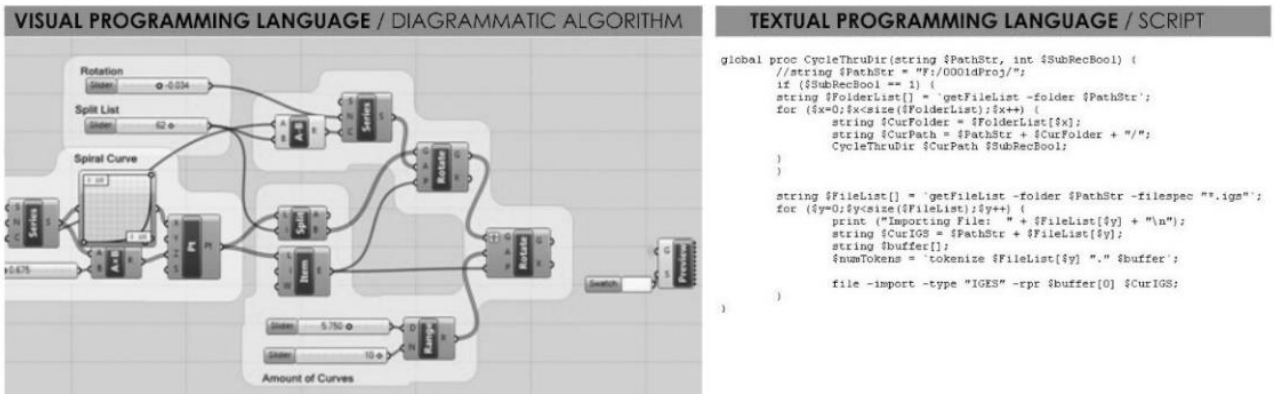


Exhibit 2.2. Visual and Textual programming languages

There are advantages and disadvantages in both (textual and visual) types of programming languages. The biggest disadvantage of scripting is that it has very strict syntax rules, which are often hard to follow (Celani, Vaz, 2012). Syntax errors, which occur during the scripting process, can be very discouraging for many designers who are learning how to use a computational design system. Scripting requires the user to have comprehensive knowledge and skills in programming language rules and syntax. The disadvantages of visual programming environments are related to the limitations that the 'box-and-wire' system imposes on the variety of available functions and components. Essentially, each 'box' contains a script that can be a function or an action; and the number of 'boxes' is limited. Nevertheless; these limitations can be overcome when combined with textual programming, through adding a script 'box' (Leitao, Santos, 2011). Recent research in algorithmic design tools indicates that users (especially novices) are more enthusiastic and successful in understanding and realising design concepts when they use visual programming (Celani, Vaz, 2012). Examples of visual programming environments include: Grasshopper (Rhino), Generative Components' (GC) Symbolic Diagram and Houdini (Sidefx) etc.

A recent study conducted by Janssen and Wee (2011) compared these three mentioned systems. The research explored the cognitive stress

associated with iterative construction of visual dataflow modelling (VDM) environments. VDM refers to a modelling approach that uses visual programming languages to create algorithms (which generate output geometry). Visual programming was undertaken through the manipulation of graphical elements rather than entering text (scripting) (Exhibit 2.2). In order to test the visual programming systems an exercise was conducted: each platform was used to build the same complex parametric model (Janssen, Wee, 2011). All three programming environments have completed the modelling task successfully in this research (Ibid). The approximate number of nodes used to generate the model was: 80-90 for Grasshopper, 90-100 for Generative Components (GC) and 70-80 for Houdini. The authors indicated that in order to perform certain iterations in GC, a user is forced to follow a reverse-order modelling method which causes additional cognitive stress. Grasshopper and Houdini, in contrast to GC, both use the forward-order modelling method. It is also noted that GC heavily relies on scripted (textual) expressions for manipulating such data as: lists, sets or arrays. Thus it is not possible to avoid scripting while working with GC (Janssen, Wee, 2011).

With visual programming environments one can expect to have tangible design outcomes after a short series of practical tutorials, even from people who are new to algorithmic modelling. Both Grasshopper and Houdini suited the context of this study. When choosing the two software platforms, additional factors came into play. Firstly, both Rhinoceros and Grasshopper were available at Victoria University of Wellington in their computer labs where this study was conducted. Secondly, there was observed an increase of interest towards the use of visual programming with Grasshopper among the students of architecture

and design at Victoria, as its 'box-and-wire' environment was user friendly and could be explored and operated intuitively (Grasshopper3D, 2014). In addition, the author of this study was already experienced with both Grasshopper and Rhino prior to conducting this experimental research. That is why it was decided that the Design Patterns and Case-Based Design approaches would be tested on the Grasshopper (visual programming plugin for Rhinoceros) software platform.

Experiment Setup

The experiment was set in the framework of a two day algorithmic modelling workshops using Grasshopper, a visual programming platform, integrated into Rhino 3D (Grasshopper3D, 2014) (Rhino3D, 2014). The workshops were set up as a series of short lectures and intensive practical tutorials containing the systematic introduction into visual programming with Grasshopper.

The same experimental setup (treatment) (Groat, Wang, 2002) was organised for all three test groups: control group (using no approach), group using Design Patterns approach, and group using Case-Based Design approach. All participants were given an opportunity to master the same set of algorithmic modelling skills (See Algorithmic Modelling course framework Section for more details). All groups were introduced to the same programming components, computational and algorithmic form-making logic, and went through the same step-be-step practical tutorials. The only difference was that participants of the control group did not learn and use any additional design support approach.

Participants of the Design Pattern group were introduced to the concept of patterns for Parametric Design and throughout the course of

the workshops they gradually learned all thirteen patterns developed by Robert Woodbury (2010). The first pattern to be introduced in the course was 'Clear Names'. 'Clear Names' used to illustrate the concept and organisational structure of Design Patterns (Intent, Use When, Why, and How) (Ibid). The objective of the two day algorithmic modelling course was to use more simple algorithms and programming logic in the beginning and then gradually increase the complexity (See Recommendation Section and Appendix A 'Proposed curriculum of teaching programming in architecture using patterns for parametric design').

Participants of the Case-Based Design (CBD) group went through the same practical tutorials as the control group and the Design Patterns (DP) group. The only difference was that the CBD group participants were given access to the online repository of algorithmic solutions (case-base), were shown how to use it (searching cases by index (key words) and were given permission to download corresponding programming definitions).

At the end of each day of the workshop, participants were asked to design an algorithmic model (Exhibit 2.3) (See also Appendix B, pages B56-B63), based on a design task (the same for all test groups), and to answer an online questionnaire. The task for the first workshop day was 'abstract composition', the task for the second day was a 'parametric canopy'. Prior to modelling, participants were asked to quickly sketch their design ideas and think how they could build an algorithm that would generate the form that they envisioned. The time given for the development of these conceptual design models was set at 2 hours (the same for all test groups). It was suggested to participants of the DP group to use Design Patterns that they learned when developing their own design tasks. However, the use of Design Patterns was not compulsory,

and participants were free to proceed with the development of their algorithmic design models as they thought worked best for them. Similarly to the DP group, the CBD group participants were free to choose whether they wanted to reuse any algorithms from the Case-Base or not to reuse them.

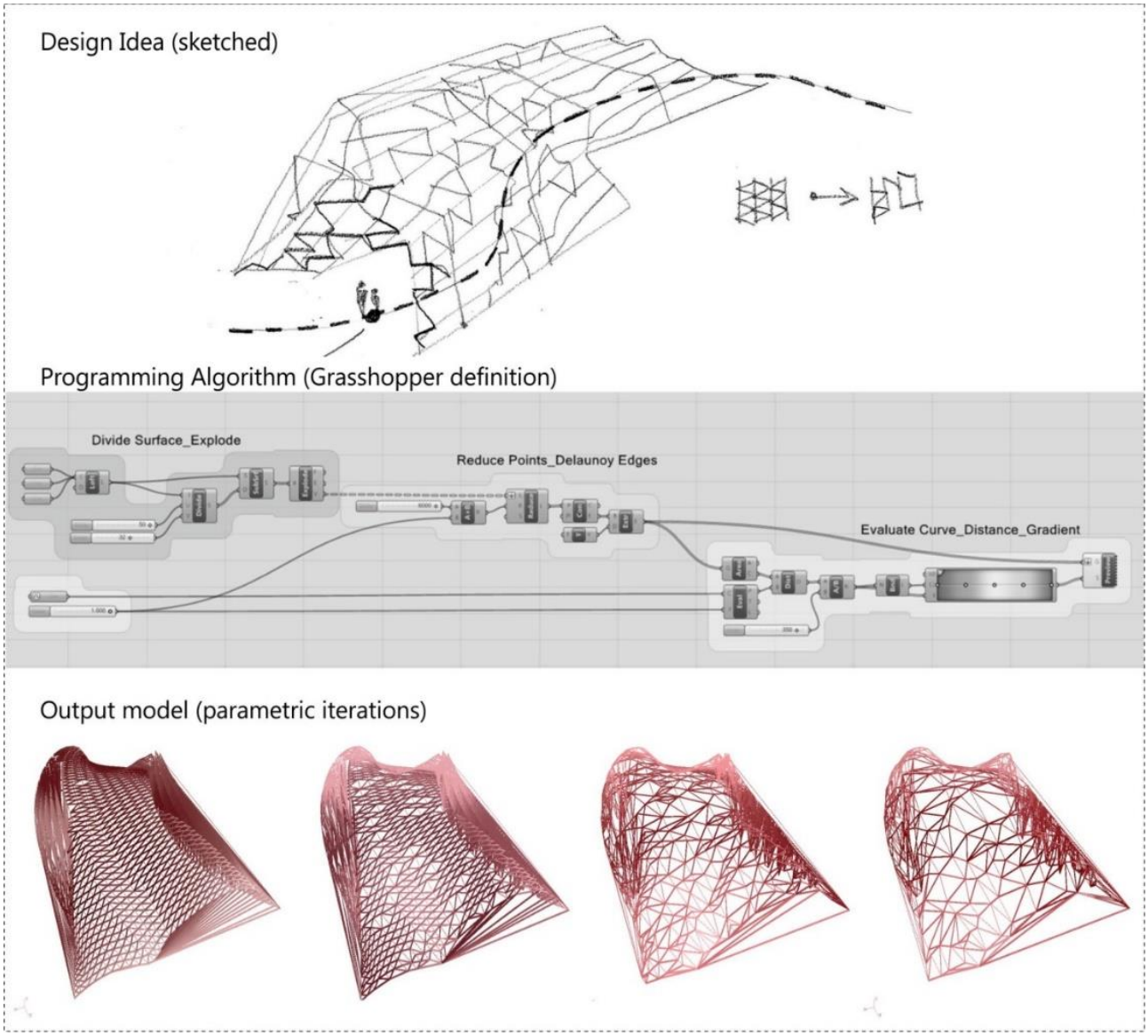


Exhibit 2.3. Example of work submission (Design Idea – Sketched, Programming Algorithm, output design model)

The collected data (from the workshops) consisted of the screen recordings (snapshots of the design process), submitted sketched design ideas, 3D models (Rhino files), programming definitions (Grasshopper

files) (Exhibit 2.3) and answered online questionnaires. The 3D Rhino models were used to calculate the level of complexity of each model. The Grasshopper definitions were used to measure the complexity of each programming algorithm and to determine the explored solution space of each algorithmic solution (See Detailed Criteria for comparing the approaches).The data collected from the online questionnaires helped to determine the largest portion of the key criteria identified for this study including the level of programming difficulties and the amount of the reused algorithms. It informed such criteria as the level of satisfaction with the design outcome and the motivation to use algorithmic modelling systems in the future. The questionnaires provided data regarding the design objectives, the ability to model the original design idea and the degree of change in the design due to programming difficulties.

The aesthetic and design qualities of the models were not judged directly, as any judgement regarding design qualities may have been to a certain degree subjective, varying in dependence to the individual preferences and the background of the person evaluating the design. However this issue was addressed indirectly. Each participant was asked to indicate their design intentions, reflecting on the design outcome, and to evaluate the degree of satisfaction with the produced model. In this way, the design quality of each model was, in fact, assessed by the designer himself/herself. This strategy also gave an opportunity to have an insight into what each person intended to achieve versus what was actually achieved.

Structuring the Comparisons

The design scope and constraints of the case studies were developed according to the two main strategies. The first strategy was to keep the design tasks simple but open to various interpretations, thus ensuring an easily controlled short-term experimental framework, and a fast and efficient analysis of the outcome results. This strategy also gave an opportunity to test the identified algorithmic modelling criteria, such as the number of programming difficulties, explored solution space, and degree of algorithm and model complexity. The second strategy was to use practical exercises which allowed the potential for algorithmic design to be expressed to its full extent, hence the choice of the exercises: 'abstract composition' and 'parametric canopy'. Although the implementation of algorithmic modelling can, hypothetically, be implemented within the context of almost any design scenario, in design studios it is often used to create such geometries as surfaces (including canopies and building envelopes), algorithmic ornaments, or urban or landscape planning, etc.

The first practical exercise consisted of designing a simple abstract composition (See Appendix B, pages B56, B58, B60, B62). Participants were expected to develop rather simple programming definitions (algorithms) which would generate intended outcome geometry. The objective of the first exercise was to introduce and get users familiar with practical implementation of algorithmic modelling. The second day exercise consisted of a slightly more specific task: a parametric canopy (complex, possibly interactive, surface) (See Appendix B, pages B57, B59, B61, B63). In both cases participants were asked to describe their design ideas prior to modelling. This was done to track the relations between the design concept and the resulting model. It was anticipated that on the

second day of the workshops participants would develop more complex algorithms and geometries compared to the first exercise.

The set-up of the workshops structure was informed by a number of existing experimental studies in design. For example, a similar design scope (exercises) was used by Celani and Vaz (2012) for a comparative study of the use of scripting and visual programming in computational design, as well as by Jasses and Chen (2011) for their experimental study, which compared three visual dataflow modelling (VDM) systems.

Algorithmic Modelling Course Framework

To ensure equal treatment, participants of all three test groups, including the control group, the DP group and the CBD group, went through the same practical algorithmic modelling tutorials. This meant that all test groups had the same set of lectures and practical programming exercises which were given to them on the first and the second day of the workshops. 'Parametric Architecture with Grasshopper' (Arturo, 2011) and 'Grasshopper Primer' (Payne, Rajaa, 2009) informed the development of course structure. The course was adjusted to accommodate the gradual introduction of the Design Patterns in the DP group (See Appendix A for more detail). The basic principle for course organisation was to gradually increase the complexity of introduced concepts and programming components. Practical step-by-step tutorials using Grasshopper for Rhino covered such topics as (in order of introduction):

- Working area (Interface); Components and data; Components' connection;
- Parameters and components; Import from Rhino (Linking geometry/data); Data Management;

- Numeric data; Coordinates; Mathematics;
- Vector Basics; Point; Vector Manipulation;
- Operators (Move, Rotate, Scale);
- Curves; Types of Curves; Creating Lines; Polylines; Curves from Points;
- Surfaces; Creating Surfaces from Points and Curves
- Lists; Shifting Data; Data Management;
- Reparameterise; 'Remap Numbers'
- Numerical sequences; Series; Range; Random; Fibonacci series;
- Data Tree; Flatten Tree; Merge; Graft Tree; Tree Branch; Explode Tree;
- Paneling Tools; Surfaces' analysis; Divide Surface;
- Transformations with shape variation; Project; Graph Mapper; Deformations: Morphing;
- Conditional Statements, Split List; Cull Nth; Cull Pattern; Dispatch;
- Distance; Attractors;
- Colours, Gradients, Text Display,
- Script Components, Arrays and Lists; Loops; Visual Basic, Recursion, Fractals

At the end of each workshop day participants of all test groups developed and submitted the same design tasks and answered the same questionnaires (except that the control group had no questions regarding their experience with the approach, as they used no approach). The key difference was that that the CBD group had access to the online repository of algorithmic solution and that the DP group was introduced to thirteen patterns for parametric design and was shown how to use these patterns in practice.

Principles of the abstract and case-based solutions reuse

Both the Design Patterns and Case-Based Design approaches are based on the idea of knowledge reuse. The difference between the approaches is that one of them utilises abstract design solutions while the other utilises specific design solutions. This results in the substantial difference of the reuse methodology between the DP and CBD approaches (Exhibit 2.4, Exhibit 2.5).

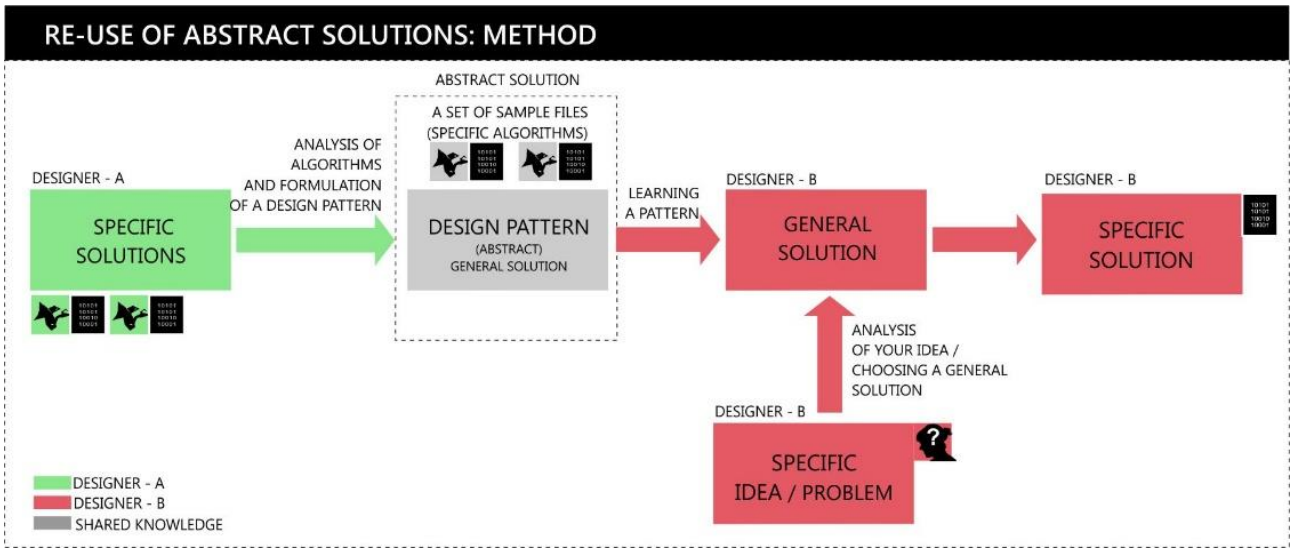


Exhibit 2.4. Reuse of Abstract solutions: Method

Exhibit 2.4 illustrates that initially, an abstract generalised solution (pattern) can be formulated through the analysis of existing algorithms which have the same underlying logic ('Designer A' 'Specific Solutions'). After a pattern is documented and the information is published *('Design Pattern')* other designers can learn this pattern *('Designer B' 'General solution')*. When working on a new problem *('Designer B' 'Specific Idea/Problem')* designers can apply this general solution (pattern) to help them solve their current design problems *('Designer B' 'Specific Solution')*.

Due to the fact that patterns are abstract, it is possible to reuse them (when appropriate) in different design contexts.

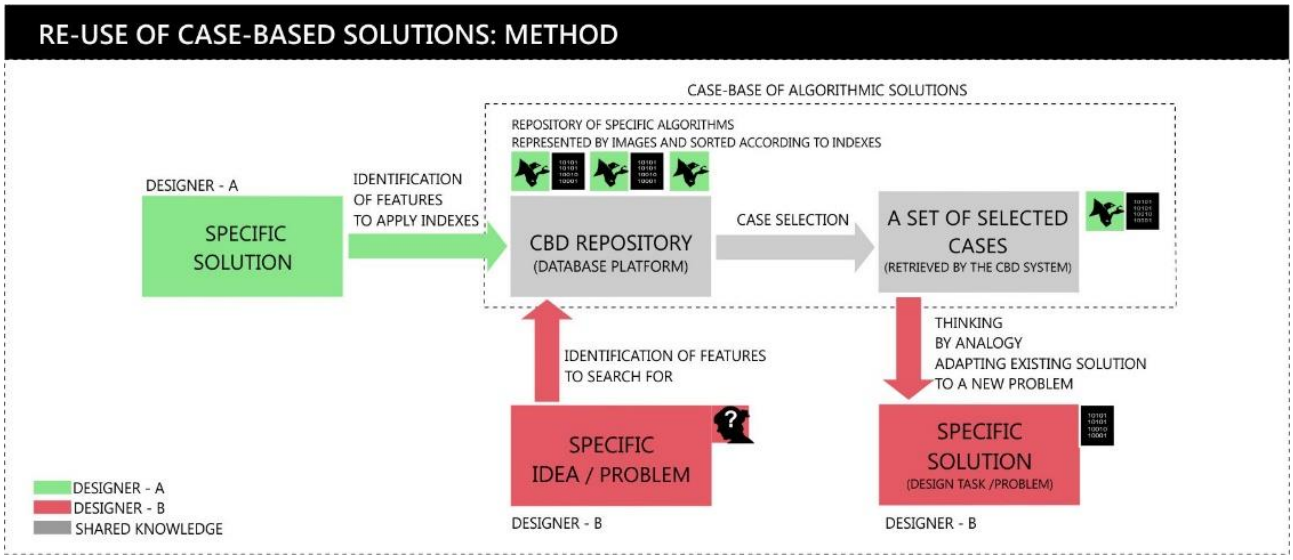


Exhibit 2.4. Reuse of Abstract solutions: Method

Exhibit 4.5 illustrates the methodology of the reuse of Case-Based Design solutions. Using data-base systems (such as weblog platforms) designers can publish any of their algorithmic solutions, making them available for others to reuse ('Designer A' 'Specific Solution'). When choosing indexes for their solutions, designers should try to identify a set of specific features (characteristics) of their designs that will be most useful, when others search for similar solutions in future *('CBD Repository')*. When other designers use the CBD system they also have to identify the features of their current design problems (ideas) to search for in the database *('Designer B' 'Specific Idea/Problem')*. Based on the match of the originally applied and search indexes a CBD system retrieves a set of selected cases *('A Set of Selected Cases')*. Thinking by analogy, designers can adapt one (or several) of these retrieved solutions to help them with the development of their current designs *('Designer B' 'Specific Solution')*.

One of the key differences in the methodologies of the DP and CBD approaches (adopted for this study) is that designers using abstract solutions are expected to learn patterns before using them. The development of a pattern can also require a certain amount of effort. However, in theory, once a pattern is learned, designers can apply this general solution to a variety of different design contexts and problems without re-learning it. To use a Case-Based Design approach users will most likely have to search for a reusable solutions each time they have a new design problem. This process can potentially be complicated and time consuming. Nevertheless, those designers who publish their designs in the case-base system, are likely to require less time and effort, because they do not have to spend time on formulating and documenting a generalised solution (with a set of sample files).

Note that, that alternatively patterns can be stored, retrieved and reused using a database repository (similar to the CBD system). In which case, designers do not have to learn patterns beforehand. However, in this thesis the thirteen Design Patterns are used as integral part of the learning process. That is why here and throughout the thesis it is assumed that designers learn patterns prior to design process.

Adaptation of the DP Approach to the Experimental Framework of this Study

To test the reuse of abstract algorithmic solutions in architecture, this study used the thirteen patterns for parametric design, developed and illustrated by Robert Woodbury (2010). In his book 'Elements of parametric design' Woodbury states that designers who use parametric modelling tools tend to create algorithms anew, rather than reuse them

(Ibid). The idea of design patterns is that instead of solving each new problem individually, architects can reuse the generalised algorithms (patterns) of existing, successfully implemented in the past, solutions (Gamma, Helm, Johnson, Vlissides, 1994). Patterns refer to the solutions, described with a high level of abstraction. This way design patterns can be individually interpreted depending on a particular design context. In Woodbury's book and a website dedicated to the patterns for parametric design (Designpatterns, 2014) each of the design patterns is explained using the 'Name', 'Intent', 'Use When', 'Why' and 'How' and is illustrated by a set of samples (specific solutions), which are shown as a sequence of images.

The following example of the 'Reactor' pattern and its sample algorithm (Circle Radii and Point Interactor) illustrates the structure of the patterns' documentation (Exhibit 2.6).

Design Pattern: 'Reactor' (Name: Reactor)

- Intent: 'Make an object respond to the proximity to other object' (Woodbury, 2010)
- When: 'Use this pattern, when you want to make an object respond to the presence of other object' (Ibid)
- Why: Designers often use the metaphor of response, when one part of a design (result) depends upon the state of the other (interactor)'. For this particular pattern the proximity (reference) factor drives the response (Ibid).
- How: 'Connect an interactor to a result through a reference'(Ibid)

'Circle Radii and Point Interactor' is one of the samples of the 'Reactor' pattern illustrating the idea behind this design pattern (Exhibit 2.6). Pattern samples are documented using the following structure:

Sample 'Circle Radii and Point Interactor' (Design Pattern 'Reactor')

- Use When: Control the size of a set of circles by a proximity to a point.
- How: As the interactor point moves closer to the circle, the circle gets smaller (Ibid).

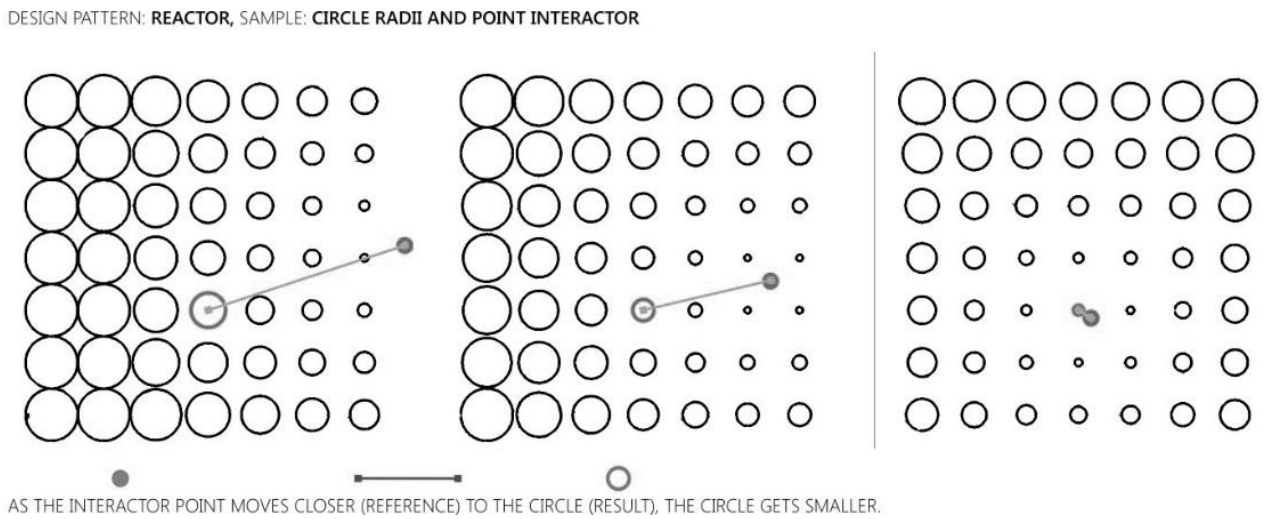


Exhibit 2.6. Diagrams illustrating Design Pattern: Reactor, Sample: Circle Radii and Point Interactor

It is very difficult to underestimate the role of samples in understanding the essence and principles of each abstract solution (design pattern). The samples perform a crucial role, illustrating the idea behind each abstract theory of the design patterns. During the experimental stage of the study, most of the design patterns' samples, suggested and explained by Robert Woodbury (Woodbury, 2010) were developed as Grasshopper definitions. These definitions were analysed to determine which particular patterns work better with which programming logic and components.

All patterns were organised in a specific order to be introduced in the course of the workshops (See Appendix A and Recommendation section for more details on the proposed curriculum to teaching

programming using Design Patterns). During the two days of algorithmic modelling workshops participants were introduced to all thirteen Design Patterns and were shown how to implement them in practice (on the examples of practical step-by-step tutorials). Three samples per design pattern were shown and explained through corresponding programming algorithms during the DP group workshops. One programming algorithm per design pattern was used in a step-by-step practical tutorial.

As a part of the Design Patterns workshop preparation eighty pattern samples were developed as Grasshopper definitions; over thirty of those algorithms were shown to the DP workshop participants. The DP sample algorithms were not made available to download for the Design Pattern test group. This was done to clearly separate and test the reuse of abstract solutions (DP) and the reuse of specific solutions (CBD). As it often stated: samples are meant to be used only as the illustrations for the design patterns (Woodbury, 2010) (Gamma, Helm, Johnson, Vlissides, 1994). If these algorithm were made downloadable they could have been reused through the 'copy/use' principle of the CBD approach. This might have blurred the differences between the approaches and altered the results. That is the other reason why the DP group participants were not given an access to all the algorithms built for patterns samples.

Adaptation of the CBD Approach to the Experimental Framework of this Study

The Case-Based Design (CBD) approach is based on the reuse of design solutions from specific design cases. In the context of this study the CBD approach refers to the reuse of algorithmic solutions in architecture. This approach was tested using an online data-base system,

specifically developed for this study, which contained over one hundred and fifty programming solutions (cases) (Exhibit 2.7). The primary purpose of these reusable solutions was to help designers and architects to solve their own (similar) design problems (Maher, de Silva Garza, 1997). In various fields, including architecture and software programming, the use of Case-Based Design approach proved to be an effective method, helping designers and developers to solve problems by reuse of previous solutions and experiences (Kolodner, 1991) (Aamodt, Plaza, 1994) (Riesbeck, Schank, 2013).

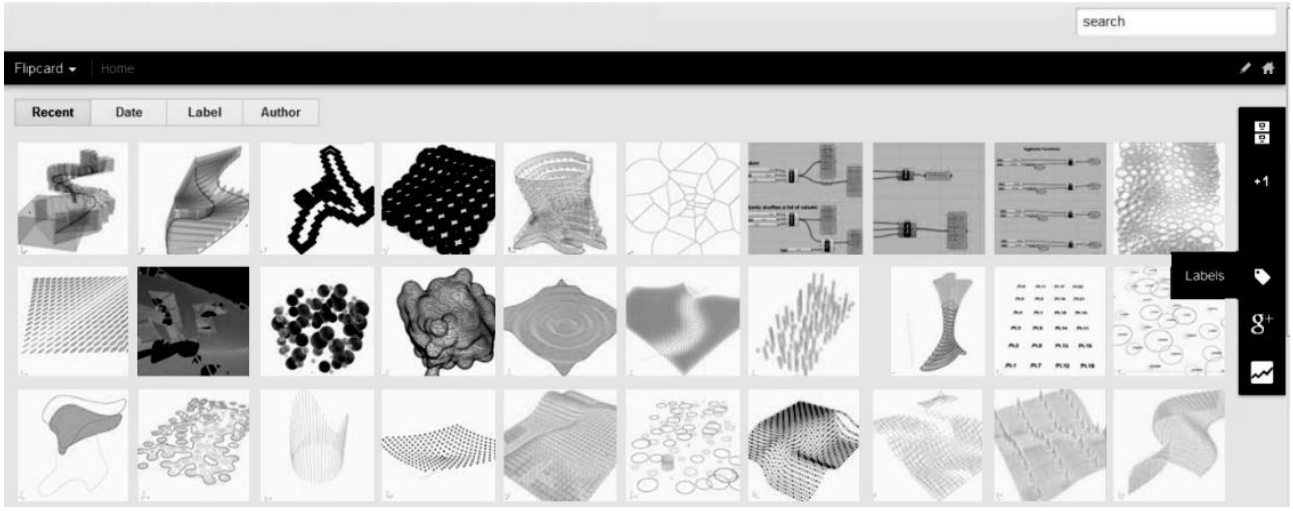


Exhibit 2.7. Snapshot of the Case-Base of algorithmic designs, used as a test the CBD approach. Left side: Search bar; and Action bar containing the Blog Archive and programming solutions indexes ('Labels'), sorted according to the frequency of use

Among the main aspects taken into consideration when designing a CBD system for this study are the following points:

- how the design solutions are going to be represented;
- what the process is for selection and retrieval of solutions; and
- what the process is for adapting design solutions (Maher, de Silva Garza, 1997).

In many ways, the representation of a design case can be understood as an abstraction, communicating the essence of each design, interpreted into a symbolic form that any designer or architect can understand (Ibid). To give participants of this experimental study an opportunity to see and understand, how a resulting geometry reacts to changes in parameters, the images, representing an output geometry of each case, are animated using Graphics Interchange Format (gif). In addition to geometry related animations (Exhibit 2.7), each solution is also represented with a snapshot of its source Grasshopper definition, to allow users to 'read' (comprehend) the programming logic behind each design case.

The developed Case-Base for algorithmic solutions (testing the CBD approach) was an online database system. This system was organised based on the indexes assigned to each solution, which are used to sort and retrieve reusable items. Systematic and adequate structuring of the CBD database content was essential to ensure effective selection and retrieval of solutions. That is why various features (characteristics) of algorithmic designs were addressed by indexing, including: a) design concept features; b) geometrical/shape features; and c) programming logic characteristics. Up to twenty indexes were assigned to each design case to allow participants engage with various search features for recalling cases.

To ease the process for adapting design solutions, each case in the developed CBD system had a corresponding downloadable Grasshopper files, to allow direct 'copy/use' or 'copy/modify' option.

The CBD approach was tested on a database of programming solutions, specifically developed to accommodate the scope and needs of the algorithmic modelling workshops.

The following principles were established to guide the development of Case-Base for algorithmic solutions (informed by the context of this study):

- Keep algorithms relatively simple, as a) participants are expected to spend only two hours on the development of their conceptual models; and b) most participants did not have advanced enough skills with Grasshopper, to tackle complex programming solutions)
- Develop solutions that explore different programming and form – making logic, explained during the course of the workshops; (to allow participants to expand the space of explored algorithmic solutions)
- Complex projects are to be divided into independent parts or segments

Prior to development of the final version of the online Case Base system, used to test the CBD approach, three online blog platforms (web publishing tools) were tested as a means to host the repository of algorithmic designs, including: Blogger (Blogger, 2014), Tumblr (Tumblr, 2014) and WordPress (Wordpress.com, 2014). All three of these platforms allowed images and programming algorithms to be published and shared; all allowed multiple indexes to be applied; and selected (search) solutions to be used based on those indexes. To determine which platform suited this study, the best of a hundred of algorithmic solutions were uploaded to each of the blog platforms and were made available to be viewed online (worldwide). After four months, the number of visits to each blog was compared between three platforms. The Blogger platform appeared to be the most popular compared to Tumblr and WordPress. That is why the Blogger platform was used to host the Case-Base of algorithmic

design solutions and all one hundred and fifty algorithms developed to test the CBD approach were uploaded to the online data-base system.

During the first day of the workshop the CBD group participants were given a link to the online Case-Base of algorithmic designs and were explained how to select and retrieve solutions from the database. When developing their design tasks the CBD group participants were provided with constant access to this database, so they could select and reuse any of these solutions.

To use the CBD system, participants were expected to identify specific features (indexes) characterising their design idea in order to find a similar solution within a database. Each programming solution was represented with illustrations, so designers could visually search for a solution using animated images. This way the CBD users could potentially find a visual match to the originally sketched design concept, available in the repository. If a fitting CBD solution was identified, a designers could check a corresponding programming algorithm (by downloading its Grasshopper file or using a snapshot of the Grasshopper definition). This allowed participants to understand how the algorithm worked and to decide whether they wanted to reuse a particular solution, following the 'copy/use' or 'copy/modify' method.

2.2 Evaluation of the approaches

Research methodology

The proposed methodology has been drawn from a range of studies which have examined the application of CAD technologies through case studies (Celani, Vaz, 2012) (Hamade, Artail, 2008) (Shah, Smith, Vargas-

Hernandez, 2003) (Groat, Wang's, 2002) (Toth et al, 2011). The criteria relating to the fluency and novelty of design ideation were informed by the work titled 'Metrics for measuring ideation effectiveness' (Shah, Smith, Vargas-Hernandez, 2003). The experimental setup was influenced by the recent and relevant research work by Gabriela Celani and Carlos Vaz (2012): 'Cad Scripting and visual programming Languages for implementing computational design concepts'. The overall methodology was drawn from Groat and Wang's (2002) guidelines for the development of experimental studies: a carefully controlled study with at least two groups, random selection of participants, no systematic differences between groups, and with the same treatment applied for all groups.

After careful consideration and comparison between research objectives and the relevance of available methods (which dealt with design process) it was decided that the experimental methodology suited this study best. There were several experimental methods to study and evaluate design processes such as: controlled tests (Schon, 1991), protocol studies (Christiaans H. and Dorst K., 1991), (Sobek and Ward, 1996) and case studies (Ericsson, K and Simon, H, 1984). Case study analysis (namely students' design works, which was produced during algorithmic modelling workshops) and surveys reporting participants' experience meeting all the research requirements and objectives and therefore were chosen as most suitable.

The data gathering methodology was based on two types of approaches:

- Outcome-based analysis (Shah, Smith, Vargas-Hernandez, 2003);
- Questionnaires

The data (values for each identified criterion) obtained from the questionnaires and outcome-based analysis was used to compare

whether and how each criterion varied depending on designers' use of the Design Patterns and Case-Based Design approaches. Most of the collected data was interpreted as numeric values (metrics), allowing explicit comparison between the approaches (See Statistical Analysis section). This allowed the use of empirically obtained results as a means to determine the answers for the research questions (See Research aims and Objectives section).

Metrics measuring the key aspects of algorithmic design performance

Metrics measuring the key aspects of algorithmic modelling in architecture were based on criteria developed to accommodate the research objectives of this comparative study. These criteria were divided into five groups:

- Programming criteria;
- Design ideation criteria;
- Motivation criteria and;
- Approach characteristics criteria;
- Algorithmic modelling criteria (metrics for measuring qualitative aspects of algorithmic models and programming solutions);

The questionnaires also had a design background section, where respondents indicated their level of experience in architecture and design; as well as their experience with computational design tools and specifically the use of Grasshopper 3D for Rhino. Furthermore participants were asked to indicate their gender. These characteristics were used as covariates,

testing whether experience or gender had any significant influence on the results (See Statistical Analysis Section).

Programming criteria

The first evaluation metrics group covered such programming criteria as: programming difficulties, learning curve and reuse of algorithmic solutions. Programming difficulties criteria referred to how often participants came across programming difficulties, while developing their algorithmic designs; and what type of difficulties they had. Learning curve criteria evaluated how often participants implemented new components while developing their algorithmic designs. The reuse of knowledge concerned how often participants reused algorithms from any external sources, such as the CBD repository or other locally or internet based sources.

Number of programming difficulties (barriers) [Questionnaire]*

**Method of information extraction*

Participants were asked to indicate how often they had come across programming difficulties (barriers) which they could not overcome. The study took into account the fact that almost every problem or mistake could eventually be solved (corrected). That is why the cases when users spent a significant amount of time on solving a particular programming issue (more than 30 minutes out of 2 hours given for the development of a task) were reported as a programming difficulty. The answers were gathered as numeric values (metrics).

Types of programming barriers [Questionnaire]*

In order to investigate the typology of barriers that designers face when they used algorithmic modelling tools, participants were asked to report their difficulties. This question was set as an open ended type of enquiry,

meaning that participants had no predefined options or categories. Afterwards, these responses were analysed and sorted into the most re-occurring categories (See Findings Section).

Learning curve [Questionnaire]*

The amount of times that participants took to implement a new (never used before / not explained in the tutorials) programming component. The answers were reported as numeric values (metrics).

Reuse of solutions [Questionnaire]*

This criterion measured how often participants had re-used algorithms or parts of the algorithms from any external sources while developing their own programming solutions. It referred to cases when participants had re-used existing algorithms or parts of the algorithms (copy/paste/modify approach), including the re-use of algorithms shown during the workshop tutorials. The answers were reported as numeric values (metrics).

Design ideation/performance criteria

The Design Ideation Criteria group investigated how the use of the Design Patterns and the online Case-Base of algorithmic solutions affected design thinking. This included: change in design objectives, participants' ability to realise their original design ideas, ability to accomplish all that was wanted etc. These criteria explored how each approach affected the design process and the participants' feedback regarding the 'achieved' versus 'intended' was evaluated. The Secondary aim of the design ideation criteria was to evaluate the degree to which each approach was likely to affect (alter) a design outcome (result compared to the initial design intent). Due to participants' lack of experience with programming environments (programming barriers) it was expected that the initial idea would often be modified.

In order to better understand the ways architects and designers think about their design models, workshops participants were asked to describe different aspects of their designs using:

- the key words (indexes) related to geometry / shape of their designs;
- metaphors and abstract attributes that characterised their models;
- the key words related to algorithmic modelling;

The index (key word) study aimed to determine the effective ways to structure a repository of algorithmic design solutions (cases) for architects and designers. This investigation provided an insight into how one could organise and label a database of algorithmic solutions in a more effective way. The response, indicating the type of design objectives that participants had was reported as an open-ended type of answer with no predefined options or categories. The rest of the responses for the design ideation criteria were reported as closed-ended answers indicating the level of agreement with the statements on a five point scale (Celani, Vaz, 2012) (See Statistical Analysis Section for more detail regarding the answer scales and types of questions).

Change in the design intent [Questionnaire]*

- Ability to model original idea
- Change in the design strategy due to programming difficulties
- Change in the design strategy because participants found interesting solutions, which they decided to reuse;
- Design objectives (What participants intended to accomplish);
- Ability to accomplish what was intended/wanted;

Satisfaction/Motivation criteria

The motivation criteria group evaluated the degree of satisfaction with the design outcome and motivation to use algorithmic modelling in the future. The objective was to compare results and identify whether there was any dependency between the levels of satisfaction with output/motivation to use algorithmic design tools in future and the use of each approach. The responses were reported as closed-ended answers indicating the level of agreement with the statements on a five point scale (Celani, Vaz, 2012) (See Statistical Analysis Section).

Degree of satisfaction/motivation [Questionnaire]*

Degree of satisfaction with the design output and motivation to use algorithmic modelling in future

- Level of satisfaction with the design outcome
- Motivation to use algorithmic modelling tools in future

Approach characteristics criteria

The approach characteristics group referred to the usability, intuitiveness, flexibility and utility criteria, which were identified to represent the overall features related to the use of each approach. Usability was how easy it was for participants to learn/ implement the Design Patterns and the Case-Based Design approaches. Intuitiveness attributes were how intuitive participants found each approach. Flexibility (re-usability) referred to participants' ability to find and adapt a Design Pattern or a CBD solution which fitted their design concept; and how often participants actually implemented Design Patterns or CBD solutions in their designs. Utility related to how helpful participants found each approach. All approach

criteria except 'Flexibility' (how often participants re-used algorithmic solutions) were collected with a five point scale level of agreement with the statement (Celani, Vaz, 2012) (See Statistical Analysis Section). When reporting how often participants re-used Design Patterns or Case-Based solutions from the online repository, they entered numeric values (metrics).

Usability [Questionnaire]*

How easy it was for participants to learn/ implement the DP and CBD approaches.

Intuitiveness [Questionnaire]*

How intuitive participants found each approach.

Flexibility [Questionnaire]*

- Ability to find and adapt a Design Pattern or a CBD solution, which fitted participants' design concepts;
- How often participants implemented Design Patterns or CBD solutions in their designs;

Utility [Questionnaire]*

How helpful participants found each approach.

Algorithmic modelling performance criteria

This group of evaluation criteria referred to algorithmic modelling performance in general and can be applicable for various experimental frameworks. These criteria can potentially be used as a metric for measuring qualitative aspects of algorithmic models and programming solutions in architecture and design. The focus of the metrics was

evaluation of algorithmic modelling performance in the context of the early stages of design (conceptual models) where the emphasis was on ideation and the qualitative aspects of the models produced.

The objective of the metric was to provide a means of systematically:

- categorising models according to their complexity;
- ranking the complexity of the algorithms used to generate output geometry;
- evaluating the explored solution space of programming solutions (algorithms) as evidenced by variety and novelty.

Only one of the metrics was limited to the visual programming context. That was the method of evaluating the complexity of the algorithms (Grasshopper definitions). The variety and novelty criteria, which formed the explored solutions measure, focussed on the programming components, but the overall logic was suited to both textual and visual programming. The measure of model complexity was widely suited to the general evaluation of geometrical complexity of architectural and design models.

Model Complexity [Output Model Evaluation]*

Various approaches measuring output model complexity were investigated, including: considering meshes to have distinguishable shape characteristics; Shape Grammar; and measuring the complexity of shapes and representation (Mitchell, 1990). From this, a point system for determining complexity was developed. It was informed by geometrical, combinatory and dimensional criteria for 3D model classification. In the context of this study, this measure was used to determine the speed of modelling because it was assumed that a more complex model developed within a given period of time required a greater modelling speed.

3D models can be created with various form-making algorithms and operations, but final representation is usually stored in the form of polygonal meshes (Shikhare et al., 2001) or NURBS (Non-Uniform Rational B-Splines). That is why one of the approaches is to consider meshes to have distinguishable shape complexity characteristics (Garland, 1999). An alternative approach to classifying 3D models is based on measuring the complexity of shapes and representation of a model (Forrest, 1974) (Stiny, 2008) (Krishnamurti, 2011). Forrest suggested three types of model classification: geometric, combinatory and dimensional. Geometric complexity refers to the models basic elements; such as lines, planes, curves, surfaces, etc. Combinatorial complexity considers the number of component (elements) and dimensional complexity classifies model as a 2D, 2.5D or 3D model. The other method to analyse models refers to Shape Grammars. The Shape Grammars approach interprets a model as a set of rules (Heisserman, 1994). Shape grammars can be considered to be visual mathematics. This method argues that a design can be seen as series of transformations, such as rotation, translation, reflection, scale (Cui J, MX Tang, 2013). The Shape Grammar design method is based on form computation and logical analysis of the formal properties (Heisserman, 1994). In practice, it can be applied using methods of shape decomposition into basic components (actions).

The point system, which formed the criterion measuring complexity of geometric models for this research, were informed by the combination of geometrical, combinatory and dimensional complexity criteria for 3D model classification; as well as the form computation mechanism of a design – Shape Grammars.

Model Complexity Evaluation: Point System

The method for measuring complexity of algorithmic models is based on a point system. Numbers in [N] brackets were the score points. Each model was analysed according to the following seven categories: Basic Elements, Composition Space, Arithmetic of Shapes, Number of Elements, Shape of the Element Transformations and Colour. Each model was awarded a certain number of points in each category. The total number of points was combined to form the final score.

The *Basic elements* category evaluated models according to how advanced the geometry was, starting with the simplest geometry – points and ending with most advanced – solids (Forrest, 1974). In many cases outcome models, submitted by participants, had various types of elements: points, lines, surfaces and solids. In some cases, all elements (including intermediary geometrical structures, such as centre points and surface edges) were kept visible. In other cases only the resulting geometry was left visible. That is why the points were not awarded to all the types of elements of the model, but only to the most advanced type of element geometry. Six types of basic elements geometry were identified (from simple to complex): 'Points' (a point can be defined by XYZ coordinates), 'Lines' (a straight line; can be defined by two points), 'Curves' (a curved or straight line, can be defined by two, three or more points. It includes all splines such as polylines, curves, interpolated curves; and primitives such as: circle, ellipse, rectangle and polygon), 'Planes' (a flat, two-dimensional surface), 'Surfaces' (three-dimensional open surface) and 'Solids' (a solid three-dimensional geometric figure (includes closed surfaces)) (Exhibit 2.8).

Basic elements (Geometrical Complexity): Points – [0]/Lines – [1]/Curves – [2]/Planes – [3]/Surfaces – [4]/Solids – [5] (Exhibit 2.8)

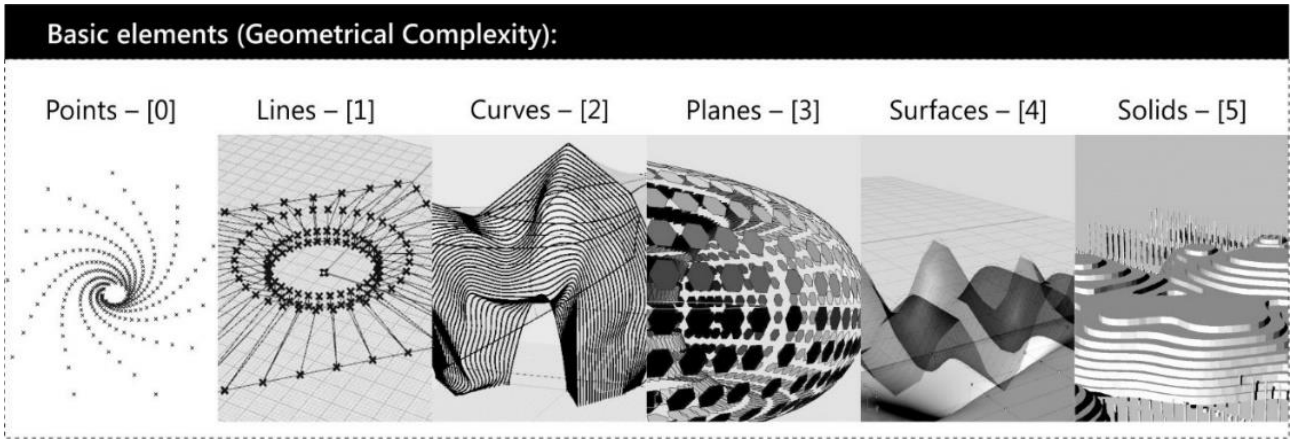


Exhibit 2.8 Basic elements (Geometrical Complexity)

While basic elements geometry could be two or three dimensional the distribution (composition space) of those elements could also vary. Two types of spatial compositions were identified: 2D Composition - a flat, two-dimensional distribution of elements and 3D Composition – a three-dimensional distribution of elements. In this category, more dimensions mean more complexity, that is why 2D compositions were awarded [0] points and 3D compositions were awarded [1] point.

Composition Space (Dimensional complexity): 2D – [0]/3D – [1] (Exhibit 2.9)

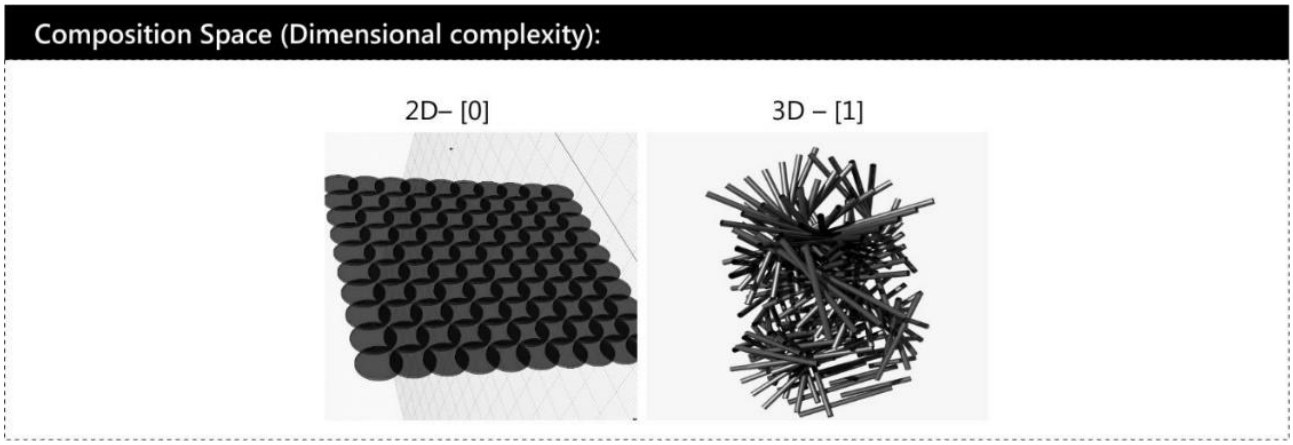


Exhibit 2.9 Composition Space (Dimensional complexity)

Arithmetic of Shapes was a category concerning operations which could happen when geometrical shapes intersected. They were often

referred to as Boolean Operations or when elements had been culled according to a mathematical function or condition. 'Addition' (+) is an operation of transformation of two or more intersecting objects into a single object, such as the union of Region, Mesh or Solid. 'Subtraction' (-) is an operation that is opposite to 'Addition' and occurs when intersecting objects are being deducted from one another (such as Curve, Surface or Solid Trim and Region, Mesh or Solid Difference). In Grasshopper 'Cull Pattern' is an operation of selecting certain elements and deleting or transforming them, such as Cull Index, Cull Pattern (true/false), and Random Reduction etc. These operations were also referred to as arithmetic of shapes (custom type of subtraction or addition). As the Evaluation method of model complexity was based on visual analysis of models, it was often difficult or next to impossible to define if an 'Addition' operation has been performed. In many cases, when several shapes or volumes intersected they formed a complex geometry and it was difficult to tell if they had been transformed into a unit or if they were separate and just intersecting. That was why, in order to avoid confusion, 'Addition' operations were given [0] points. 'Subtraction' operations were given [1] point and 'Cull Pattern', and a more complex function, was given [2] points.

Arithmetic of Shapes (Shape Grammars): Addition – [0]/Subtraction – [1]/Cull Pattern (Reduce or add elements according to a certain logic) – [2] (Exhibit 2.10)

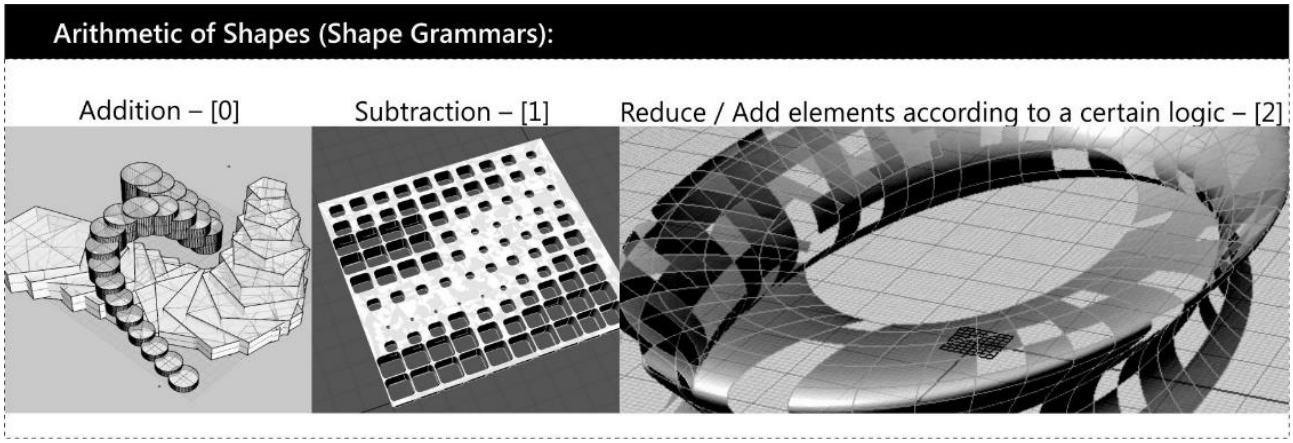


Exhibit 2.10 Arithmetic of Shapes (Shape Grammars)

The *Transformation* category was closely related to the 'Arithmetic of Shapes' category, as it also dealt with operations. Transformations were divided into five clearly identified types: Scale, Rotation, Reflection, Deformation and Translation (Cui J, MX Tang, 2013). Each type of transformation was given one point. In many cases a combination of transformations took place, where elements of the model were both rotated and scaled. 'Scale' is a type of transformation which deals with the elements size change. 'Rotation' is the process of turning the element around a centre or an axis. 'Reflection' is a type of transformation in which one element is the mirror image of the other. 'Deformation' includes a variety of operations dealing with shape changes, such as Bend, Twist, Blend and Morph. 'Translation' is the process of moving an object from one location to another. In practice, when looking at the resulting model, it is near impossible to tell for certain if an object has been moved (as a copy) or if the same objects have been generated in different locations. That is why, to avoid all uncertainties regarding the type of underlying modelling logic, in the cases where the same elements (same type of elements) had reoccurred in different locations it was considered to be a 'Translation'.

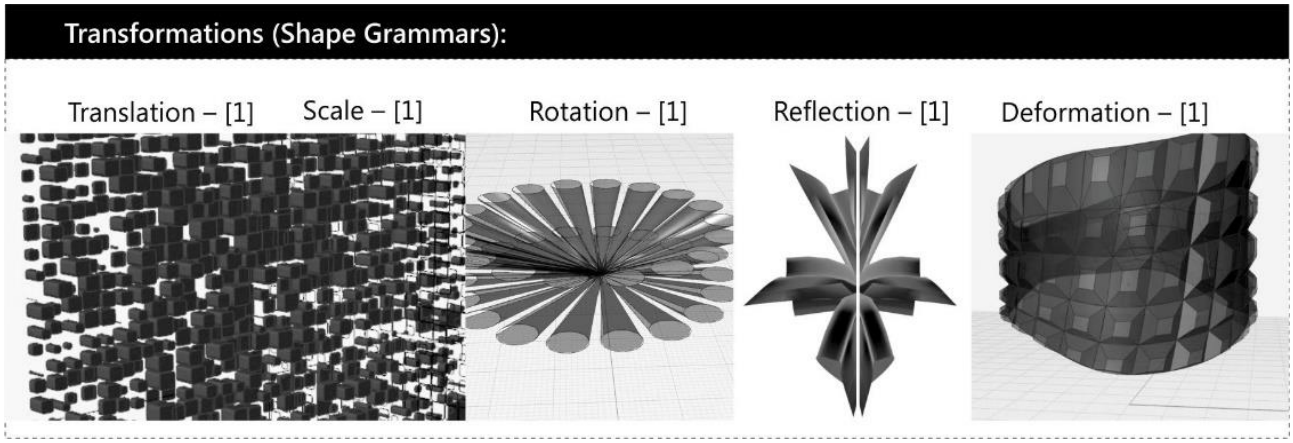


Exhibit 2.11. Transformations (Shape Grammars)

Transformations (Shape Grammars): Scale – [1]/Rotation – [1]/Reflection – [1]/Deformation – [1]/Translation – [1] (Exhibit 2.11)

Number of Elements categorises models into four types of groups. The first group, 'One Element' (where a model has only one element) is considered to be the most simple – [0] points. The second group of models are those that have from two to ten elements of the same type (for example, nine cylinders) – [1] point (Exhibit 2.12). The Third group is 'Multiple Elements', when a model has more than ten elements and they have the same type (for example, a structure composed of hundreds of pipes) – [2] points (Exhibit 2.12). The last group in this category 'Multiple elements N Types', where 'N' stands for a number of types of elements (for example, when a model contains planes, surfaces and different types of solids). The score for this group was calculated according to the following expression: [X= N +1] points, where N stands for a number of types of elements.

Number of Elements (Components): One Element – [0]/Two-Ten Elements – [1]/Multiple Elements (one Type) – [2]/Multiple Elements ('N' Types) – [1 +'N'] (Exhibit 2.12)

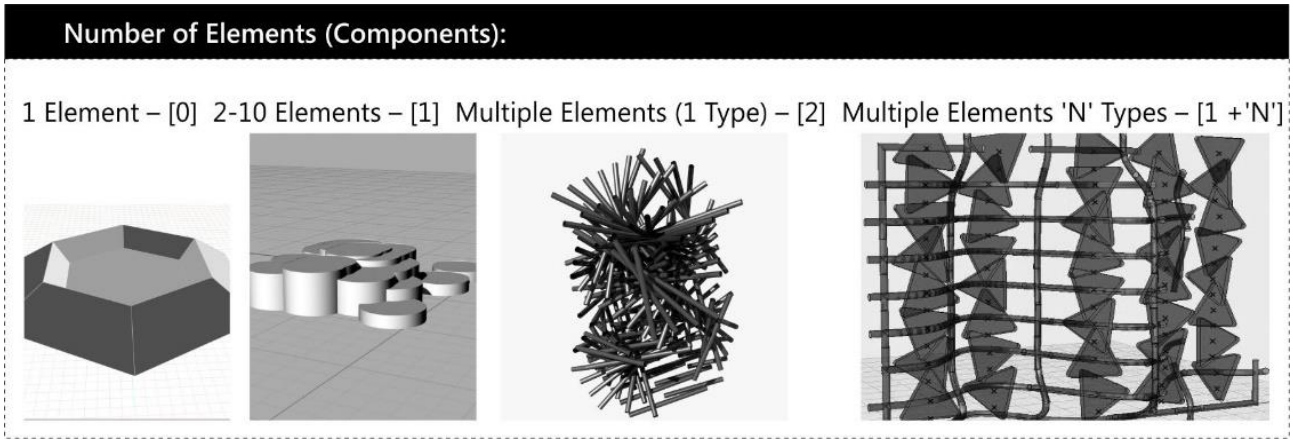


Exhibit 2.12. Number of Elements (Components)

The *Shape of the Elements* category evaluated the characteristics of elements of a model. When 'Standards and Primitives' were used (such as a circle, cube, sphere etc.) [0] points were awarded. In cases where a certain type of element(s) had a repeating 'Non-standard Shape' (such as rhombus shaped panels with filleted corners) [1] a point was awarded. The third group included elements which had a non-repeating nature, (for example, extruded sections or non-standard shaped objects). These were referred to as 'Complex Shape' elements and were given [2] points.

Shape of the Element: Standards and Primitives – [0], Non-standard Simple Shape – [1]/Complex Shape – [2] (Exhibit 2.13)

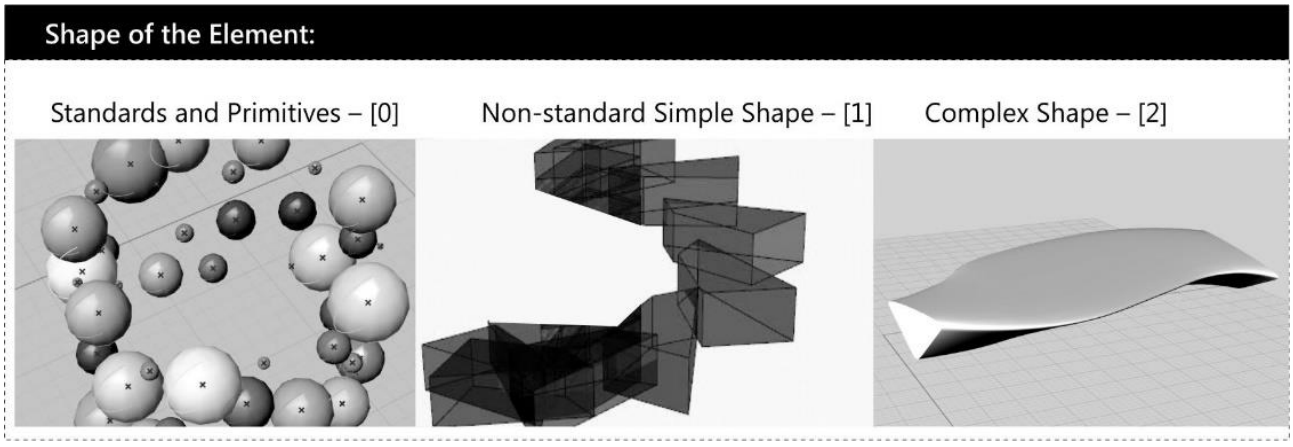


Exhibit 2.13. Shape of the Element

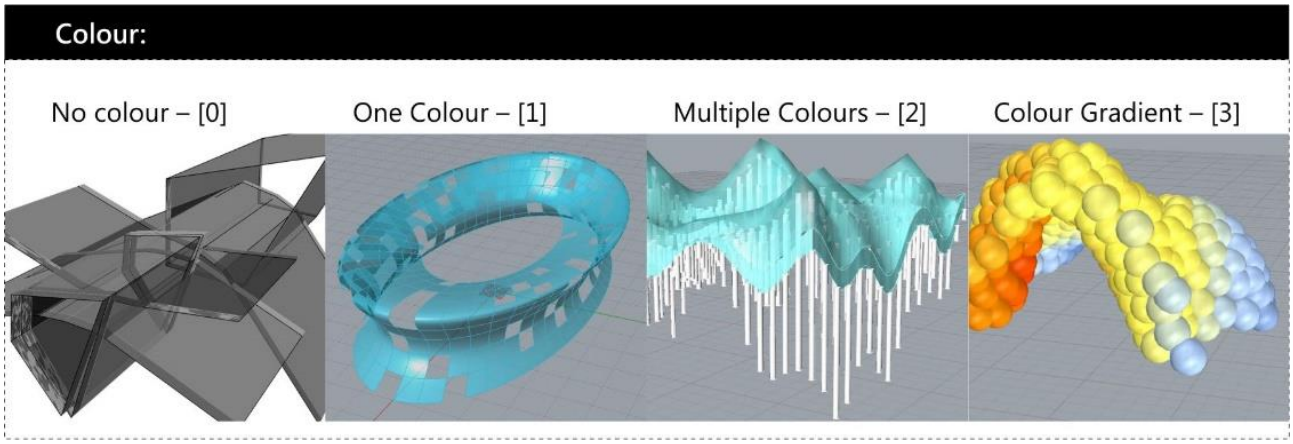


Exhibit 2.14 Colour

Colour: No colour – [0]/ One Colour – [1]/Multiple Colours – [2]/Colour Gradient – [3]

The final category dealt with the use of *colours* (shades) in the model. The first group of models were models with 'No colour', which were given [0] points. When at least 'One colour' was used the model was given [1] point. Models which had 'Multiple Colours' was given [2] points. When complex shading materials or 'Colour Gradients' were used, it was given [3] points (Exhibit 2.14).

The total Model Complexity score was calculated as a sum of all the scores that a model got in each category including: Basic Elements, Composition Space, Arithmetic of Shapes, Number of Elements, Shape of the Element Transformations and Colour. All Model complexity score calculations were done using Excel tables (Exhibit 2.15) (Microsoft Excel, 2014).

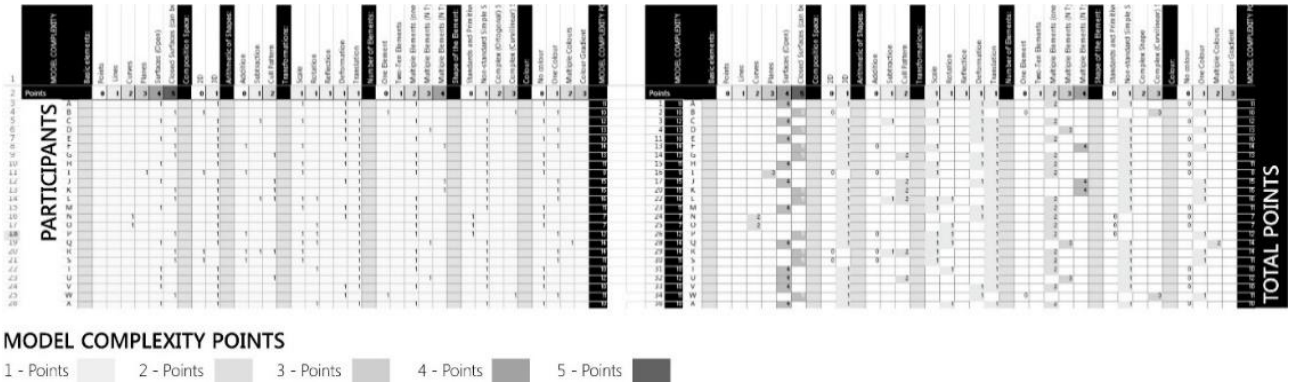


Exhibit 2.15. Model complexity evaluation Graph (Excel table) Example. Control group. No Approach

Columns in Exhibit 2.15 referred to the categories such as Basic Elements, Composition Space, Arithmetic of Shapes etc., and rows referred to models developed by participants on each day.

Prior to calculating the scores following the logic of the model complexity point system, all models were sorted into five groups from most simple to most complex, according to visual comparison (personal judgement). The majority of models in both types of analyses (personal judgement and complexity point system) matched the complexity group choice. Visually simple models – got lower model complexity scores and visually complex models got higher model complexity scores. Although, a fair number of models were within the middle of the spectrum of complexity (according to the model complexity point system) they appeared to be more complex than anticipated. Some models scored more points than expected and were sorted into groups with higher complexity. The overall conclusion was that this model complexity point system was an adequate method to evaluate complexity of models.

These metrics were successfully implemented as a practical method to evaluate the complexity of output models developed by participants of the algorithmic modelling workshops using Grasshopper

for Rhino. In theory, these metrics were applicable for any geometric models including, virtual and physical.

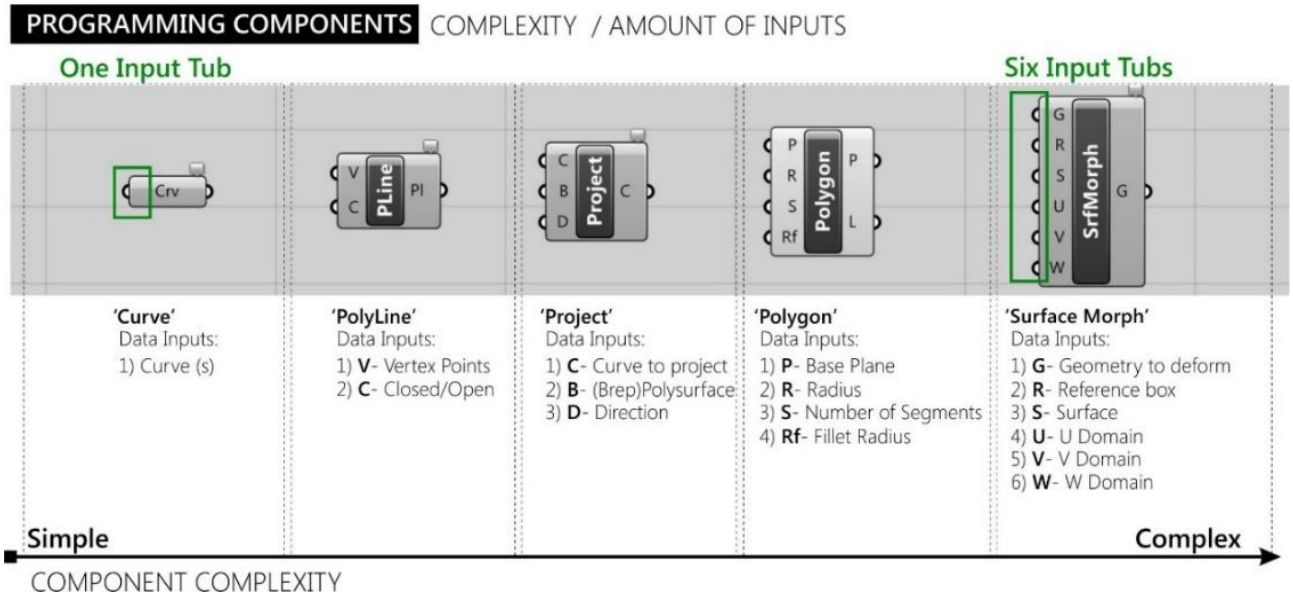


Exhibit 2.16. Algorithm complexity evaluation. Programming components. Inputs vs Complexity points

Algorithm Complexity [Grasshopper definition analysis]*

The evaluation of the degree of algorithm complexity was based on the analysis of the Grasshopper definitions (programming algorithms). A second proposed points system was utilised. Points were awarded to each input tub (See Exhibit 2.16) of each component used in a programming algorithm (Grasshopper definition). The logic behind this type of evaluation was that the more inputs/variables a component required the higher its degree of complexity (as illustrated in Exhibit 2.16).

The sum of the inputs of all components implemented in a Grasshopper definition formed a total Algorithm Complexity score. Similar to Model Complexity score, the calculations for Algorithm Complexity criterion were done using Excel tables (Exhibit 2.17) (Microsoft Excel, 2014)

Columns in Exhibit 2.17 referred to programming components, with corresponding complexity points (number of inputs) sorted from most simple components (left) to most complex components (right); rows

referred to the algorithms developed by participants on each day of the workshops.

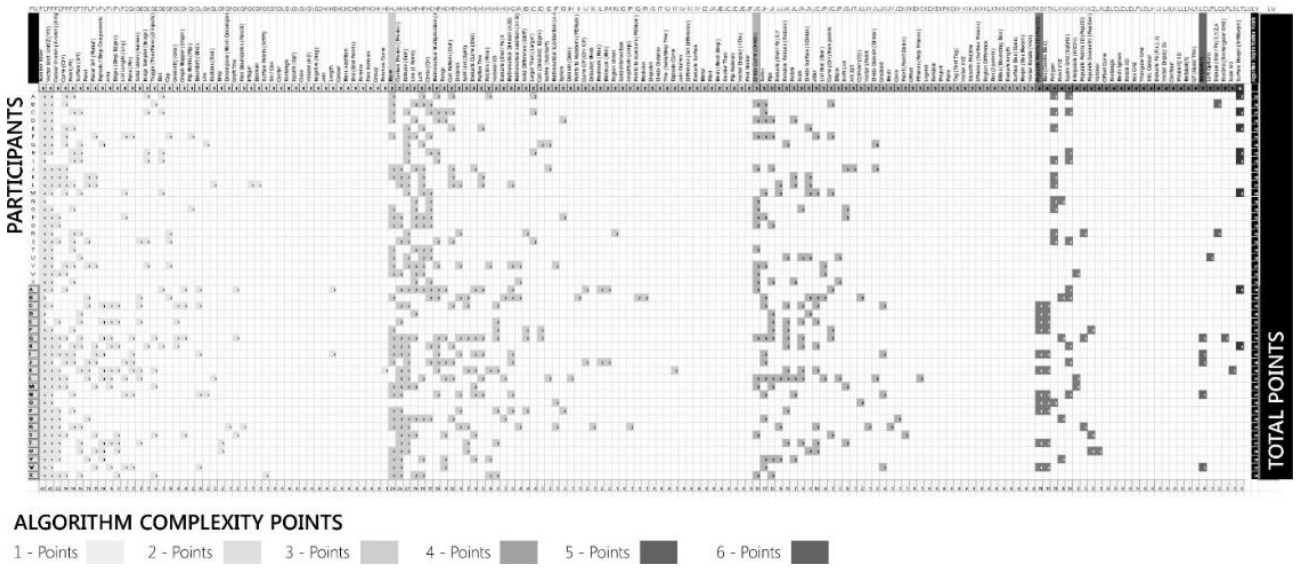


Exhibit 2.17 Algorithm complexity evaluation Graph (Excel table) Example. Control group. No Approach

Explored Space of Algorithmic Solution [Grasshopper definition analysis]* The third algorithmic modelling metric sought to evaluate the explored space of algorithmic solutions developed by the workshop participants. Given the context of the early stages of design two criteria were identified to evaluate the boundaries of explored solution space: variety (range of explored solutions) and novelty (how original a solution was compared to the pool of algorithmic solutions). The methods of measuring these criteria were informed by research work 'Metrics for measuring ideation effectiveness' (Shah, Smith, Vargas-Hernandez, 2003).

Variety refers to a range of unique programming components used during the design generation process. The bigger the count of various programming components used by participants, the higher the variety score.

Novelty refers to how unusual a programming algorithm is compared to other programming solutions, developed during the course of the workshops. In order to measure a novelty of an individual algorithmic solution it was necessary to work at a group level (all test groups). During the first stage, all algorithms developed by participants were analysed based on how often each programming component (logic) was used throughout the course of the workshops. After that each component was awarded a novelty score (from 0 to 10, where 0 points indicates not novel logic/frequently used by participants; and 10 points indicates a very novel programming logic/rarely or never used by others) (Exhibit 2.18).

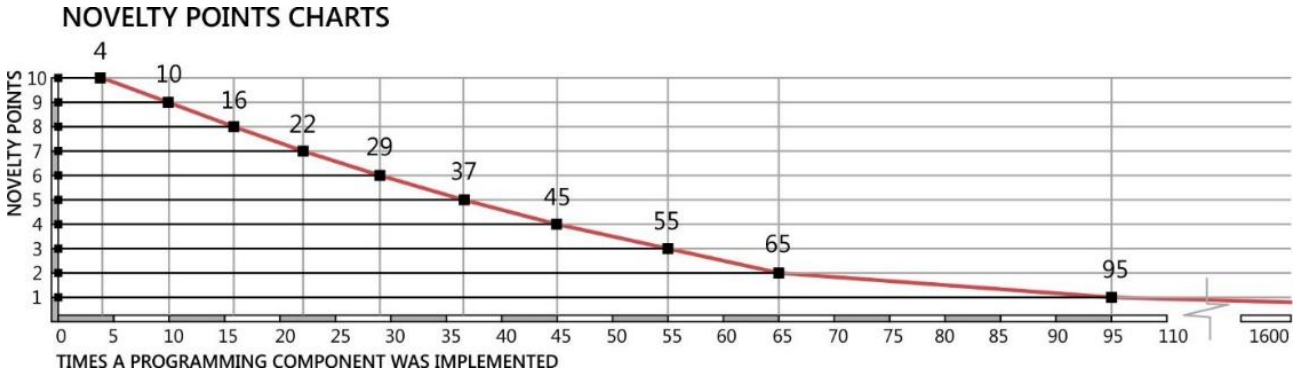


Exhibit 2.18. Novelty points chart (Programming Algorithms Analysis)

Exhibit 2.18 illustrates the distribution of Novelty scores and corresponding number of times a component was implemented. For example, if a programming component was used only 4 or less times by all 126 participants during two days of the workshops, it was given 10 novelty points. Components implemented from 5 to 10 times were given 9 novelty points, components used 11-16 times were given 8 novelty points; and so on (Exhibit 2.18). Most frequently used programming components used within the range of 100 to 1660 times get 0 Novelty points, such as 'Number Slider' components (used 1660 times throughout

the course of the workshops), 'Vector Unit' (used 431 times), or 'Move' (used 294 times) (Exhibit 2.18).

The sum of the novelty scores for each implemented component comprised the resulting total Novelty score of each programming algorithm. The less a characteristic programming component (logic) was re-occurring in the pool of all algorithmic solutions, the higher its novelty (Shah, Smith, Vargas-Hernandez, 2003). The calculations for both Novelty and Variety criteria were done using Excel tables (Exhibit 2.19)

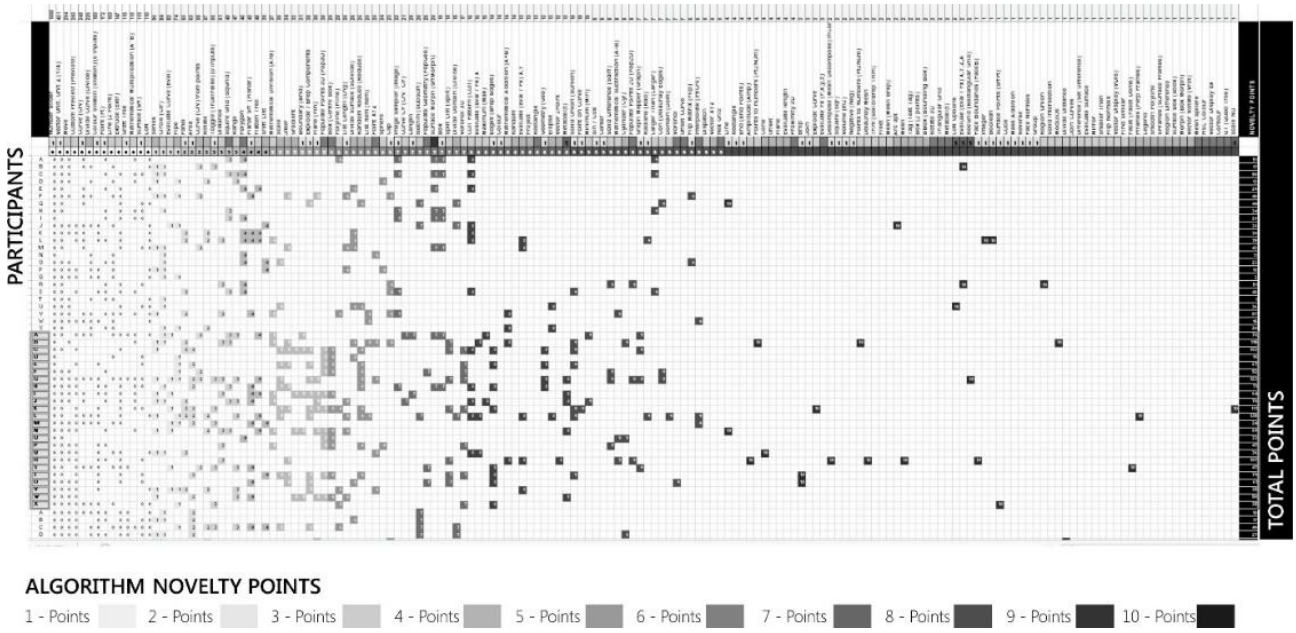


Exhibit 2.19 Algorithm Novelty evaluation Graph (Excel table) Example. All groups.

Columns in Exhibit 2.19 referred to programming components, with corresponding novelty points sorted from most typical/frequently used (left) to most novel/rarely used programming components (right); rows referred to the algorithms developed by participants on each day of the workshops.

See a detailed summary chart of all the evaluation criteria groups in Appendix B, page B55)

2.3 Statistical methods

Sample size

This research was designed as an experimental comparative study between the DP (reuse of abstract design solutions) and CBD (reuse of case-based design solutions) approaches aimed to support designers in use of algorithmic modelling environments in architecture. Both approaches were tested through a series of algorithmic modelling workshops, with at least thirty participants per approach.

Obtaining the appropriate sample size, in our case it is the number of individuals to include in the experimental study, is an important consideration. In theory, the more collected data the better, since increasing the sample size improves statistical power (Martin, Bateson, 1986). Determining the sample size also depends on how much confidence is required and what is the acceptable level of error (Alreck, Settle, 1995). A large sample size ensures that results are representative of the entire population and can be generalised. In statistical testing a large enough sample size is needed to achieve the results that are statistically significant (Mehta, Patel, 1998). The term statistically significant is used as a means to indicate the probability of the results occurring by chance alone. A probability level of 0.05 has been established as a generally acceptable level of confidence (Fisher, 1925). The 0.05 level indicates that there are at least 95 out of 100 chances that the results obtained from the study sample would be similar, when tested on the entire population.

However in practice, the sample size is often limited by both the amount of time required for data collection and the availability or expense of the resources. That is why it is important to determine the 'large enough' minimum of the sample size (Gay, Diehl, 1992). Roscoe's rule of thumb for

determining sample size states, that a sample size larger than thirty and less than five hundred is appropriate for most research cases (Roscoe, 1975). When comparing groups of data, the appropriate sample size of at least thirty participants for each category, that is being compared, is commonly accepted, (Weisberg, Bowen, 1977).

In this comparative study the sample population is split into three groups, which correspond to the Design Patterns (DP) and Case-Based Design (CBD) approach groups and the control group (NA - No Approach), therefore a minimum sample size of thirty for each category is necessary (Ibid). One of the other reasons, which can influence the minimum sample size, is that at least thirty subjects are required to establish a relationship in correlational research (Cohen, 2013) (See 'Dependence between the criteria' section). (See also Appendix B, Exhibit B1. Evaluation Criteria Groups, page B77)

Considering all these requirements, for this experimental study, the sample size of minimum thirty participants per group was adopted.

Collection of data

The population size of at least thirty participants per test group meets the significance level of statistical testing, ensuring that the results of this experiment did not occur by accident. The data collected from the No Approach, Design Patterns and Case-Based Design workshops was produced in three ways: online questionnaires, output design models (virtual Rhino models) and programming algorithms (Grasshopper definitions). This data was analysed according to the five groups of criteria: algorithmic modelling, programming, design ideation, approach characteristics and motivation; which were identified as likely to typify

differences between the two approaches (See Research Methodology/Evaluation Metrics section).

The data obtained from the design models and programming algorithms was used to quantify algorithmic modelling criteria (See Methodology Section), such as output model complexity, complexity of programming algorithms and explored solution space of programming solutions. The data obtained from online questionnaires was used to measure programming criteria, such as amount and typology of programming difficulties, learning curve and reuse of existing algorithms. The questionnaires also provided data for design ideation criteria, which include: types of design objectives, ability to model original design idea, change in the design strategy due to programming difficulties or the discovery of interesting algorithmic solutions. The data from questionnaires was used to measure the approach characteristics criteria, such as: usability, intuitiveness, flexibility and utility; as well as motivation with the design output and motivation to use parametric modelling in future.

Typology of collected data

The data obtained from the online questionnaires, 3D models and programming algorithms was originally recorded, post-factum interpreted or calculated as numeric values/variables (See Methodology section). Depending on the method of measuring the criteria, these variables have different range and distribution. For example; the variable of 'Algorithm Complexity Score' for Design Patterns' (day 2) exists within a range of numbers, which go from 4 to 55 points. Variables for some other criteria had only two possible options: Yes (1) or No (0), for example in 'Types of

Difficulties', where the participants either had a particular type of difficulty (1) or they did not (0).

These variables can be sorted into two types of data classes: continuous variables and categorical variables. Continuous variables refer to the numeric values, which exist within a certain domain of numbers, for example: 10, 5.5, 12, 8.1 and so on. They can be described as a set of numbers between two given points: minimum and maximum values. The following graph (Exhibit 2.20) illustrates the continuous type of variable on the example from this study. The left-hand chart shows the distribution of 'Algorithm Variety Score' (evaluating the range of programming components used in each algorithm) for Design Patterns on the second day of the workshop. The vertical axis represents algorithm variety score and the horizontal axis represents participants. The right-hand chart shows the same data as a histogram, generated by SPSS (IBM, 2013) for normal distribution of data test. Note that, in this case, the vertical axis is frequency and the horizontal axis is algorithm variety score.

Exhibit 2.20. Example of Continuous variables. Algorithm Variety Score. Day 2. Design Patterns.

Continuous variables can have different numerical domains and different distributions, but they do have one thing in common; they are not limited by any categories.

Categorical data, on the other hand, refers to the data that can be sorted into categories, or can only take on one of a limited, often fixed, number of possible values. There are several types of categorical variables. They could be: ordinal, nominal and dichotomous (binary) (Feller, William, 1950). When data can take on exactly two values, for example in 'Yes'/'No' questions, it refers to a dichotomous or binary type of categorical variables. The difference between the ordinal and nominal variables is that for a nominal variable the order of the categories has no meaning. Colour categories such as 'Blue', 'Green' and 'Orange' can be an example of a nominal variable. The order for these categories has no meaning, as 'Blue' is not less than 'Orange' and 'Green'; or as 'Male' is not more than 'Female' and vice versa. An ordinal variable has a meaningful order, usually from smallest to largest, as, for example, in level of agreement: 'strongly disagree' is less than 'neutral' and 'neutral' is less than 'strongly agree.' An ordinal variable, where intervals between the values are equally spaced, is called an interval variable.

Two categorical data types: ordinal and binary data were collected from the online questionnaires and used in this comparative study. The data, collected from the 'Yes'/'No' questions, such as, of whether participants used any Design Patterns or Case-Based solutions in their designs, refers to binary, whereas the level of agreement or such scale questions as 'never', '1-3 times', '4–6 times' and so on, are ordinal.

Chart: types of data collected for evaluation criteria

DATA	Continuous variables
CRITERIA	Experience in architectural design and programming;
	Complexity of the output design model
	Novelty and variety of the programming algorithms (Explored Solution Space);
	Algorithm complexity scores;
DATA	Binary categorical variables
CRITERIA	Gender
	Type of programming difficulties ('Yes' or 'No' option for each type of programming difficulties)
	Type of design objectives (Yes' or 'No' option for each type of design objective)
	Flexibility of the approach: used or did not use Design Patterns/Case Based solution in their design
	Types of Key words ('Yes' or 'No' option for each category of key words)
DATA	Ordinal categorical variables* **these variables were treated as continuous variables in parametric statistical testing*
CRITERIA	Programming criteria (the scale of how often participants implemented a new components, reused algorithms or faced programming difficulties)
	Design ideation criteria (ability to model original idea and change in design intent scales)
	Approach characteristics criteria ('Usability', 'Utility', 'Intuitiveness' and 'Flexibility' scales)
	Motivation criteria (satisfaction and motivation scales)

Exhibit 2.21. Criteria sorted according to the data types: Continuous variables, Binary and Ordinal categorical variables.

Statistical analysis of data

The ordinal variables, such as level of agreement with the statement were treated as Likert scales (Lubke, Muthen, 2004). These scales have points that indicate the degree of agreement with a statement. In this study the scale went from: 1='Strongly Agree' to 5='Strongly Disagree'. When it comes to the analysis of these types of scales, the data, being in fact a set of ordered categories, can be considered and treated as continuous variables (Carifio, Perla, 2007). Treating the Likert scales as continuous variables gives an opportunity to use a greater variety of statistical tests. However, there is a split of opinions on this subject in the field of statistics. One group of scientists insists that the intervals between the scale values in the ordered categories are not absolutely equal. That is why the results of the parametric testing applied to the ordered variables cannot be considered valid (Jamieson, 2004). The other camp argues that, while Likert scales are technically ordered, in some situations the use of parametric tests is valid (Lubke, Muthen, 2004) and returns accurate values (Glass, Peckham, Sanders, 1972). This study has addressed this issue by applying both parametric and non-parametric tests on ordinal variables. In this comparative study a number of the ordinal categorical variables, such as Likert scales, obtained from the questionnaires were treated as continuous data. However all the results obtained by treating the ordinal variables as continuous, were validated by non-parametric testing.

When designing the questions for the ordinal data collection, the study considered the following principles. In order for ordinal variable to work properly in parametric statistical testing, the scale item should have at least five points; the concept underlying the measuring logic should be continuous, and the intervals between the points should be as equal as

possible. These conditions shaped the design of scale items used in the online questionnaires (Exhibit 2.22).

This example of the question (Exhibit 2.22), designed with Likert scale, illustrates the logic of collecting and interpreting data. The scale, visible to the participants, has five categories: Strongly Disagree, Disagree, Neither Agree nor Disagree, Agree, Strongly Agree. Only one category can be chosen. The underlined logic behind this scale item is that each category is awarded a numeric value (the 4th row, Exhibit 2.22). The scale goes in order from 1 = Strongly Disagree to 5 = Strongly Agree. As a result, the collected numeric data can be mathematically analysed. For example, it becomes possible to calculate a central tendency (a mean value) of the ordinal variable for each study group.

Example of the question designed with the Likert scale item

QUESTION	Please indicate the level of agreement with the following statement:				
	I was able to model my original design idea.				
ANSWER	Strongly Disagree	Disagree	Neither Agree nor Disagree	Agree	Strongly Agree
VALUE	1	2	3	4	5

Exhibit 2.22 Example of the Statement from the online questionnaire with the Likert scale, where the scale item has five points. The level of agreement goes from 1 = Strongly Disagree to 5 = Strongly Agree.

The following graph (Exhibit 2.23) illustrates the distribution of ordinal variables for the Design Ideation criterion: 'Ability to model original design idea'. The left-hand chart vertical axis represents the level of agreement from 1='Strongly Agree' to 5='Strongly Disagree'; and the

horizontal axis represents participants. Categorical data can often be interpreted as a proportion: right hand chart, where the vertical axis is the frequency of answers and the horizontal is the level of agreement. Proportion, being a number considered in comparative relation to a whole, can be calculated as a percentage for each category. For example the same criteria: 'Ability to accomplish original design idea' can be represented as: 'Strongly Disagree': 3%, 'Disagree': 37%, 'Neither Agree nor Disagree': 33%, 'Agree': 23%, And 'Strongly Agree': 3% (right-hand chart, Exhibit 2.23).

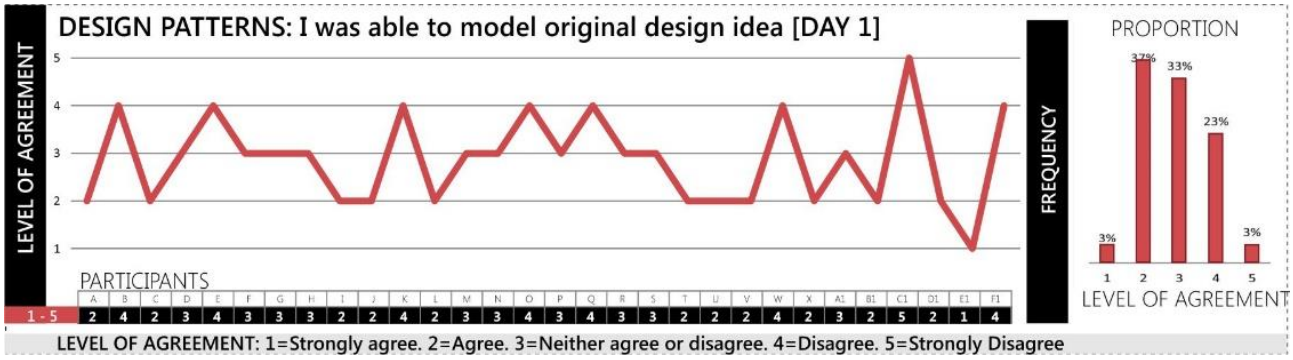


Exhibit 2.23. Example of Categorical variables. Ability to accomplish original design idea. Day 1. Design Patterns group.

Comparison of data

The purpose of the workshops was to collect the data and through the comparison of the data to ascertain whether there are differences that might be observed between results for the NA, DP and CBD test groups. The purpose of the statistical analyses was to determine whether the differences that were observed are statistically significant. The data obtained from the workshops indicated that there is indeed an identifiable difference in the results for every single criterion between the No Approach (NA), Design Patterns (DP) and Case-Based Design (CBD) groups. For some criteria the difference between the responses seemed to be substantial. In

other cases the results seemed very close – almost identical. In order to avoid ambiguity in the interpretation of the results, there needed to be a method to determine whether the differences in results did not occur by chance.

The role of the statistical testing can be illustrated using the example of comparison of the Design Ideation criteria: 'Ability to model original design idea' from the first day of the workshop, between the DP and CBD groups. The information in the chart is represented both as two separate column-charts for the Design Patterns approach and the Case-Based Design approach (left-hand diagrams) and as the overlapping line-chart for both approaches (right-hand diagram) (Exhibit 2.24). The vertical axis refers to the frequency and the horizontal refers to the level of agreement (five point scale from 1 = 'Strongly Agree' to 5= 'Strongly Disagree').

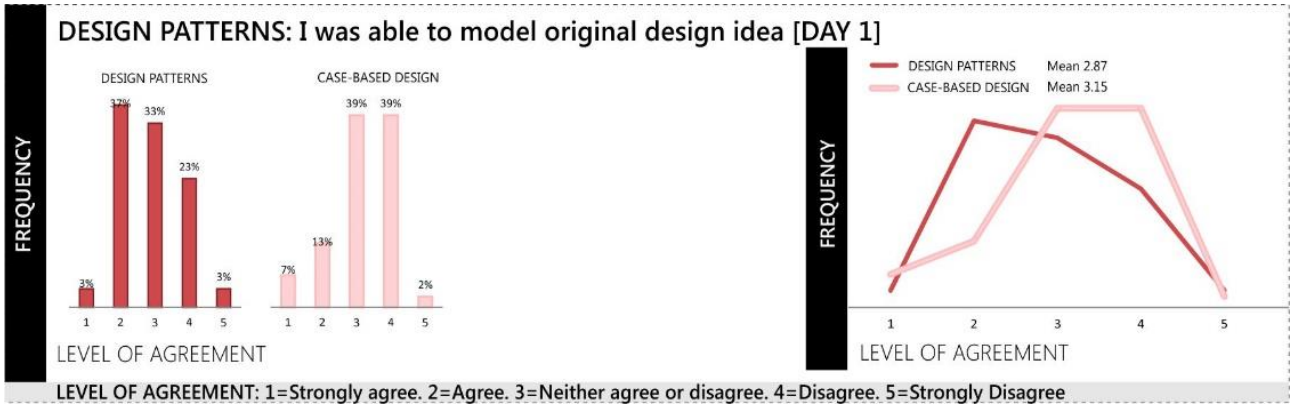


Exhibit 2.24. Comparison chart: Ability to accomplish original design idea. Day 1. Design Patterns. Case-Based Design groups.

The distribution of categories for the Design Patterns and the Case-Based Design approaches are seemingly different. As a result, the sensible conclusion might have been that the participants, who used the CBD approach, are more capable of modelling the original design idea. This conclusion, however, was not confirmed by both parametric and non-parametric statistical testing. Statistical tests and such statistical values, such as mean and level of significance, are often utilised in this comparative

study. That is why before going into the details of the interpretation of the tests interpretation it is necessary to explain some of the most relevant definitions and concepts.

Explanation of the statistical terms used in the comparative study

Mean – refers to a 'measure of central tendency of a distribution or the arithmetic average of a set of values' (Feller, Feller, William, 1950). In this study mean-values are used to compare the results between the three test groups: two test groups using the DP/CBD approaches and the group, which used no approach (NA). Means values help to determine which measure is greater and thus had a more positive or negative effect on a criterion. Although, the DP, CBD and NA mean-values can be (and almost always are) different, we cannot draw conclusions by reasoning that one value is greater/better than the other, unless we test if the difference between the means is statistically significant.

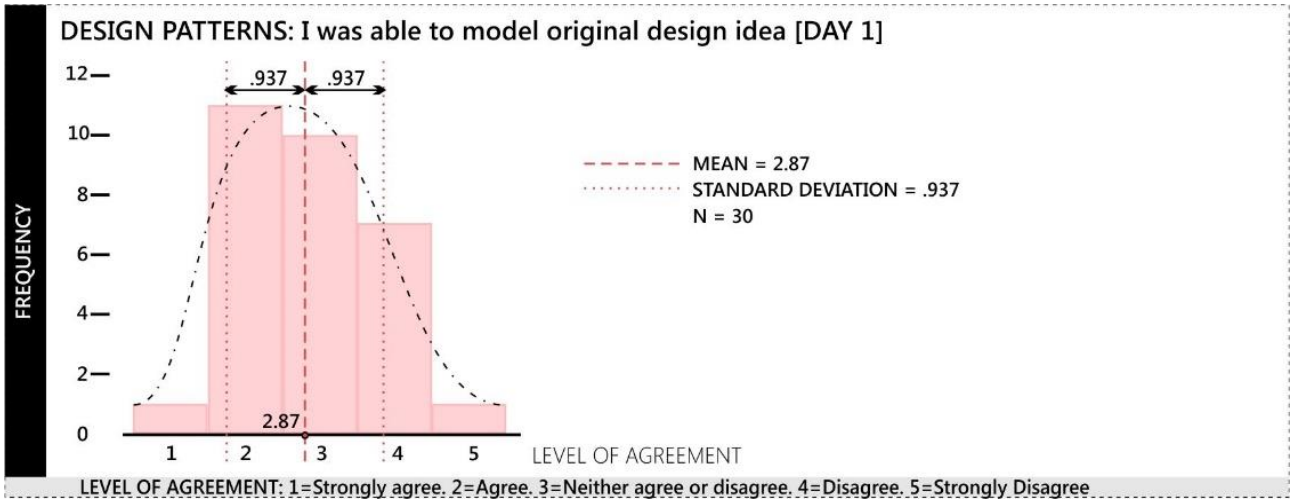


Exhibit 2.25. Mean and Standard Deviation of data. Criterion: Ability to accomplish original design idea. Day 1. Design Patterns.

Mean value example: The chart of Design Ideation criteria: 'Ability to accomplish original design idea' for the Design Patterns approach on day 1 (Exhibit 2.25.). The Mean for this distribution of ordinal variables equals 2.87 +- .937. Number 2.87 is an average answer and .937 is a standard deviation, which shows how much variation from the average exists (Saeed, 2000).

Hypothesis testing

To compare approaches the study uses statistical hypothesis tests. To illustrate, we can make a hypothesis that one of the approaches has a better effect on the architect's ability to model original design idea. This is our hypothesis and we are testing whether this hypothesis is true and, therefore, the ability to model original design concept of one of the groups is significantly better.

The null hypothesis states that the means of two samples are equal or not significantly different (Fadem, 2008). In our case the samples refer to the two approaches: Design Patterns and Case-Based Design. Unless rejected or disproved, the null hypothesis states that approaches have the same effect on the results. When rejecting the null hypothesis the analyst is able to state with a degree of certainty expressed as a probability that there is a significant difference between the mean scores for two groups. In this study, it becomes possible to determine whether there is a statistically significant difference between the performance of the DP and the performance of the CBD groups (and compare them to the control group (No Approach)).

P-value is a measure used to test the null hypothesis. When a p-value is below the statistical significance threshold, which is generally

accepted as 0.05 (or 95 % of confidence) (Zimmerman, 1997). (Stigler, 2008), then the null hypothesis is rejected in favour of the alternative hypothesis (Fadem, 2008). In other words:

If a p-value is above the 0.05 level, the null hypothesis is true. Therefore we can assume that there is no significant difference between the means;

If a p-value is below the 0.05 level, the null hypothesis is rejected Therefore we can assume, with 95% certainty, that there is a significant difference between the means.

It should be noted, that fundamentally the p-value is a measure of how likely the difference in the results could have occurred by chance. That is why, ultimately, the p-value alone does not justify the reasoning between the different hypotheses and should be combined with other types of evidence for and against the hypothesis (Hubbard, Lindsay, 2008).

Example:

The example of how seemingly different results for 'Ability to accomplish original design idea', between the approaches on day 1 (Exhibit 6.5), when tested statistically, did not prove to be significantly different.

Design Patterns group: Mean = 2.87 +- .937;

Case-Based Design group: Mean = 3.15 +- .932;

P-value = 0.200

The 0.200 is above 0.05 (level of confidence), which means that the null hypothesis is true, therefore, statistically speaking, there is no significant difference between the results, shown by participants who used the DP and CBD approaches. The confidence level of 0.200 indicates the

likelihood that the null hypothesis is true. It implies that the risk, that the difference in results has happened by chance, is 20%. Since the p-value gives us only 80% certainty, we cannot reject the null hypothesis. Thus we should assume that there is no significant difference in ability to accomplish original design idea between the DP and CBD groups.

Statistical tests used in the comparative study

Continuous variables, such outcomes as: Model and Algorithm Complexity Score, Novelty and Variety scores, as well as some the ordinal variables, were compared between approaches using the Independent samples T-test and Univariate Analysis Of Variance (ANOVA). The T-test was used to compare result between the two approach groups in cases where the data was relevant only to the DP and CBD groups, for example to compare how 'Intuitive' or 'Easy to use' each approach was. The No Approach (NA) group in this case had no such criteria. When comparing the results of all three test groups, for example 'Number of programming difficulties' of the Design Pattern group, Case-Based Design group, and No Approach group, the ANOVA testing was used. The ANOVA tests whether there is any difference in the means of all (more than two) groups and determines whether at least one mean is statistically different.

Comparison of the continuous variables/interpretation of the t-test

Independent samples T-Test is a null hypothesis test, designed to compare means of same variable between two groups. In this study the t-test was

used to examine whether the difference between the means of the DP and CBD approaches is statistically significant or if it is due to random chance.

The SPSS Independent samples t-test (SPSS) (IBM, 2013) provides a large number of various values as the output, including: mean, standard deviation, p- value, t-value, standard error difference and so on. Each of these outcome values has its own meaning and can be used for different aspects of the results interpretation. However, for this comparative study we need only means and p-values to interpret the results of the t-test (See Exhibit 2.26).

Colour-coding in the diagrams:

The following colour-coding is used for all the data-tables used throughout the comparison chapters, in other to make the reading of the results easier.

**green: the Mean value is greater, compared to the other group*

**light green: the Mean value is minor, compared to the other group*

**pink: the p- value indicates that there is a significant difference between the groups (for this particular criterion)*

**grey: the p- value indicates that there is NO significant difference between the groups (for this particular criterion)*

T-test example table. Comparison of approach 'Usability' between the DP and CBD groups

Criteria	DP (Mean)		CBD (Mean)		t		df		p - value	
USABILITY	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
It was easy to implement DP/CBD approach.	2.90 +- .885	3.03 +- .809	3.66 +- .668	3.77 +- .666	-4.280	-4.326	75	75	.000	.000

Exhibit 2.26. Independent samples T-Test example. Approach Usability

Day 1: t(75) = -4.280, p = 0.000; the t-value (t), the degrees of freedom (df) and the p-value

Day 2: t(75) = -4.326, p = 0.000;

Interpretation of the data in the t-test table (example):

The DP and CBD mean-values show the central tendencies for the DP and CBD groups (Exhibit 2.26). These values give us an opportunity to understand how each approach affects the criteria. For example the DP and CBD mean values (the 2nd and 4th column of the 3rd row of the table) indicate that (on first day of the workshop) when grading the agreement with the statement 'It was easy to use the approach in my design' on the scale from '1' – Strongly Disagree to '5' – Strongly Agree, the CBD group tend to agree with this statement more compared to the DP group (Exhibit 2.26). The average 'Usability' (easy to use) of the approach on day one is 2.90 +- .885* for the DP approach and 3.66 +- .668 for the CBD approach.

** Where 2.90 is mean and .885 is standard deviation (for the DP group);*

The p-value (the 10th column, Exhibit 2.26) is used to determine whether the difference between the two means (DP/CBD) on day one is significant. In this example the p-values on both days are 0.000. The 0.000

level is below the established level of significance 0.05, which is why we can reject the null hypothesis. Hence we can state that statistically, the Case-Based Design approach (as reported by participants) is significantly easier to use compared to the Design Pattern approach.

When reporting the statistical results of the t-test, the t-value (t), the degrees of freedom (df) and the p-value are stated. The following format can be used: **t(75) = -4.280, p = 0.000.** In this example, comparing the 'Usability' criterion of the DP and CBD approaches on day one, the t-statistics is -4.280 with 75 degrees of freedom and corresponding two-tailed p-value is 0.000.

Determining differences between three groups/interpretation of the ANOVA test

To determine whether there is any significant difference between the means of all three test groups: Design Pattern group, Case-Based Design group, and No Approach group, the One-way Analysis of Variance (ANOVA) was used. Unlike the t-test, ANOVA provides an opportunity to compare the means of several (more than two) groups for statistical significance. The analysis of variance is regarded as a 'robust procedure' when sample sizes are similar or equal (Wallenstein et al., 1980). In this study ANOVA was used to test the null hypothesis, stating that the effect of the DP, CBD approaches have no effect (the same effect) on the amount of programming barriers, which designers face, their algorithmic modelling performance, and other established criteria. Rejecting the null hypothesis would imply that the use of different approaches to reuse algorithmic solutions does have a significant effect.

In a similar manner to the t-test, ANOVA testing gives a range of values as output of calculations: such as p-values (Sig), F ratio, mean square, degrees of freedom, sum of squares and so on. Most of these values are not used in this study. The p-value is used to determine whether the difference in results (means) of the DP, CBD and NA groups has happened by chance or it is statistically significant. When a p-value is below the 0.05 level, the difference in results is determined as statistically significant. It should be noted that the ANOVA test only indicates whether there is a difference in the mean values. ANOVA does not actually tell which specific groups were significantly different from each other. That is why to determine which specific mean is different from which, one needs to use a Post Hoc tests (SPSS, 2014). This study used the Post Hoc Tukey's test to compare each pair of groups. The Tukey's test is recommended for estimation of pairwise differences and regarded as an 'exact and optimal' test for comparisons (Stoline, 1981). This method is also considered to be easy to use and robust (Ibid), giving 'reasonably accurate results' when the sample sizes are similar (Wallenstein et al., 1980). The p-values are used to interpret the results of the Post Hoc Tukey's test: above 0.05 level – not significantly different, p-value below 0.05 level – significantly different.

Interpretation of the data in the ANOVA/Post Hoc Test table (example):

The NA, DP and CBD mean-values of the output 'Model Complexity Score' are shown on the first table (the 3rd row, the 2nd – 7th columns) (Exhibit 2.27). The p-values (10th and 11th column of the 3rd row) indicate the significance of difference in means (Exhibit 2.27). On day one the p-value equals 0.560, which is above the 0.05 level. Therefore the null hypothesis is true: the means are not significantly different. No additional testing is needed. On day two the p-value is 0.031, which is below the 0.05, that is why we can state that on the second day the

average 'Model Complexity' is significantly different (between at least two of three test groups). Using only ANOVA, makes it impossible to determine which group is different from which and additional testing (Post Hoc) is needed.

Example of ANOVA. Comparison between the 'Model Complexity' of the No Approach group, the DP group and the CBD group

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Model Complexity Score	11.73 +- 2.465	13.94 +- 2.585	12.23 +- 2.046	14.10 +- 2.551	12.15 +- 2.246	12.74 +- 2.246	.583 (126)	3.569 (126)	.560	.031

ANOVA/Post Hoc, Tukey's test

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		p – value NA with DP		p – value NA with CBD		p – value DP with CBD	
		DAY 2		DAY 2		DAY 2		DAY 2		DAY 2		DAY 2
Model Complexity Score		13.94 +- 2.585		14.10 +- 2.551		12.74 +- 2.246		.960		.062		.065

Exhibit 2.27. ANOVA test example with Post Hoc Tukey's Test. Model Complexity

The second table (Exhibit 2.27) shows the results of the Post Hoc Test (Tukey's test). The p-values are used to determine which specific groups are different from each other. The comparison between No Approach and Design Patterns groups indicates no significant difference, the p-value is 0.960 (above the significance threshold) (Exhibit 2.27, Second table the 9th column of the 3rd row). The comparison between the No Approach group and Case-Based Design group shows that even though the p-value (0.062) is above the 0.05 level it is very close to it (Exhibit 2.27, Second table the 11th column of the 3rd row). This means that there is 93.8% certainty that the CBD approach had a significant effect on model complexity. The comparison between the DP and CBD groups

gives the p-value of 0.065 (Exhibit 2.27, Second table the 13th column of the 3rd row). Again though technically this p-value is above the level of significance (0.05), the results still might be interpreted as significantly different. The final conclusions could be based on additional data, such as the fact that the models produces by the DP group have a more advanced colouring and a larger range of elements compared to CBD group.

To report the statistical results of the ANOVA, the F ratio (F), the degrees of freedom (df) and the p-value are used. The following format can be utilised (Exhibit 2.27)

Day 1: F (126) = .583, p = 0.560;
Day 2: F (126) = 3.569, p = 0.031;

The results acquired from the t-testing and ANOVA, comparing the ordinal variables, were confirmed by the non-parametric Mann-Whitney test, also known as Wilcoxon test (See Appendix).

Testing for the gender and design experience influence

After determining that a criterion differs by approach, as indicated by the t-test results or ANOVA, one cannot automatically assume that it was the approach factor that made all the difference. There could be other factors that might have influenced the results. Two main control variables or covariates were identified for this comparative study: design experience and gender. In theory, both those covariates might have influenced participants' performance. Experience with programming modelling tools could have been a strong factor as well (and potentially a third covariate), but it was not applicable, as the dominant part of workshops' participants (>95%) had no programming experience.

Covariate or control variable - is a secondary variable that can affect the relationship between the criteria variables and the approach.

Covariate example: Control Variables identified for this study are gender and design experience.

Univariate Analysis Of Variance (ANOVA) was used to control for the covariates and to determine whether the design experience or gender variables have a significant effect on the criteria - dependent variables. Design experience and gender in this case are the independent variables. The No Approach, Design Patterns and Case-Based Design groups were used as a fixed factor.

The ANOVA test with dependent variables (SPSS) (IBM, 2013) helps to determine whether the changes in the independent variables (experience/gender) have a significant effect on the dependent variables (criteria). The only ANOVA output values that are utilised and interpreted, when testing for gender and design experience influence (control variables), are the p-values. Again, a p- value is used as a measure to determine, whether there the control variables have a significant effect on the results.

ANOVA for testing for covariates (example table):

Dependent Variable	Approach/p-value		Approach/F (df)		Design Experience/p-value		Design Experience/F (df)	
How often you have come across program. difficulties	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach/Design Experience		.180		(1,67) 1.836		.536		(4,67) .790
	Approach/p-value		Approach/F (df)		Gender/p-value		Gender/F (df)	
Approach/Gender		.014		(1,73) 6.351		.880		(1,73) .469

Exhibit 2.28. Univariate Analysis Of Variance example. Criterion: 'Number of programming difficulties' (second day of the workshop); Fixed factor DP and CBD approach. Control variables: Design experience and Gender.

Interpretation of the data in the ANOVA test for covariates table (example):

In order to check that it was indeed the approach that affected the dependent variable the following two conditions has to be true:

First Condition:

When testing for design experience and gender, the approach variable makes a difference to the dependent variable. The approach p-values should be below the 0.05 level (See Exhibit 2.28). In the example ANOVA testing for covariates table, the approach p-values are in the third column. This table shows the data for 'programming difficulties' criterion. This statistical testing helps to determine whether the number of programming difficulties was influenced by the approach and not by the gender and design experience factors. The p-value for approach/gender analysis is 0.014 (See Exhibit 2.28, 3rd column, 5th row), which is below the 0.05 significance level. That means that when testing for gender influence, the approach makes a difference to the number of programming difficulties. The p-value for approach/design experience is 0.180, which is above the 0.05 level (See Exhibit 2.28, 3rd column, 3rd row). That means that the first conditions is not true and this issue might need additional investigation. In this particular case, the results might suggest that the difference in the number of programming difficulties between the approaches was influenced by the 'design experience' factor. It should be noted that in terms of design experience, all study groups had very similar distribution.

Second Condition:

When testing for approach, design experience and gender variables (control variables) do not make a difference to criteria variable. In this case the p-values of control variables should be above the 0.05 level. In other words design experience and gender does not affect the

results. In the example table both p-values are above the level of significance: design experience/approach p-value is 0.536 and gender/approach p-value is 0.880, hence the condition is true. (See Exhibit 2.28, 7th column, 3rd and 5th rows)

When one or both of two conditions are not complied, it is necessary to carry on an additional investigation, and look at the descriptive statistics in order to clarify the results. This should be done individually for each criterion.

Comparison of categorical variables/interpretation of the chi-square test

Categorical Variables, such as proportions of types of programming difficulties, design objectives, and key words, were compared between approaches using the Chi-square test of Significance (X^2) (SPSS/Cross Tabs) (IBM, 2013). It is used to test for significance in relationship between categorical variables. The Chi-square test works only with bivariate data tables, such as: Yes/No, Pass/Fail, Male/Female, which can be mathematically represented for example as 1/0. Unlike the t-test and ANOVA, the Chi-square test compares counts, not means. That is why it is not applicable for comparing continuous data, such as model or algorithm complexity score. However, there is a number of similarities between the t-test, ANOVA and the Chi-square test. Similar to these two tests, the Chi-Square Test of Significance is a hypothesis test. The null hypothesis, which is being tested, states that there is no relationship between the variables in the bivariate table. In our case, the null hypothesis states that, any difference between the distribution of categorical data in

the No Approach, Design Patterns and the Case-Based Design groups is due to chance.

Chi-Square Test (example table):

Criteria	No Approach Count/Total (%)		DP Count/Total (%)		CBD Count/Total (%)		X^2		p – value	
DESIGN OBJECTIVES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
To experiment with parameters	**8%** (2/25)	**12%** (3/25)	**20%** (6/30)	**46.7%** (14/30)	**19.1%** (9/47)	**8.5%** (4/47)	1.801	17.800	**.406**	**.000**

Exhibit 2.29. Chi-Square Test example. Criterion: 'Design Objectives', category 'to experiment with parameters';

Similar to the t-test and Univariate Analysis Of Variance, the Chi-square test of Significance uses the p-value as a measure to test the null hypothesis. The p-value, indicates how likely it is that the differences in distribution (count) of the NA, DP and CBD variables is due to random sampling error. The predetermined significance level for the p-value was set as 0.05* (Zimmerman, 1997) (See Hypothesis Testing section). *The p-values located between 0.05 and 0.07 are considered – a strong trend. Similar to the ANOVA (testing the NA, DP and CBD groups), the first Chi-square test is made between all the groups. In cases when the p-values of this initial (three groups) comparison is significant (below the 0.05) additional testing was carried on. The multiple Chi-square test between each pair of groups was performed to determine which group differs from which.

Interpretation of the data in the Chi-Square Test table (example):

This particular Chi-Square test example illustrates the comparison of the 'Design Objectives' criterion, namely, how many participants wanted 'to experiment with parameters'. The test helps to determine whether there is a significant difference in results between the NA, DP and CBD groups.

The percentage and counts of participants who had: 'to experiment with parameters' as one of their objectives are located in the 3rd column of the 2nd-3rd row for the No Approach group, in the 4th-5th column of the 3rd row for the Design Patterns group, and the 6th-7th column for the Case-Based Design group (See Exhibit 2.29). The values are written in % and the 'Yes/Total' format, where the first number in brackets refers to the count of people who had this objective (Yes) and the second number refers to the total number of participants in this group (Total). For example, in the DP group 14 out of 30 participants wanted 'to experiment with parameters, while there were only 4 out of 47 participants in the CBD group with the same design objective (See Exhibit 2.29). These results also shown in percentage, as the number of participants in each groups is not equal. The 8th-9thth column of the 3rd row shows the Chi-Square – value and the 10th-11th column of the 3rd row shows the p-value, which, in this case, equals 0.406 on day 1 and 0.000 on day two (See Exhibit 2.29).

The closer a p-value to zero the more significant is the difference between the results. As on the second day the p value equals 0.000 ('Design Objectives' example, Exhibit 2.29) one can state that the number of participants, who indicated 'experimentation with parameters' as their design objective, is significantly bigger in at least one of the groups compared to others: NA – 12%, DP – 46.7%, CBD – 8.5%.

To report the statistical results of the Chi-Square test, the count of responses, the percentage, the Chi-Square – value (X^2) and the p-value are used. The following format can be utilised: NA 3/25 (12%), DP 14/30 (46.7%), CBD 4/47 (8.5%), $X^2 = 17.800$, $p = 0.000$ (values are taken from the Chi-Square test example table).

Dependence between criteria

In statistics, correlation refers to any statistical dependence between variables (Dowdy, Wearden, 1983). For example, this study has identified statistical dependence between the 'number of programming difficulties' that designers have and their 'ability to realise original design idea'. The more problems participants had with programming the less it was likely that they will be able to model their original design idea.

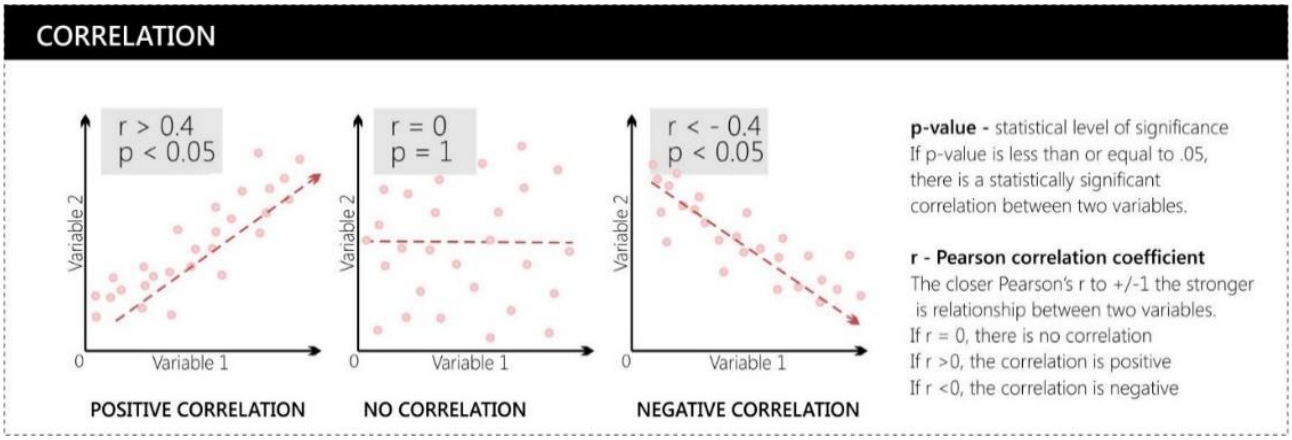


Exhibit 2.30. Correlation Diagrams. Positive Correlation, No Correlation, Negative Correlation

The most common measure of correlation in statistics, which shows the linear relationship between two variables, is the Pearson correlation (Buda, Jarynowski, 2010). Pearson correlation test (SPSS) (IBM, 2013) is used to determine the degree of linear dependence between algorithmic modelling criteria, programming criteria, motivation criteria and etc. The test gives the correlation coefficient value (r-value) between +1 and – 1 and the p-value (See Hypothesis Testing Section). The p-value indicates the probability that the correlation has occurred by chance. The smaller the p-value the more significant is the dependence between the variables. When the p-value is below the 0.05 level, one can assume (with 95% confidence) that the correlation did not happen by chance. The Pearson correlation coefficient (r-value) indicates the strength and (negative or

positive) direction of the correlation. When variables are absolutely independent the correlation coefficient equals 0; and that means that there is no correlation. The closer the correlation coefficient to +1 the stronger is positive correlation, the closer it to -1, the stronger is the negative correlation (See Exhibit 2.30).

Pearson correlation (example):

The 'Algorithm Complexity' has a strong positive correlation with the 'Algorithm Variety' (variety of programming components) criterion for all the test groups. For example, on the second day of the workshops the Pearson correlation coefficient calculated for these two criteria equals 0.599 and the p-value is 0.000 (See Exhibit 2.31). This means that there is a very strong positive dependence between the variety of components, which participants implement, and the level of complexity of the resulting programming algorithm.

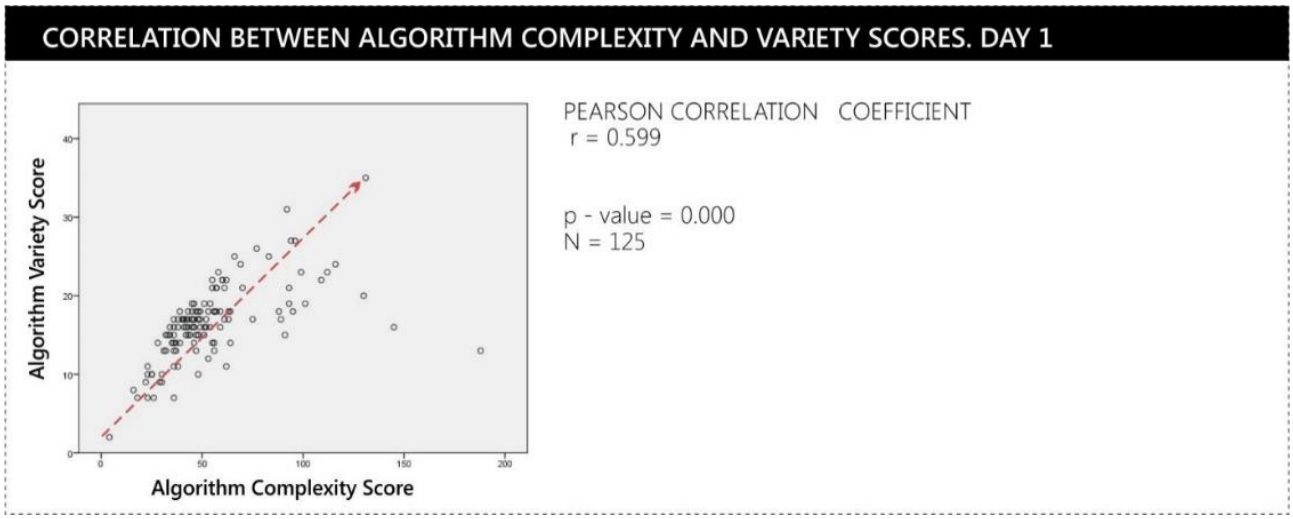


Exhibit 2.31. Correlation between the Algorithm Complexity Score and the Algorithm Variety Score, day 1

The following chart (Exhibit 2.32) shows what tests were used to compare different criteria.

TEST	ANOVA (comparing three test groups: No Approach group, Design Patterns group, CBD group) Post Hoc (Tukey's) Test was used to compare between each group
CRITERIA	Model Complexity score, Algorithm complexity scores;
	Novelty and Variety of programming algorithms (Explored Solution Space);
	Programming criteria (number of programming difficulties, implemented a new components)
	Design ideation criteria (ability to model original idea and change in design intent scales)
	Motivation criteria (satisfaction and motivation scales)
TEST	T-test (comparing two approach groups: Design Patterns and Case-Based Design)
CRITERIA	Approach characteristics criteria ('Usability', 'Utility', 'Intuitiveness' and 'Flexibility' scales)
TEST	Chi Squire Test *comparing binary data*
CRITERIA	Flexibility of the approach: used or did not use Design Patterns/Case Based solution in their design
	Type of programming difficulties ('Yes' or 'No' option for each type of programming difficulties)
	Type of design objectives (Yes' or 'No' option for each type of design objective)
	Types of Key words ('Yes' or 'No' option for each category of key words)
TEST	ANOVA test for covariates
COVARIATES	Experience in architectural design;
	Gender;

Exhibit 2.32 Chart: Criteria and Statistical Tests Used For Comparison

(Also refer Appendix B, pages, B55, B64)

This research was designed as an experimental comparative study. The results of three test groups, including the control group using no approach, the group using Design Patterns, the group using Case-Based were compared using relevant statistical tests and analyses. The objective was to measure and compare the effect of the knowledge reuse approaches,

1) by testing the null hypothesis, stating that there is no statistically significant difference between the results of the test groups. If the testing rejected the null hypothesis that indicated that the difference in results is statistically significant – which was used as empirical evidence for answering the research question); and

2) by investigating the dependency (correlations) between the measured criteria, this thesis aimed to attain a better insight of how participants' performance is related to their experience with the respective approaches. This correlational analysis helped to interpret the results, by suggesting why or how the reuse of abstract and case-based solutions effects the design process and design outcomes.

A total of 126 designers participated in the study. These numbers provided sufficient numbers within each test group to permit rigorous studies of the statistical significance of the observed differences.

2.4 Design Outcomes

Exhibits 2.32 and 2.33 illustrate the designs that participants of all the test groups produced during the first and the second days of the workshops. More detailed images of these design works can be found in the Appendix B (pages B56-B63).

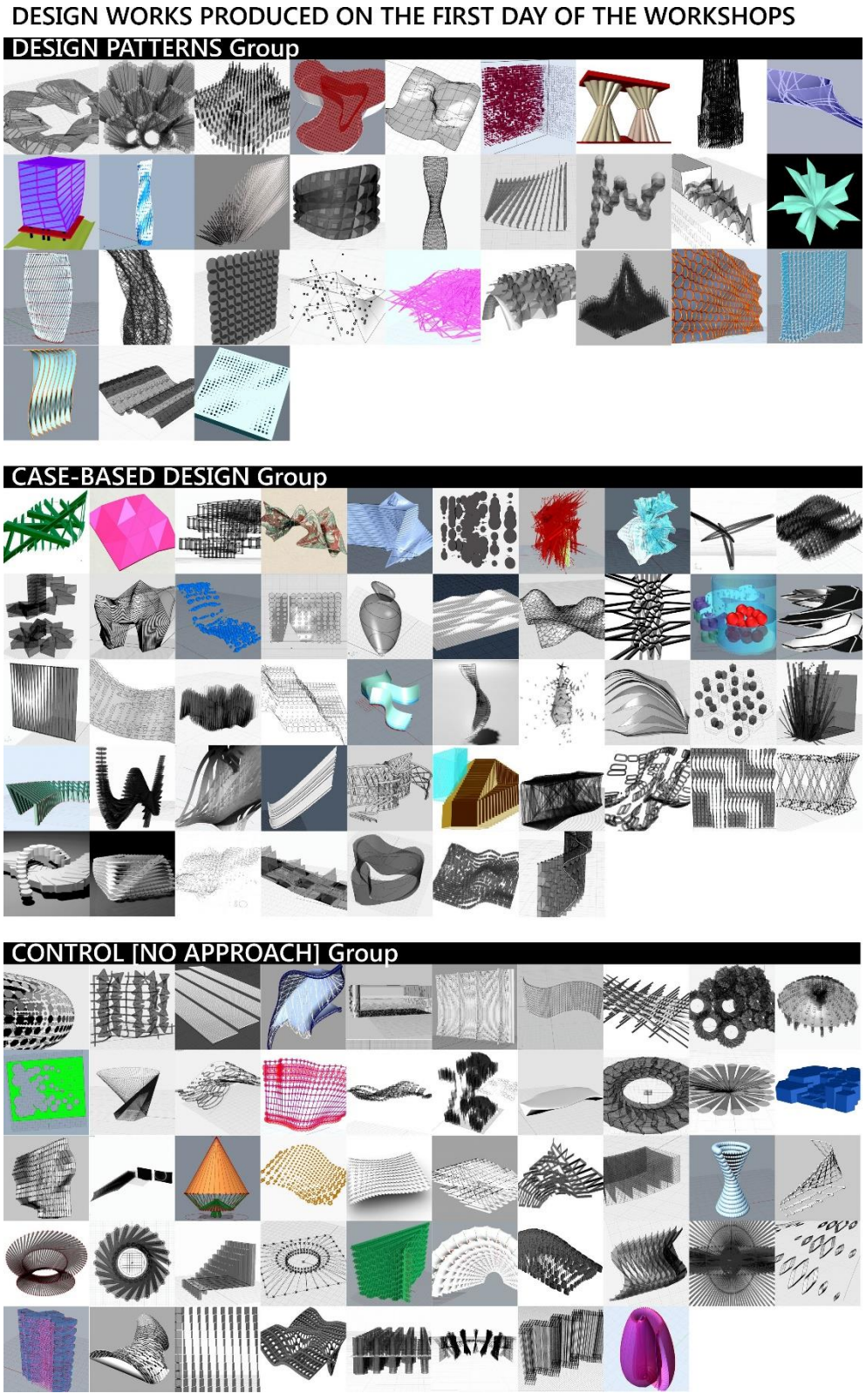


Exhibit 2.32. Design works produced by the participants of the DP, CBD and NA groups on the first day of the workshops

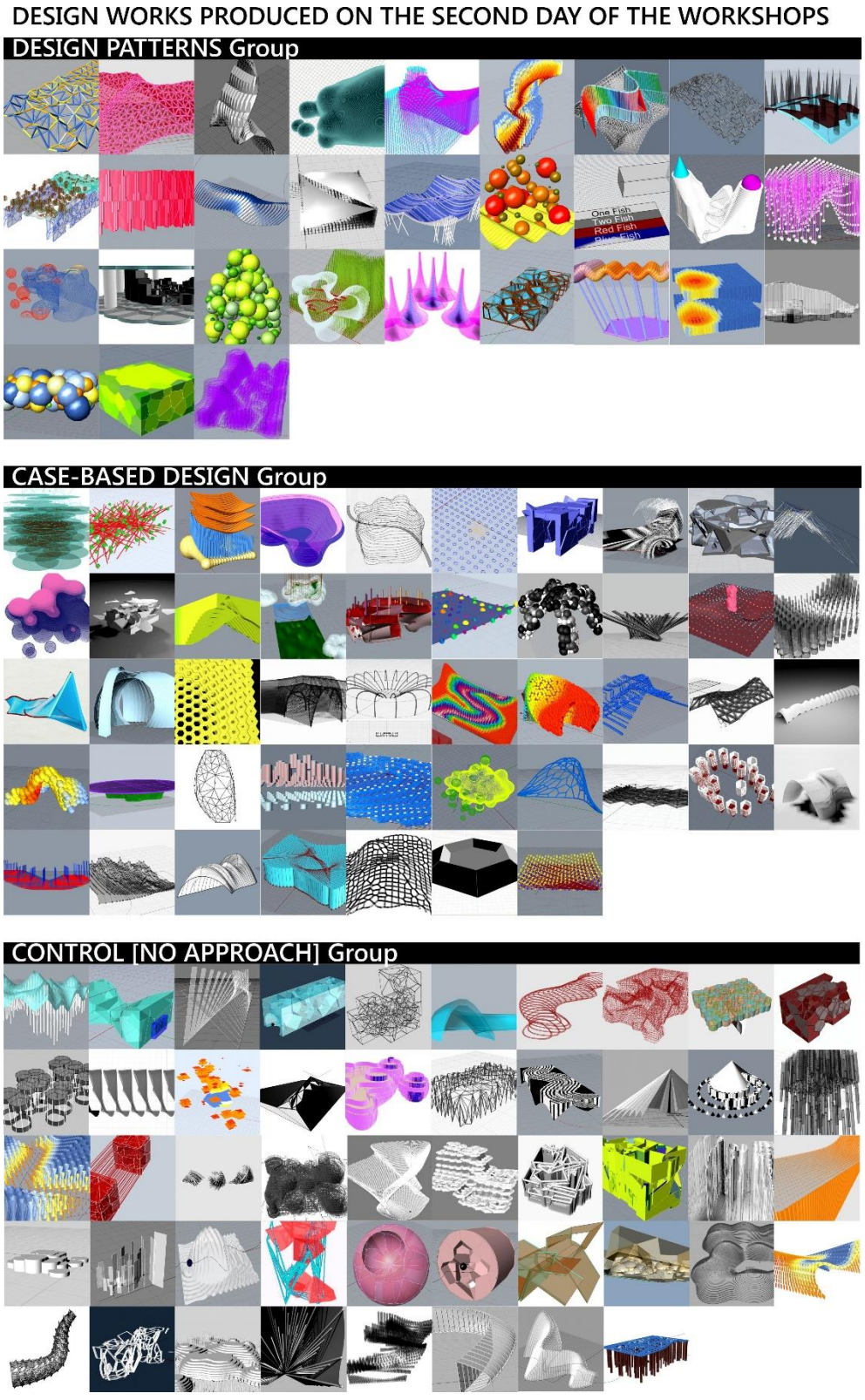


Exhibit 2.33. Design works produced by the participants of the DP, CBD and NA groups on the second day of the workshops

3. Results

The Results chapter reports the results of statistical tests and analyses and relates them back to the discussions raised in the Background and Methodology chapters (sections 1.1 – 2.3). How (in theory) each reuse approach was expected to affect designers' performance versus the experimentally obtained results measured by this study (how it affected designers' performance in practice). The Results chapter is split into four main sections. The first section presents the overall results and reflects back to the discussion on the opportunities and challenges of algorithmic design. The first section of this chapter discusses participants' feedback regarding the use of algorithmic modelling tools. It also presents the most common types of programming barriers, which designers and architects faced when using algorithmic modelling in their designs. The thesis compares these barriers with the typology of programming barriers discussed in literature in the context of software design. This comparison indicates that, when using programming languages, designers and architects face similar barriers to software designers. The first section also presents the key differences between the control group (using no approach) and the test groups that integrated the knowledge reuse approaches into the design process.

The second section of the Results chapter presents the comparison between the results of the control group and the test group that used

Design Patterns (abstract solutions). It refers the measured and compared results back to the hypotheses drawn from the literature. What the integration of Design Patterns was expected to do, versus what was observed and measured.

The third section of this chapter discusses the effect of the reuse of case-based algorithmic solutions, comparing the results of the CBD (Case-Based Design) group with the results of the control group. This section also expands on the expected effect (informed by the literature studies) versus the measured effect.

The final fourth section of the Results chapter compares the performance of the participants using Design Patterns and participants reusing algorithmic solutions from the Case-Base. It discusses how the reuse of abstractions affects designers and contrasts these findings with the Case-Based Design approach. This end of this section presents the summary of key findings.

3.1 Outline of the overall results

The use of algorithmic modelling tools in architecture and design

Results of this experimental study suggest that despite the barriers that programming imposes on architects, the use of algorithmic modelling tools can provide a means for dynamic form-finding and design exploration during conceptual design stages. Architects and designers, who participated in the study and used programming as a drafting method for development of their conceptual models, reported that they were able

to accomplish what they wanted and were satisfied with the design outcome.

On a five point agreement scale from 1- 'Strongly Disagree' to 5 – 'Strongly agree': (all test groups)

'*I was able to accomplish all what I wanted*' (mean, std. deviation)

Day 1: **3.37** ± 0.855 (median 4 – '***Agree***'),

Day 2: **3.53** ± 0.855, (median 4 – '***Agree***');

'*I am satisfied with what I was able to accomplish*' (mean, std. deviation)

Day 1: **3.63** ± 0.909, (median 4 – '***Agree***'),

Day 2: **3.83** ± 0.830, (median 4 – '***Agree***')).

It is often argued that the use of computer-aided design tools is not particularly effective during the conceptual form-finding stages of design (Dorta, 2007) (Cao, Protzen, 1999) (Pérez, Dorta, 2011). One of the arguments is that in many cases the form of a design concept is not properly defined, while the form of a computer model has to be specific/defined in the digital space (Ibid). Hand sketches on the other hand can be rather vague and abstract, leaving a room for interpretations, which allows architects to gradually reveal/develop the future form of their design solution. In many ways the objective of the conceptual design stage is not only about finding the right design solution, but rather figuring out what is the right question/design problem. Design ideation is not a straightforward process of logical reasoning and heavily relies on intuition (Shih, Williams, Gu, 2011). A further argument is that because human and computer logics do not always follow the same patterns the use of digital tools can limit

and even suppress a designer's ability (Ibid). Some of the recent research findings suggest that CAD tools are unable to fully support the ideation process during conceptual design stages and that computer technology fails to compete with hand sketching and modelling (Dorta, 2007).

The validity of the arguments regarding the issues and modelling limitations of the algorithmic CAD environments can be supported by the results of this study. However the feedback from designers, who used algorithmic modelling for their conceptual designs, also suggests that the advantages, which algorithmic form-making systems offer to CAD users, can outweigh the limitations and disadvantages. Unlike hand sketching and manual CAD drafting, algorithmic modelling gives designers an opportunity to generate numerous variations of the output forms. The development of design artefacts (solutions) often requires designers and architects to explore multiple alternatives. Algorithmic (generative) design enables users to generate thousands of design possibilities (Krish, 2011). This enables designers to instantly see and evaluate all changes in the form of their output models, and make alterations by changing parameters and logic in the form-making algorithm. Unlike a sketch, an algorithmic design model isn't a fixed visualisation of a concept, but it is rather a fluid and dynamic system. One programming algorithm can generate as many configurations and iterations as necessary (Exhibit 3.1). A designer has an opportunity to understand and evaluate different form versions. Thus an algorithmic model has an advantage: representing not one design option, but a range of design options.

This type of design process can be described as an exploration of 'if–then' constructs, when a designer experiments with the forms and processes to see how each model variation is going to look (Exhibit 3.1. Model variations generated by the same programming algorithm/different

parameters). It was observed that some workshop participants got results that they did not entirely anticipate (such as form instances generated by their programming solutions, which they did not foresee). On the one hand this unpredictable outcome could be seen as a disadvantage (as something that was not intended), on the other hand these unpredicted (experimentally obtained) results could potentially lead to new discoveries and further progress of the design concept. That is why it has been argued that the use of algorithmic systems can enrich and improve design innovation, contributing to the 'pro' arguments in the debate (controversy in opinions) regarding the relationship between CAD and creativity (Chen, 2007), (Benton, 2007) (Zeid, 2005).

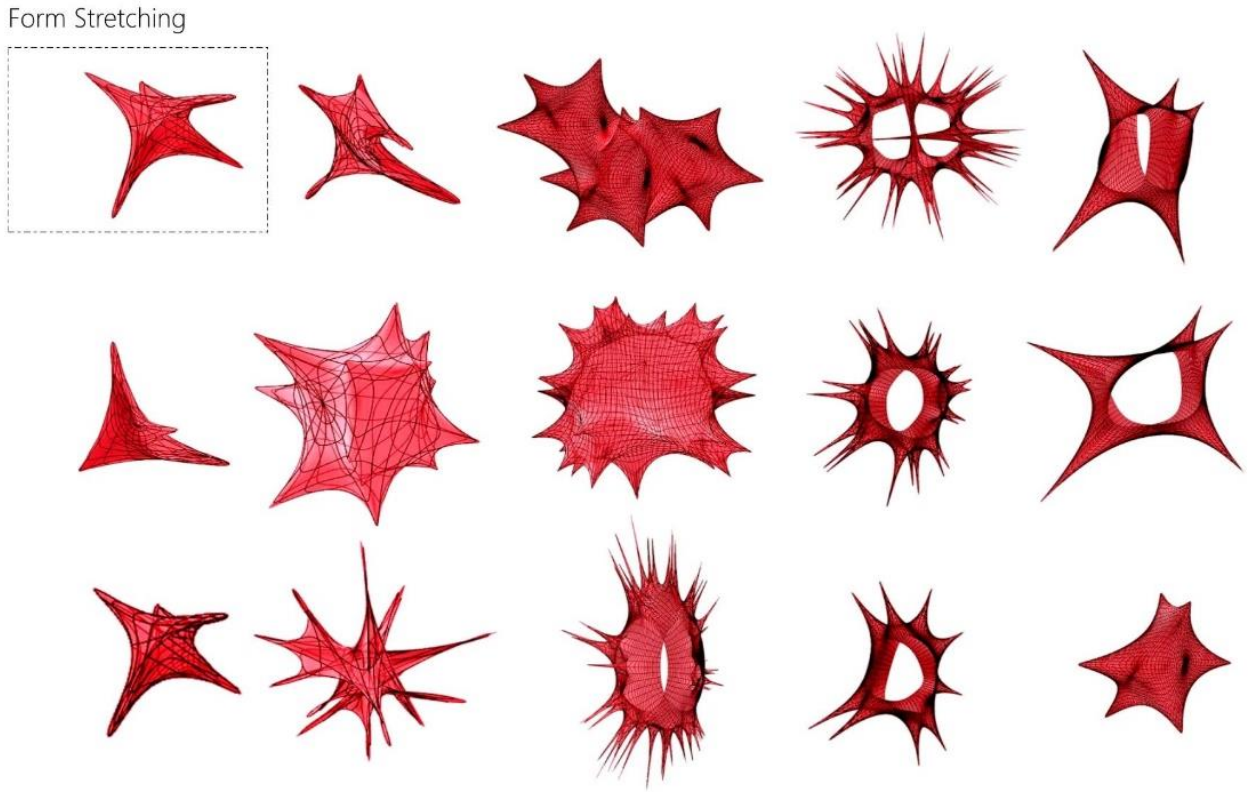


Exhibit 3.1. Algorithmic form finding 'Stretching'. Output model variations.

The algorithmic methods of open-ended form exploration diverge from conventional progressive form making. Design through programming offers a dynamic way to probe conceptual designs. In this respect algorithmic design exploration provides a unique opportunity, which is missing from manual CAD modelling and hand sketching. This dynamic form exploration works because the end form of their conceptual designs was still abstract (not clearly identified or fixed). Therefore, despite an opinion suggesting that Computer-Aided Design is only appropriate for the post-development stages and that its value as a design development tool is very limited (Charlesworth, 2007), it can be argued that parametric CAD systems can be extremely effective and useful during initial design stages. However, the use of parametric modelling systems also challenges designers, because even on early design stages it requires a systematic abstract thinking. That is why it is crucial to support the reusability of knowledge during this parameterisation process (Turrin, von Buelow, Stouffs, 2011).

Barriers associated with the use of algorithmic tools in architecture

Although participants (in all three test groups, including the control group, Design Patterns group and the Case-Based Design group) gave positive feedback regarding their experience with use of algorithmic modelling environments (ability to accomplish what was wanted/satisfaction with output), designers also reported that the use of programming caused substantial difficulties, which in some cases they failed to overcome on their own.

It is acknowledged that the use of programming tools in design can cause substantial, often insurmountable, barriers in end-users, especially for novices (Ko, Myers, Aung, 2004). Some argue that while programming systems offer effective and powerful means for modelling, many architects and designers struggle to adopt their logic and syntax, because of the mismatch in the initial principles of human and computer reasoning (Celani and Vaz, 2012). This study confirms that many novel users find the programing framework and syntax rules highly frustrating and not intuitive (as was often pointed out in previous research in this field) (Ibid), (Woodbury, 2010). In all the test groups, designers reported difficulties when integrating algorithmic thinking into their design process. For example, on average on the first workshop day designers had from 4 to 6 substantial difficulties, which they were not able to overcome on their own. On the second day the average dropped to 1-3 difficulties.

On a five point scale, with 1- 'Never', 2 - '1-3 times', 3 – '4-6 times', 4 - '4-6 times', 5 – '10 times or more';

'How often have you come across insurmountable programming difficulties, while developing your design model', (mean, std. deviation) (all test groups)

Day 1: 2.77 ± 0.989 (median 3 – '*4-6 times*'),

Day 2: 2.50 ± 0.787 (median 2 – '*1-3' times*').

Parallel to examining the overall number of programming difficulties, this study also investigated the nature (typology) of barriers associated with the use of algorithmic modelling systems. Previous

research on learning barriers in programming systems carried out by Ko et al. identified six types of most re-occurring types of barriers: design, selection, coordination, use, understanding, and information (Ko, Myers and Aung, 2004) (See Context of the Study section). The participant designers of this study were asked to indicate the overall amount of difficulties that they had while developing their design assignments and also to specify what type of difficulty it was. The analysis of responses was carried out independently of previous research findings (existing typologies). The aim was to identify the original groups of programming barriers, and afterwards compare them to the typology discussed by Ko et al (2004). The responses were collected as an open-ended type of enquiry, where designers expressed and individually articulated their own understanding of the nature (description) of the difficulty encountered. These responses were analysed and sorted into the five most re-occurring categories: idea-to-algorithm translation, problems with implementation of particular components, knowing what programming component to use, logic connections, and valid parameters. For example, the identified category 'Idea to Algorithm Translation' refers to cases when participants expressed the barriers as: not knowing how to get from a sketched idea to an algorithm of actions (generating this form). Participants expressed it in a variety of ways:

- *'Not quite knowing how to create what I want';*
- *'I just can't get it to do what I want it to. My logic is not attuned to that of the machine';*
- *'Struggle to achieve the form I wanted'*

Results show that the five barrier groups identified by this study (tested for visual programming using Grasshopper/Rhino) closely correspond to the typology identified by Ko et al. (2004), who investigated

learning barriers in programming systems on 40 participants learning programming with Visual Basic. NET (VB).

The five most common categories of programming barriers (difficulties) identified by this study are explained and referred to corresponding categories proposed by Ko et al. (2004):

1. Idea-to-algorithm translation (Figuring out how to get from a sketched idea to a programming algorithm, which generates a model). 61 out of 126 participants, who used algorithmic modelling for their conceptual designs had this type of difficulty on day 1, 64 out of 126 on day 2. This category corresponds to the *design barriers* (cognitive difficulties): 'I do not know what I want the computer to do' (Ko, Myers and Aung, 2004).

2. Syntax Problems/Problems with implementation of particular components (when participants knew which programming component they need, but struggled with how exactly to use/implement it. In scripting it can also refer to the syntax or 'grammar' errors, for example opening brackets without closing them). 42 out of 126 participants had this type of difficulty on day 1, 48 out of 126 on day 2. This type of difficulties corresponds to *use barriers*, 'I think I know what to use, but I do not know how to use it' (Ibid).

3. Knowing what programming component to use. 41 out of 126 respondents reported that the barrier was 'not knowing what to use' on day 1 and 34 out of 126 on day 2. This category matches the *selection barriers,* described as: 'I think I know what I want the computer to do, but I do not know what to use' (Ibid).

4. Logic Connections (what is the correct sequence of programming logic, for example should 'vector' go before or after 'move',

or how to organise a correct sequence of programming components to incrementally rotate multiple elements). 'Logic Connections' can also be described as problems with syntax: structuring of statements in programming algorithm. On the first day of the workshop 30 out of 126 designers reported problems with 'Logic Connections', on the second day it was 28 out of 126. This category accommodates two corresponding types of programming barriers identified by Ko et al. (2004): the coordination barriers, described as: 'I know what to use, but I do not know how to make them work together'; and use barriers: 'I think I know what to use, but I do not know how to use it' (Ibid).

5. Valid Parameters and Unexpected Errors were grouped as the last category of programming barriers identified for this study (these could be, for example, the *functional errors*, when an action/programming component is given an incorrect input information, such as improper domains of numbers, or the path to a source file, which doesn't exist. On the first day 18 out of 126 participants encountered problems with figuring out valid parameters/getting 'red boxes' and error messages, on the second day 16 out of 126. This fifth category is very close to Ko et al.'s understanding barriers type, occurring mainly due to the mismatch between the designers' expectations and program's actual behaviour: 'I thought I knew how to use it, but it did not do what I expected' (Ibid).

Thus the most common type of barrier identified by this study for novice users of algorithmic modelling tools was: 'Idea-to-Algorithm Translation'. This type of programming barrier was reported by half of the workshop participants. Even on the second day of the workshops, when participants were more experienced in algorithmic modelling, the number of issues with translation of a design idea into a programming algorithm

was still very high, it actually increased from 48% (day 1) to 51% (day 2). The workshop participants expressed this in a variety of ways:

- *'You understand the end product, but the way to derive it is confusing and challenges the way you think about your form.'*
- *'Not able to translate concept into script logic.'*
- *'Struggling to find a method to put what I wanted to do into reality.'*

The substantial difficulties with the Idea-to-Algorithm translation which designers and architects face when adopting the ways of programming and algorithmic modelling systems can be explained in a number of ways. To use algorithmic design tools, one has to step back from direct manipulations with the form itself. Instead one has to focus on developing a logic/step by step algorithm of a design solution. This takes a particular attitude of mind, which people with typical design backgrounds often find alien and counterintuitive (Woodbury, 2010). The algorithmic design technology requires a designer to think and act like a programmer (design developer) and therefore it inevitably affects the design process itself (Shih, Williams, Gu, 2011). The technology shifts from being a passive (inert) aid tool, which replicated conventional form-making principles, to being a system which enables novel principles of design thinking (Matcha, 2007). Mastering these novel algorithmic principles, however, seems to cause substantial difficulties in half of the design population (48-50% of the participants: designers and architects, novices in visual programming).

Not knowing *how to use* programming components and commands, identified as the 'syntax problems/problems with the implementation of programming components' was the second most common category of barriers reported by participant designers. More than a third of all participants (33% on day 1 and 38% on day 2) have reported

this type of difficulty. This barrier has actually increased (become more common) when designers gained more experience in using algorithmic modelling (on the second day of the workshop). On the second day participants often knew what particular component they needed to use, for example 'divide curve' or 'project a curve onto a Brep', but they still struggled with how exactly to use it. Syntax problems are closely related to the problems with the 'Logic Connections' (day 1: 24%, day 2: 22%), when participants knew (or thought that they knew) which programming components they needed to use, but could not properly arrange/connect them. For example, on the first day of the workshop one of the common mistakes that participants made was putting the 'move' component before the 'vector' component. As one of the participants explained it: 'I want to take this curve and move it up, so it is first 'move' and then 'unit Z' (vector)'. This means that for some people these 'invisible rules' of programming languages (Ko, Myers and Aung, 2004) do not appear to be consistent or intuitive.

The frustration and most of the programming barriers can decrease after users gain enough experience (for example *selection barriers*, 'knowing what to use', which dropped from 33% to 27% on the second day of the workshop). However, some studies point out that the implementation of algorithmic functions and syntax of CAD programming languages cause difficulties not only for novice but also for advanced users (Celani, Vaz, 2012). Ko et al. claim that while experienced programming users can easily overcome barriers associated with selection, coordination and use, they still have significant difficulties caused by understanding barriers (functional errors) and information barriers (not knowing how to acquire information about the internal behaviour of a program) (Ko, Myers, Aung, 2004).

It is recognised that the use of Computer-Aided Design tools inflicts limitations on architects and designers (Walther, Robertson, Radcliffe, 2007). This study has shown that algorithmic design can inflict additional limitations, associated specifically with the use of programming. As one of the criteria investigating design ideation and ability to use algorithmic modelling for conceptual design, participants were asked to indicate whether they had to change their design because of unsurmountable programming barriers.

On a five point scale, from 1 – 'Strongly Agree' to 5 – 'Strongly Disagree' (all groups) '*I had to change my design because of programming difficulties'*, (mean value, std. deviation) Day 1: **3.04** ± 0.852, Day 2: **2.68** ± 0.745.

These results suggest that designers and architects can be substantially bounded by programming barriers, and that to a certain degree algorithmic design tools can limit designers' abilities (as tested on novice users).

Effect of the reuse of programming artefacts in algorithmic design

This study concludes that both Design Patterns (DP) and Case-Based Design (CBD) approaches to reuse of programming solutions help designers to overcome programming barriers and improve algorithmic

modelling performance. One of the main objectives of this study was to test whether the reuse of abstract and case-based programming solutions can reduce programming barriers. On each workshop day participants of all three test groups (the control (No Approach), Design Patterns and Case-Based Design groups) were asked to report the number of programming difficulties they had when modelling their conceptual designs (Exhibit 3.2) (See Methodology section).

On a five point scale: with 1 – 'Never'; 2 - '1-3 times'; 3 – '4-6 times'; 4 – '7-9 times'; and 5 - ' 10 times or more'
'*How often have you come across programming difficulties, while developing your design?'*
The No Approach group (mean, std. deviation)
Day 1 **2.88** ± 1.053
Day 2 **2.71** ±0.890 (both days median=3 – '**4-6 difficulties'**).
The Design Patterns group (mean, std. deviation)
Day 1 **2.37** ± 0.669
Day 2 **2.10** ± 0.403 (with both days median=2 – **'1-3 difficulties'**)
The Case-Based Design group
Day 1 **2.91** ± 1.039 (median=3 – '**4-6 difficulties')**
Day 2 **2.53** ±± .776 (median=2 – **'1-3 difficulties'**)

Exhibit 3.2 illustrates the outcomes of a statistical analysis (See Statistical Methods Section) of the differences in these means. The mean values of 'Programming Difficulties' for each group are shown as the

colour-coded bars: grey for the No Approach group, red for the Design Patterns group and pink for the Case-Based Design group. First day results are on the left and the second day results are on the right. The p-values were used to measure the probability that the gap in results did not happen by chance and thus that the difference in the means was statistically significant. The p-values below the 0.05 level are shown in black (Exhibit 3.2), indicating that the difference is statistically significant, the p-values above the 0.05 level are shown in light grey indicating that the difference might have happened by chance. Initial comparison tests are done between all three test groups.

The resulting 'p-value All Groups' is shown in a bigger block: for day 1 the p-value = 0.036 (on the left), for day 2 the p-value = 0.003 (on the right). Both p-values are below the 0.05 threshold, meaning that the participants of at least one test group had significantly more (or significantly less) 'Programming Difficulties' than participants of other groups. In order to determine which specific groups differ from which, additional tests were carried out. The resulting p-values are shown in the smaller (narrow) blocks: the 'p-value DP/CBD' comparing the Design Patterns and Case-Based Design groups, 'p-value DP/NA' comparing the Design Pattern group with No Approach group and 'p-value CBD/NA' comparing Case-Based Design group with No Approach group (Exhibit 3.2).

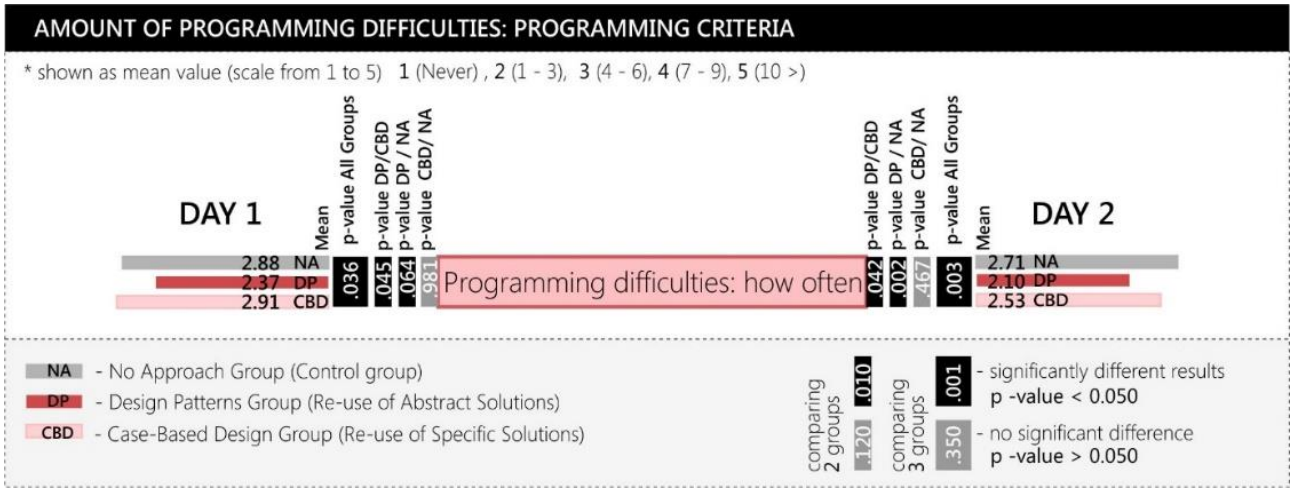


Exhibit 3.2. Number of programming difficulties, comparison between three test groups: NA, DP and CBD. [Also refer Appendix B, section Diagrams and Illustrations pages B64-B66]

This testing indicates that the reuse of abstract solutions (Design Patterns) has a significant positive effect on designers' ability to overcome programming barriers. On both days the DP group had significantly less difficulties then the NA and CBD groups (day 1 DP mean at 2.37 is significantly less than the CBD mean of 2.91 with a DP/CBD p-value = 0.045; similarly, the DP mean of 2.37 is significantly less than the NA mean of 2.88 with a DP/NA p-value = 0.064: day 2 DP/CBD p-value = 0.042; DP/NA p-value = 0.002).

The reuse of case-based solution did not prove to have a significant effect on the overall number of programming difficulties compared to the control group (day 1 CBD/NA p-value = 0.981, day 2 CBD/NA p-value = 0.467), even though on the second day of the workshop the middle number (median) of the insurmountable difficulties, which designers faced when using parametric modelling, dropped from '4-6' difficulties (day 1) to '1-3' difficulties (day 2). However the CBD approach did help to overcome certain types (categories) of programming barriers.

Comparison of the types of barriers that designers of each test group faced when using algorithmic modelling shows that the **Case-Based Design approach helps to overcome** ***use barriers:*** 'Problems with implementation (Syntax Problems)', that can be described as 'I think I know what to use, but I do not know how to use it' (Ko, Myers and Aung, 2004). Exhibit 3.3 illustrates that on both workshop days designers who used CBD approach had significantly less difficulties with 'Syntax/Component Implementation' compared to other groups. Almost half of the No Approach group participants struggled to overcome this type of programming barrier (**44.8%** on day 1 and **48.9%** on day 2). More than a third of the Deign Patterns group participants faced similar difficulties, caused by the implementation of programming components (**33.3%** on day 1 and **43.3%** on day 2). Only less than a quarter of the CBD group participants were unable to overcome these *use barriers* ('Syntax/Component implementation') (**21.3%** on day 1 and **23.4%** on day 2). When comparing all three groups, the p-values (on both days) indicate that the difference in the percentages is statistically significant (day 1 p-value = 0.049, day 2 p-value = 0.029) (See Exhibit 3.3 'p-value All Groups'). The follow-up post hoc testing (See Statistical Methods Section) confirmed that on both days the CBD group had significantly less *use barriers* ('Syntax/Component implementation') compared to the control group that used no approach (day 1 CBD/NA p-value = 0.012, day 2 CBD/NA p-value = 0.008). On the second day the CBD group had less use barriers compared to the test group that used the Design Patterns approach (day 2 DP/CBD p-value = 0.066). The 0.066 is technically above the 0.05 level, but it is very close to it. It means that there is 93.4% of certainty that the difference in results between the CBD and DP groups did not occur by chance.

The comparison of designers' ability to overcome programming barriers confirms that the reuse of programming artefacts is an effective strategy to support design and an important part of programming practice, as stated by previous studies in the field of software design [Krueger, 1992]. Algorithmic design progresses through programming; and this study illustrates that designers and architects can improve their ability to overcome programming barriers by reusing programming algorithms (both abstract and case-based), as is often done in software design. This study empirically grounds the idea that architects and designers who use algorithmic modelling tools (programming) gain from not trying to solve every problem from scratch, but, rather, reusing existing solutions that worked in the past (Gamma, Helm, Johnson, Vlissides, 1994). It further proves the point that one of the key identifiers of a designers' success is to strategically re-cycle (reuse) existing solutions instead of rediscovering them (Ibid)

Many architects and designers struggle to overcome barriers associated with the use of programming design systems. However, unlike programmers, architects and designers who use algorithmic modelling tend to rebuild programming algorithms rather than reuse existing solutions (Woodbury, 2010). The results of this experimental study support the arguments stating that the architectural design profession could learn from the computer science profession (Ibid) and start systematically reusing parametric solutions (both abstract and case-based). This can become a norm in algorithmic design practice because the reuse of programming artefacts helps to overcome difficulties with the implementation of programming languages (as proven by the reuse of case-based solutions (Exhibit 3.3)). The reuse of abstract solutions (Design Patterns) can help to improve overall performance by reducing time and

effort that end-users spend trying to surmount programming difficulties (Exhibit 3.2).

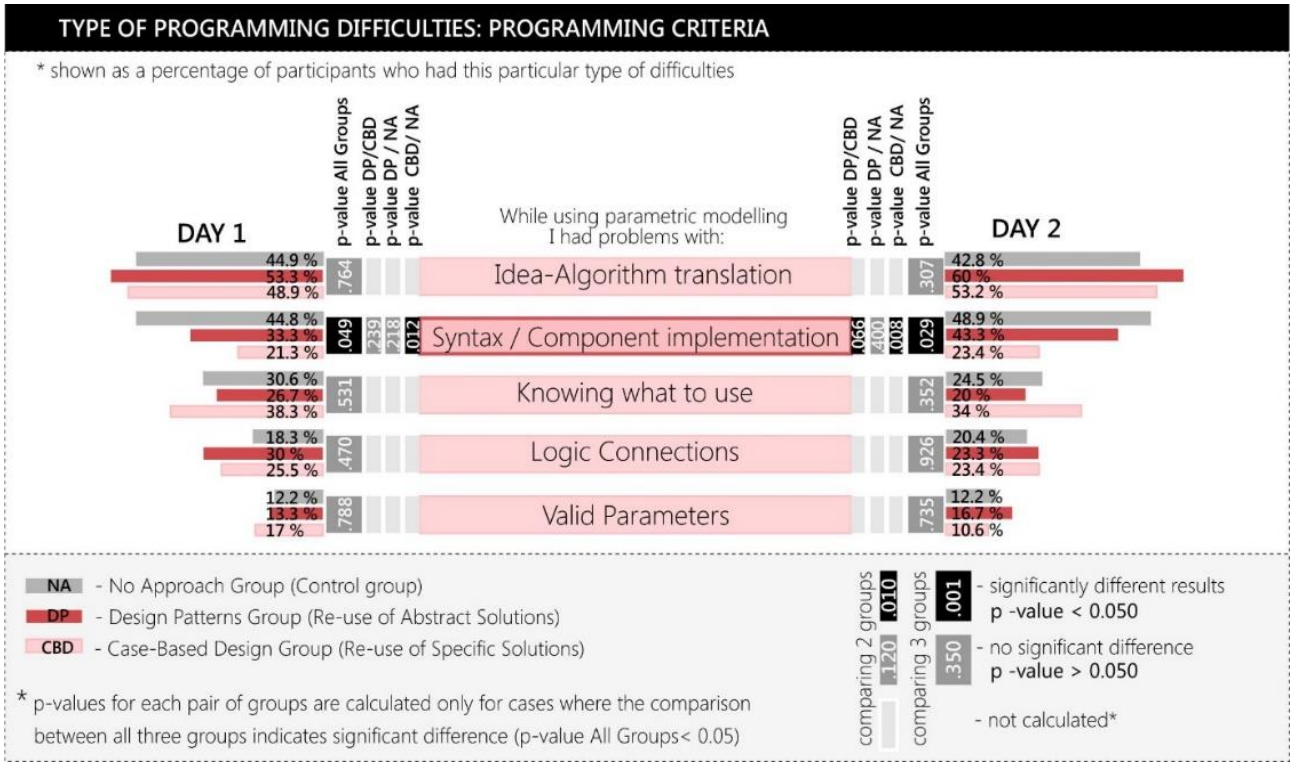


Exhibit 3.3. Types of programming difficulties, comparison between three test groups: NA, DP and CBD. [Also refer Appendix B, section Diagrams and Illustrations pages B64-B66]

In theory, it is highly probable that reuse of programming artefacts can make it easier for designers to build-up more complex algorithms, based on the existing solutions, as opposed to building everything from scratch. Some claim that the core of algorithmic design is a process of rediscovery rather than the creation of something absolutely new (Terzidis, 2006), because it is very likely that someone already did invent 'the wheel you are about to reinvent' (Mann, 2005). The re-discovery can naturally be founded on the existing algorithmic solutions (Terzidis, 2006). Results of this study show that both abstract and case-based reuse strategies can help designers to learn from existing knowledge and improve their ability to overcome programming barriers (Exhibit 3.2) (Exhibit 3.3).

'Algorithm Complexity Score' The No Approach (NA), Design Patterns (DP), Case-Based Design (CBD) groups (mean, std. deviation) Day 1 NA **40.69** ± 18.275; DP **50.60** ± 33.14; CBD **50.40** ± 30.11 Day 2 NA **54.61** ±26.988, DP **56.57** ± 28.22, CBD **53.59** ± 27.48

There is no statistically significant evidence suggesting that the reuse of programming artefacts helps designers to master complexity faster. Even though comparison of the complexity of programming algorithms produced by the participants in each test group shows that during the initial stages of learning visual programming (first day of the workshop) the participants of the DP and CBD groups managed to produce noticeably more complex algorithms compared to the control group (NA) Exhibit 3.4. On day one, two groups reusing programming artefacts (DP/CBD) produced algorithms that were 20% more complex compared to the group using no approach (NA). However statistical testing indicates that differences in average algorithm complexity between the DP/CBD and the control group (NA) are not statistically significant (p=0.136).

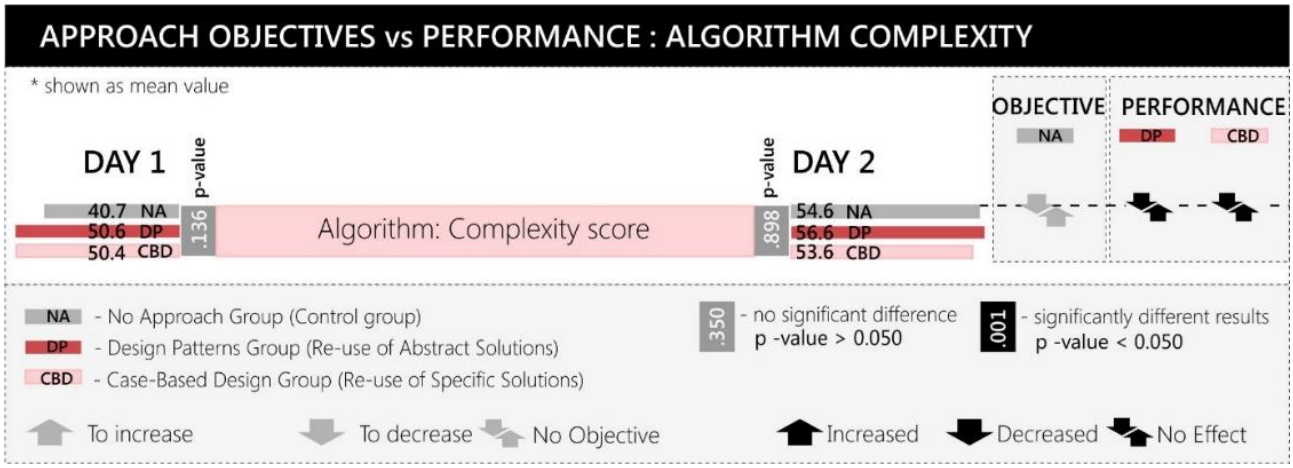


Exhibit 3.4. Algorithm Complexity, comparison between three test groups: NA, DP and CBD. [Also refer Appendix B, section Diagrams and Illustrations pages B64, B65, B69]

There are three important points that should be noted regarding these results. Firstly, statistical testing did not prove that the difference in 'Algorithm Complexity Score' is statistically significant (day 1 p-value = 0.136; day 2 p-value = 0.898, both above the 0.05 threshold). Secondly, on the second day all groups produced algorithms with very similar complexity (Exhibit 3.4). And thirdly, in general, more complex algorithms are not necessarily better algorithms. In some cases simple programming solutions can be highly effective, and likewise complex algorithms can be ineffective.

This section discussed the overall effect of the knowledge reuse approaches on participants' performance, such as their ability to overcome programming difficulties and use algorithmic modelling systems. However the study has found that in many aspects the reuse of abstract solutions and the reuse of case-based solutions had a very different effect. The following two sections discuss separately 1) testing Patterns for Parametric Designs (Woodbury, 2010); and 2) testing the use of Case-Based Design approach in the context of algorithmic modelling in architecture.

3.2 The reuse of abstract solutions in algorithmic design

Testing patterns for parametric design as a medium to reduce effort required to learn algorithmic modelling software

The Design Patterns developed by Robert Woodbury (2010) proved to be an effective medium to understand and learn algorithmic design in architecture. The pattern approach was previously tested by Gamma et al. in the context of object-oriented software design, and the results of these tests showed a number of positive effects (Gamma, Helm, Johnson, Vlissides, 1994). The authors state that the reuse of abstract programming solutions (design patterns) reduces the effort required to learn new programming software and helps during design development (Ibid). Results of this study also show that patterns for parametric design work as an effective support and learning method when introduced into design process in the field of architecture. The comparison between the performance of three test groups (No Approach Group, Design Patterns group and Case-Based Design group) shows that the use of Design Patterns helps designers to reduce programming barriers, which prove to be a big issue for a large number of end-users of algorithmic modelling tools (See 'Effect of the reuse of programming artefacts in algorithmic design' section discussing the amount of programming barriers in each test group).

The vast majority of designer and architect participants of the DP group found the Design Patterns to be very helpful. On the last day of the parametric modelling workshop participants were asked to indicate their level of agreement with the following statement:

On a five point scale from 1 – 'Strongly Disagree' to 5 – 'Strongly Agree'.

'I find Design Patterns to be a helpful medium to learn and use algorithmic modelling',

3.93 ± 0.640 (mean, std. deviation) with the median = 4 ('*Agree*')

Below are some of the participants' comments on their experience with the use of Design Patterns, as a medium to learn and use algorithmic modelling:

- *'I was introduced to parametric modelling through design patterns, and I found this to be a very successful learning method.'*
- *'They (Design Patterns) are useful starting blocks, and useful to get familiar with the types of geometry generated by program…'*

In the book 'Elements of Parametric Design' Robert Woodbury discusses the methodology for the use of thirteen Design Patterns. He describes them as reusable abstract parts and a medium to understand and express the craft of parametric modelling (Woodbury, 2010). He proposed to use Design Patterns as thinking and working tools to help designers master the complexity of algorithmic design systems. However, he points out that the (Design Patterns) method is a theory, which is yet to be tested (Ibid).

One of the objectives of this study was to test this approach to reusing abstract algorithmic solutions in design. The approach was tested using Woodbury's Design Patterns (Ibid). Therefore, the (empirically measured) results of this experimental study can be viewed as a test for Woodbury's parametric patterns theory.

Along with the evaluation of the DP approach as a method helping designers and architects to learn and use algorithmic modelling systems (Woodbury, 2010), this study also gives an opportunity to investigate how potentially (if necessary) the Design Patterns method can be improved. For example, some of the participants found the DP approach to be not very intuitive and not so easy-to-use. Although the majority of them still found patterns to be helpful. When asked to report their agreement with the statements:

On a five point scale from 1 – 'Strongly Disagree' to 5 – 'Strongly Agree'. (DP group)
'I find Design Patterns intuitive'
3.37 ± 0.718 (mean, std. deviation) with the median= 3 – **'Neither Agree nor Disagree'**
'It was easy to use the Design Patterns approach in my design'
Day 1 **2.90** ± 0.885
Day 2 **3.03** ± 0.809 (both days median = 3 **'Neither Agree nor Disagree'**)

These responses indicate that on average, participants using the DP approach would *not* refer to Design Patterns as being an intuitive method (*Neither Agree nor Disagree*), as well as they would *not* refer to it as easy to use method. These are some of the responses of the DP group participants giving their feedback regarding the usability (how easy to use) and intuitiveness of Design Patterns:

- *'They are good, but not intuitive, so perhaps looking at more examples will help to really understand what is going on.'*

- *'The hard part is taking out what is useful for your own design ideas'*
- *'That's good to get the sense of a program (Rhino/Grasshopper), but for my own design I do not know how to use it.'*
- *'More possible examples, actual cases that achieve the intended design using parametric tools'*

The feedback from designers who used Design Patterns for learning visual programming and used patterns while developing their designs revealed two main issues. The first issue, is that some of the designers found Design Patterns to be not completely intuitive. That is understandable because usually, learning through abstractions is harder (less intuitive) then learning through case-based reasoning, and it is generally easier for humans to learn by following a specific example, than to 'generalise from it' (Aamodt and Plaza, 1994). To understand each abstract set of principles (patterns) requires a designer to look at a problem from a specific pre-defined point of view. This point of view, however, might not feel natural for every individual. The name of a design pattern or the explanation (the *'why'*, the *'what'* and the *'how'*) of an abstraction may not necessarily agree with each person's intuitive way of thinking and reasoning, which can potentially lead to the increase of intellectual effort.

The second issue is related to the application (actual reuse) of Design Patterns for individual designs. Some of participants found it hard to figure out which patterns could be useful (reusable) for their own design ideas. In order to apply Design Patterns, designers have to use them as thinking and working tools (Woodbury, 2010). More often than not participants described their ideas as some certain type of geometry (design output), rather than a certain type of behaviour (programming algorithm/design pattern). Not all designers were inclined to make an additional effort of analysing their sketches (design ideas) and trying to

generalise from them (focus on the program rather than the form). That is why, sometimes, when the examples used to explain a Design Pattern did not contain the type of geometry that visually resonated with the participant's own design concept, a pattern was dismissed as not fitting.

Both of outlined issues could potentially be improved (as was suggested by some participants of the DP group) with introducing additional examples (pattern samples) to the DP approach, perhaps developing a library of cases for each Design Pattern, that cover multiple practical (visually diverse) applications of patterns. The strategy of re-enforcing case-based reasoning in the use of generalised constructs (Design Patterns), can help designers to better understand abstractions and easier locate patterns that can be used for their own design solutions (engage thinking by analogy). Similar strategy was used by Gamma et al, in the field of software design, and it was observed that introducing patterns together with examples is an effective way to teach object oriented design by example (Gamma, Helm, Johnson, Vlissides, 1994).

However, despite the issues with *intuitiveness* and *design application*, most of the DP group participants agree that Design Patterns are an effective medium to understand and learn the principles of algorithmic modelling. This approach is an effective support method and definitely preferable to having no approach for learning programming in architecture and design. Participants of the DP group found the use of abstract programming solutions (Design Patterns) to be *'useful starting blocks'** and *'a very successful learning method'** (*quoting participants of the DP group). From the teaching perspective the collection of thirteen patterns for parametric design seems to work very well, providing novices in algorithmic design with a profound and systematic insight into the basic vocabulary of algorithmic modelling methods (as evidenced from the

significant reduction of programming barriers and the positive feedback from the DP group participants). (See Appendix for 'Proposed curriculum for teaching programming in architecture using Design patterns').

The reuse of abstract constructs as a method to reduce complexity and aid design performance.

Correlational analysis indicates that those designers who easily grasp the idea of Design Patterns (abstractions) and effectively use them as building blocks in their own designs also have less programming difficulties and better algorithmic modelling performance. One of the objectives of the correlation analysis was to investigate the relationship (statistical dependency) between the designers' ability to overcome programming barriers and the feedback regarding their experience with the reuse approach. This particular analysis focused on the participants' performance inside the DP group, and this was performed for each test group individually.

It was observed that designers using Design Patterns were likely to perform consistently well or consistently poorly during both days in terms of overcoming programming barriers (number of programming difficulties/change in design due to programming difficulties). The programming barriers criteria (such as number of programming difficulties and change in design due to programming difficulties) have positive correlations between the results on day 1 and the results on day 2 (Exhibit 3.5):

DP group Correlations (between results on day 1 and day 2)

'*Programming Difficulties: how often'*

r = 0.371

'Change in design idea due to programming difficulties'

r = 0.356

For comparison, the group that used Case-Based Design does not have these types of dependency. The No Approach group does have a correlation between the amount of difficulties on day 1 and day 2 but no significant correlation between the day 1 and day 2 '*Change in design idea due to programming difficulties'*.

NA group Correlations (between results on day 1 and day 2)

'*Programming Difficulties: how often'*

r = 0.406

This consistency of the DP group performance (number of programming difficulties and change in design due to programming difficulties on day 1 and day 2) means that, those DP group participants who faced substantial difficulties with programming in the beginning of the course, were likely to continue having these difficulties. Likewise those participants who could better overcome programming barriers on day 1 were likely to continue having less problems on day 2. The use of the DP approach did not change this consistency. In contrast, the CBD group participants did not exhibit similar performance consistency. The group using the CBD approach did not have any significant correlation between

amount of programming barriers on day 1 and day 2. This means that CBD group participants who had only a few problems on day 1 could have faced many more difficulties on day 2, and vice versa.

Exhibit 3.5 illustrates the relationship (correlation) between the number of programming difficulties and the rest of the investigated criteria, such as ability to find a pattern which fits participants' design ideas. The diagram shows correlations between all criteria in the DP group (including such criteria groups as: Programming criteria, Design Ideation/Motivation Criteria, Approach Characteristics Criteria and Algorithmic Modelling Criteria - Exhibit 3.5 left hand side groupings). The results of the correlational analysis are shown in a form of a box-and-wire diagram. When two criteria have a significant correlation they are connected by a wire with the attached Pearson's correlation coefficient value (r) (See Statistical Methods Section). The strong dependencies (correlation coefficient r > +- 0.5) are shown as darker wires (green for the positive correlation, red for the negative correlation) the medium correlations (r from +- 0.35 to +- 0.5) are shown in the lighter colours (pink for negative and light green for positive dependency). This particular diagram highlights the correlations between the 'Programming Difficulties'/'Change in design idea due to programming difficulties' and the other criteria (the rest of correlations, which are not connected to programming difficulties, are not highlighted, and shown in light grey) (Exhibit 3.5).

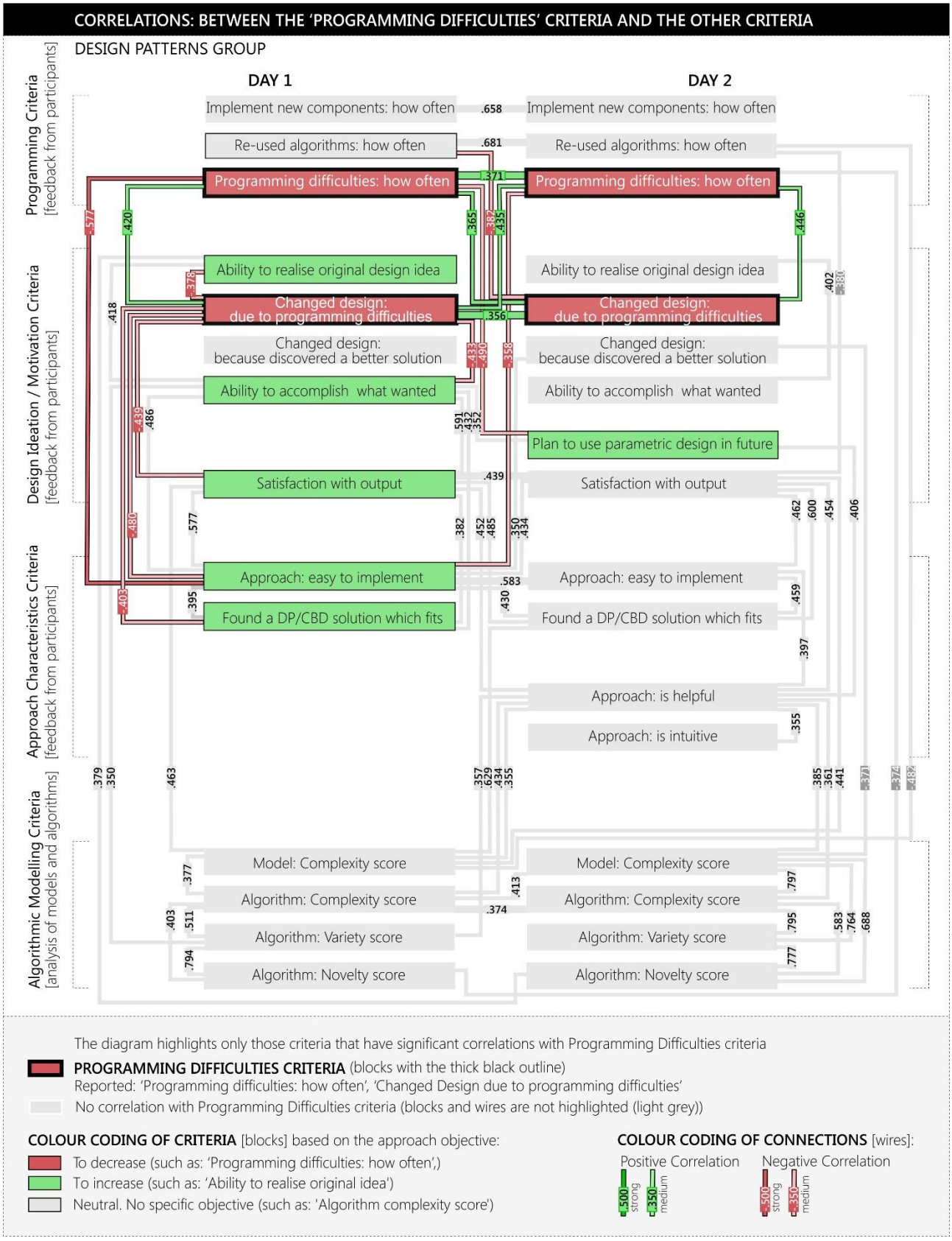


Exhibit 3.5. Design Patterns group. Correlations between 'Programming Difficulties'/'Change in design idea due to programming difficulties' and the other criteria (such as Algorithmic modelling criteria, Approach characteristics criteria, and Design Ideation/Motivation criteria). [Also refer Appendix B, pages B71-B82]

For example, the ease of the approach implementation 'It was easy to implement Design Patterns in my design' (on day 1) (Exhibit 3.5, 'Approach: easy to implement') has a strong negative correlation with the number of programming difficulties (Exhibit 3.5, Programming Difficulties: how often) on day 1 (r= - 0.577), and a medium negative correlation with the 'number of programming difficulties' on day 2 (r=- 0.358). These negative correlations mean that when one of these variables (*easy to implement the DP approach*) is high the other is likely to be low (*number of programming difficulties*) and vice versa. This seems to suggest that when participants were able to easily understand and successfully implement abstract reusable solutions in their own designs (reporting that '*It was easy to implement Design Patterns in my design*'), they were less likely to have programming difficulties (low level of '*programming difficulties*'). These results can support the claims that the use of design patterns can reduce complexity of programming solutions acting as the reusable building blocks (Gamma, Helm, Johnson, Vlissides, 1994).

The ability to overcome programming barriers, evaluated as a degree to which participants had to change their design due to programming difficulties, correlates to how well designers were able to use Design Patterns. On day 1 '*Change in design due to programming difficulties'* has a negative correlation with both how *easy to implement* designers found the DP approach: r = - 0.480, and with their ability to figure out which pattern can be used in their own design solution ('*Found a DP/CBD solution which fits*'): r = - 0.403 (Exhibit 3.5).

The evidence of this study seems to suggest that the better designers deal with the reuse of abstract algorithmic solutions the better their design performance and their ability to overcome programming difficulties. However, these findings can be interpreted in two different

ways. Firstly, this dependency might suggest that it is the effective use of patterns that helps designers to perform better at algorithmic modelling. It can be reasoned that participants learned the patterns for parametric design *before* they started to work on their design task. Therefore their performance was influenced by their ability to use patterns and not the other way around. Secondly, it can be reasonably argued that those people who are naturally more inclined to using algorithmic modelling and programming, are also more likely to understand and use Design Patterns easier than others. Either of the interpretations has valid points and it is highly likely that the actual reality is somewhere in-between these two points. Nevertheless there is clear evidence that the designers' ability to use patterns and their ability to use algorithmic modelling systems have a statistically significant positive correlation (Exhibit 3.5).

Exhibit 3.6 shows the results of the investigation regarding the relationship between the designers' performance (such as '*Ability to accomplish what was wanted', 'Ability to realise original design idea' and 'Satisfaction with output' etc.*) and the rest of the evaluation matrix, including participants' feedback regarding the use of Design Patterns. The DP group designers' *'Ability to realise original design idea'*, which participants envisioned and sketched prior to modelling, did not prove to have any significant relationship (correlation) with the approach criteria (how easy to use, how helpful etc.) (Exhibit 3.6). However, the rest of the design performance measures (including 'Ability *to accomplish what was wanted' and 'Satisfaction with output'*) have statistically significant positive correlations with the DP approach measures ('*Approach: easy to use'*, *'Found DP/CBD solution which fits'* and *'Approach is helpful'*).

These positive correlations mean that when one group of variables (positive feedback regarding the use of the DP approach) is high the other

group of variables is likely to be high as well (design performance including *'Satisfaction with output'*, '*Ability to accomplish what was wanted*'). Likewise, when approach measures are low the designers' performance measures are likely to be low as well. These results might indicate that the more successful designers with the use of patterns the better is their design performance.

Designers who identified and reused patterns in their own algorithmic solutions were more likely to accomplish their design objectives. *'Ability to accomplish what was wanted'* on day 1 is correlated with '*Approach: easy to implement':* r = 0.486, and also correlated with '*Found a DP solution which fits*': r = 0.432 (Exhibit 3.6). The satisfaction with the produced designs has also a positive dependency with how effectively designers were using algorithmic Design Patterns. *'Satisfaction with output'* has a strong positive correlation with how easy it was for designers to reuse abstract algorithmic solutions ('*Approach: easy to implement*'): r = 0.577 (on day 1), r = 0.462/r= 0.434 (on day 2) (Exhibit 3.6). The satisfaction with the produced designs is correlated with the designers' ability to find a pattern (or several patterns) that can be used in their own designs (*'Satisfaction with output'*/*'Found a DP solution which fits'*) r = 0.382/r = 0.485 (on day 1), r = 0.600 (on day 2) (Exhibit 3.6). This means that those participants who could identify patterns that fit their design solutions and could implement patterns in their designs were more likely to be satisfied with the results of their design work.

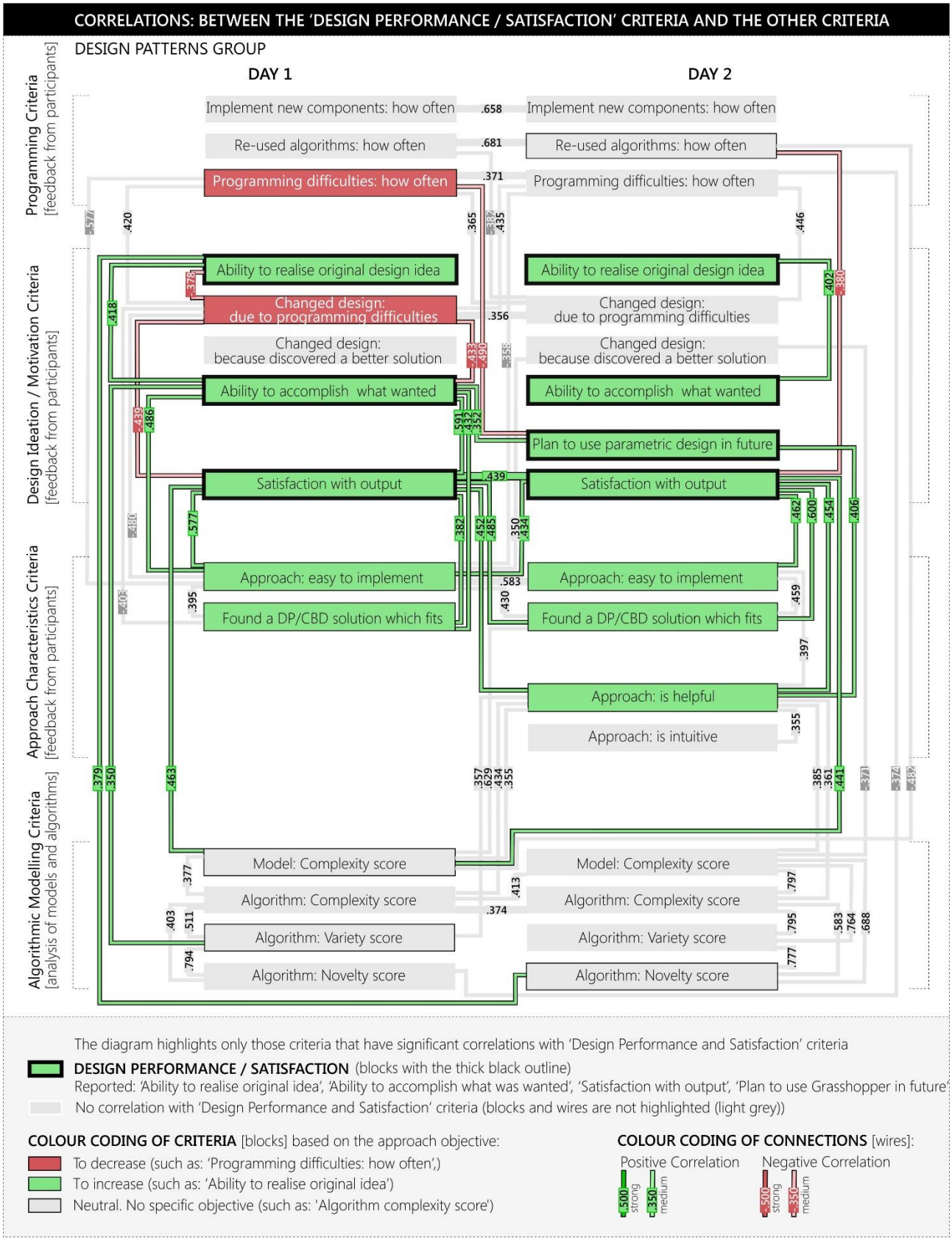


Exhibit 3.6. Design Patterns group. Correlations between 'Ability to realise original design idea', 'Ability to accomplish what was wanted', 'Satisfaction with output' and the other criteria (such as Algorithmic modelling criteria, Approach characteristics criteria, and Programming criteria). [Also refer Appendix B, pages B71-B82]

These results indicate that the reuse of abstract constructs proves to be an effective method to reduce complexity and aid design performance. Even though it may not be entirely easy-to-use or intuitive for some designers. It provides a great insight into the logic of algorithmic modelling (helps to learn/overcome programming barriers) and when duly used (reused) patterns help to improve design productivity. Moreover, those designers who found the approach to reuse abstract solutions to be helpful for learning and using algorithmic modelling, also reported a greater satisfaction with the produced designs and higher motivation to use algorithmic modelling in future. The *'Approach: is helpful'* criterion has a positive correlation with designers' *'Satisfaction with output'* r =0.452 (on day 1), r = 0.454 (on day 2), and with *'I plan to use parametric design in future'* r = 0.406 (Exhibit 3.6).

The reuse of abstract algorithmic solutions helps to explore and experiment

Along with reducing programming barriers and helping with design performance, the reuse of abstract algorithmic solutions also helps designers to increase the explored solution space and motivates them to 'go beyond' and experiment. Gamma et al (1993) states that among a number of positive effects, observed when the use of design patterns was tested in the field of object-oriented software design, some directly relate to the increase of the explored space of programming solutions. Authors state that the reuse of abstract programming artefacts helps end-users to explore alternative design solutions and motivate them 'to go beyond concrete objects' (Gamma, Helm, Johnson, Vlissides, 1993). This 'enhancing exploration' effect of the pattern approach proves to be also true when applied in the field of architectural algorithmic design.

The results of this study support the validity of Gamma et al. observations. The reuse of abstract programming artefacts encourages and supports design exploration, as tested in the context of visual programming in architecture. To come to these conclusions, three different aspects were analysed and compared between the test groups (NA, DP, and CBD): 1) how the DP approach affects the change in design objectives; 2) the explored space of programming solutions; 3) correlational analysis ('Algorithmic Modelling' criteria, 'Approach characteristics' and 'Design Ideation' criteria)

The comparison of design objectives, revealed the fact that the use of the Design Patterns (DP) and Case-Based Design (CBD) approaches has a statistically significant effect not only on the design performance, but also on design ideation: on how designers think and what design goals they choose to pursue. It was identified that the reuse of abstract and case-based programming artefacts causes a substantial shift in design objectives (Exhibit 3.7). Exhibit 3.7 illustrates the distribution of the design objectives (significantly different between the DP group and the control group (NA)). The diagram shows results for each test group (shown in percentages) as well as the results of statistical comparisons (shown as the p-values; note that the p-values below the 0.05 level indicate statistically significant differences in results) (See Statistical Analysis section for more detail on statistical measures). Originally, all test group participants were asked to describe their goals and intentions for each of their designs (individual design tasks on day 1 and day 2) in the form of an open ended enquiry: *'What did you want to achieve/accomplish for this design task?'*

Five most common types of design objectives and intentions were identified using the feedback from the participants of the algorithmic

modelling workshops (listed from most popular to least popular) (Exhibit 3.7):

- to achieve what was originally sketched;
- to explore algorithmic form-making;
- to experiment with parameters;
- to apply the programming components and logic that was learned;
- to combine a few of existing algorithmic solutions (design patterns or specific programming algorithms);

The difference between the test groups in three of these categories has proved to be statistically significant. Two of those differences can be regarded as the effect of the DP approach (Exhibit 3.7). Firstly, statistical testing, shows that the use of both the DP and CBD approaches motivates designers *'to explore algorithmic form-making'.* The difference manifests itself mostly on day 1, p – value = 0.014 (comparing all three groups). More than a half (63%) of the Design Patterns group participants wanted to explore algorithmic form-making. 63% is significantly more compared to approximately a quarter (24%) of the No Approach group (p-value = 0.004) and slightly more than a Case-Based Design group 46.8% (p-value = 0.049) (Exhibit 3.7). On the second day of the workshop statistical testing did not indicate any statistically significant difference in results between the three test groups (p-value = 0.263, comparing all three groups). However, the DP group was still noticeably more motivated *'to explore',* compared to the other two groups: NA – 28%, DP – 40%, CBD 23.4% (Exhibit 3.7).

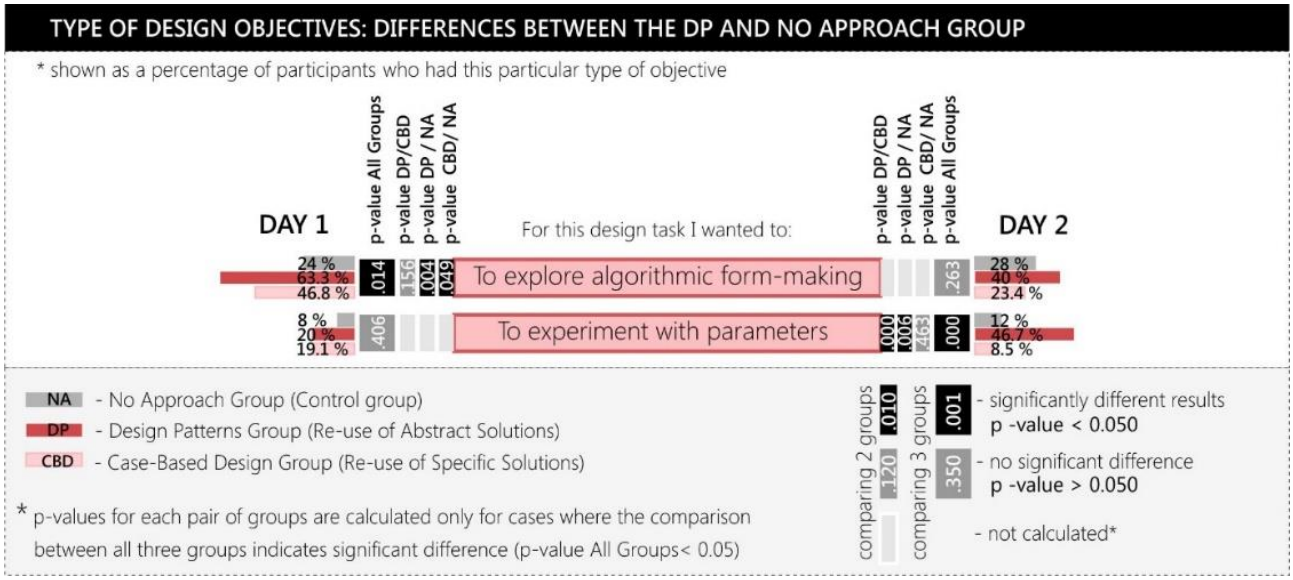


Exhibit 3.7. Types of Design Objectives. Comparison between the test groups, [Also refer Appendix B, pages B64, B67]

Secondly, the DP group was not only interested in exploration of algorithmic form-finding modelling techniques, but this group's participants were also highly invested in the experimentations with parameters and alternative variations of their programming algorithms and output models. On the second day of the workshops, almost half of the designers reusing abstract solutions (46.7%) reported experimentation with parameters as one of their design objectives (Exhibit 3.7). This percentage is considerably higher compared to both control group (12 %) and the CBD group (8.5%). These results suggest that the use of Design Patterns has a significant effect on the way designers think, shifting their interest towards exploration and experimentation.

Design Objective: 'To experiment with parameters'
Day 1: NA 8 %, DP 20%, CBD 19.1% (p-value All Groups = 0.406)
Day 2: NA 12 %, DP 46.7%, CBD 8.5%
(p-value All Groups = 0.000, p – value DP/NA = 0.006, p-value = 0.000)

That apparent shift in the design objectives had an effect on the design process and on the design output. The evidence from the analysis of programming algorithms and comparison of results between the test groups suggests that on day 1 the group using Design Patterns had a greater range (variety) of explored space of programming solutions (See Methodology Section for more detail on evaluation criteria and the Novelty/Variety point systems). The explored space of algorithmic solutions was evaluated through two criteria: Novelty (how original/not typical a solution is on a group level) and Variety (how wide is the range of implemented programming components/logic) (Exhibit 3.8).

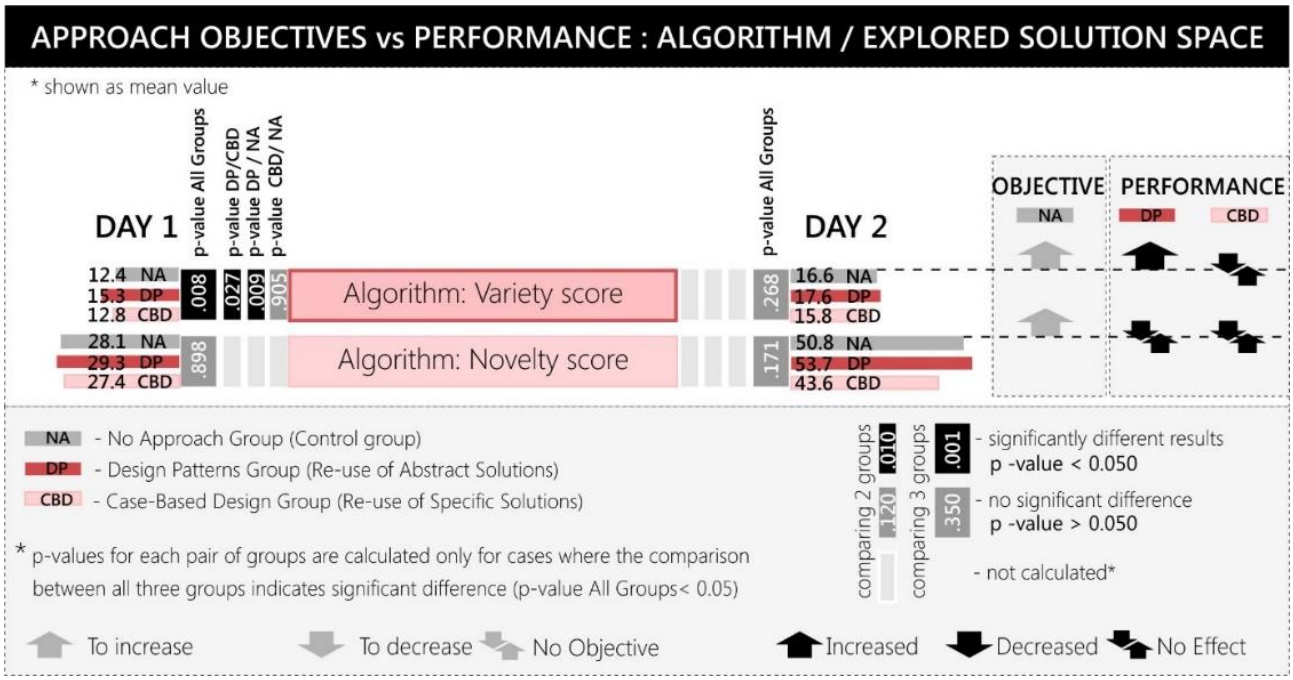


Exhibit 3.8. Algorithmic Modelling. Explored Space of Programming Solutions. Comparison between the groups [Also refer Appendix B, pages B64-B69]

The Variety measure of the explored solution space is significantly greater in the DP group compared to both NA and CBD groups (Exhibit 3.8). The statistically significant difference occurs on day 1, when designers are still in the early stages of mastering visual programming and using algorithmic modelling as a design tool. On the second day the difference

between the groups evens out, even though the DP group still have the biggest Variety score average (Exhibit 3.8). These results suggest that in the initial stages of learning the reuse of abstract programming artefacts helps designers to increase the explored solution space and produce algorithms with the wider range of implemented programming logic.

Variety Score of programming algorithms (mean value) Day 1: NA 12.4, DP 15.3 CBD 12.8 (p-value All Groups = 0.008, NA/DP p-value = 0.009, DP/CBD p-value = 0.027) Day 2: NA 16.6, DP 17.6, CBD 15.8 (p-value All Groups = 0.268)

The Novelty scores of the algorithms, produced by participants of the NA, DP, and CBD groups, were not significantly different (day 1 p-value = 0.898, day 2 p-value = 0.171) (Exhibit 3.8). This indicates that on average, designers off all three test groups produced algorithmic solutions of similar novelty. Some of those solutions were more typical, containing logic often repeated by other participants. Some solutions were very unusual, containing original logic and programming components that were never used by other participants of the workshops. It should be noted that even though the statistical testing does not indicate any significant difference in Novelty scores, on both days the DP group algorithms had the highest average scores for both Novelty and Variety criteria. Therefore, based on the evidence that the Variety scores of the DP group are significantly higher compared to the control and CBD groups, it can be concluded that overall the DP group had a greater explored space of programming solutions.

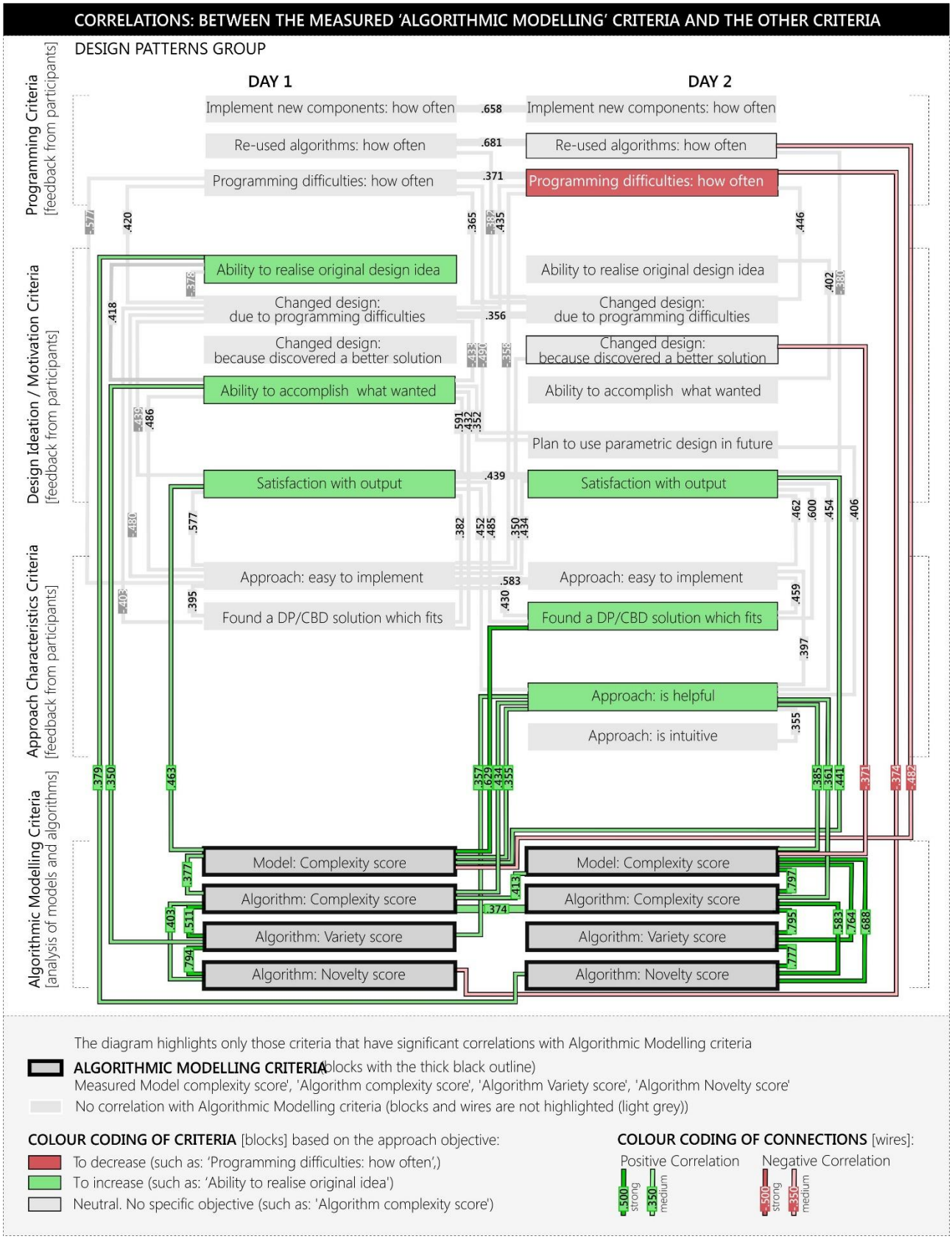


Exhibit 3.9. Design Patterns group. Correlations between Algorithmic Modelling criteria (Model and Algorithm Complexity, Explored solution space) and the other criteria. [Also refer Appendix B, pages B71-B82]

The use of the DP approach supports and encourages investigation and exploration. The correlational study indicates that when the designers, explore a wider range of programming solutions and produce more complex algorithms and models, they also find the DP approach to be more helpful and are more likely to be satisfied with what they were able to accomplish (Exhibit 3.9). The DP group's *'Algorithm Variety score'* on day 1 has a significant positive correlation with *'Ability to accomplish what was wanted'* (r = 0.379) and with how helpful designers found the DP approach (*'Approach: is helpful'* r = 0.357) (Exhibit 3.9).

There is strong evidence indicating that the use of patterns for parametric design encourages complexity. That includes the higher complexity levels in both programming algorithms and output design models. Correlational analysis shows that the higher levels of model and algorithm complexity is regarded as a positive quality by the DP group participants. *'Model complexity'* is positively correlated with *'Satisfaction with output'* (r = 0.463/r = 0.441) (Exhibit 3.9). This means that participants of the DP group were likely to be more satisfied when they produced more complex design models. Moreover '*Model complexity'* has a positive correlation with *'Found a DP solution which fits'* (r = 0.629); and with *'Approach: helpful'* criteria (r = 0.355/r = 0.385) (Exhibit 3.9). This suggests that participants who successfully implemented patterns in their own design solutions were more likely to produce more complex models as output.

There is also evidence indicating that when designers reusing abstract solutions were able to produce more complex algorithms, they were more content, finding the DP approach to be very useful. *'Algorithm complexity'* has a positive correlation with *'Approach: helpful'* criterion (r = 0.434).

To summarise the effect of the reuse of abstract algorithmic solution on design exploration:

- The reuse of abstract algorithmic solutions has a significant effect on design goals and intentions. The group using Design Patterns approach was significantly more invested in exploration of algorithmic form-making and experimentation with parameters compared to other test groups. (Exhibit 3.7 See design objectives)
- The use of Design Patterns helps to increase the explored space of programming solutions, as indicated by the comparison of the Variety and Novelty levels of programming solutions (Exhibit 3.8) This exploration enhancement effect of the reuse of abstract programming artefacts was previously pointed out by Gamma et al., who tested patterns in the field of software design (Gamma, Helm, Johnson, Vlissides, 1993).
- The higher levels of algorithm and model complexity as well as higher explored space (variety) of programming solutions and are perceived in a positive light by participants of the DP group. The higher model and algorithm complexity is also associated with the higher levels of approach utility (how useful designers find the DP approach) (Exhibit 3.9). (See Appendix B, pages B56-B63) (Also refer Section 3.4 Comparison between reuse approaches: abstraction versus case-based)

The relationship between the level of abstraction of algorithmic solutions and their reusability

The level of abstraction of the reusable artefacts does not necessarily correspond to their reusability. The comparison between the test groups

reusing algorithmic solutions with different levels of abstraction shows that the CBD group has reused significantly more case-based programming solutions, compared to the DP group, which reused abstract solutions (Design Patterns). The CBD and DP approaches are on the opposite sides of the abstraction spectrum. The Case-Based Design refers to the reuse of *specific* solutions, developed within a *narrow* design context, and there is literally no abstraction in these reusable artefacts per se (Kolodner, 1993). Design Patterns, on the other hand, by definition are *abstract solutions*, which refer to a general concept or idea and can be *applied to a shared problem* (Woodbury, 2010), (Gamma, Helm, Johnson, Vlissides, 1994). However, the approaches are not entirely specific (CBD) or abstract (DP). For example, the online Case-Base platform uses labels (indexes) as a grouping and search principle, thus this system employs certain aspects of generalisation (abstraction). Similarly, each Design Pattern has a series of examples, illustrating the abstract concept, and the use of examples is a trait of case-based reasoning approach. Nevertheless, overall, Design Patterns justifiably represent the reuse of abstract parametric solutions, while a repository of specific programming cases does clearly represent case-based reasoning.

The relationship between the levels of abstraction and reusability has often been discussed in literature. Contrary to the findings of this study, it was often suggested in literature that an effective reuse technology suggests the use of high level of abstraction (Krueger, 1992). First of all, it is argued that it is more efficient to capture 'big ideas' instead of covering every possible design solution (Winn, Calder 2002). Additionally, abstract solutions have an advantage of being applicable to a large range of design problems regardless of a particular design platform and technology (Ibid). That is why it is claimed that abstraction plays an essential role in any reuse

method, and reusability and abstraction are strongly related (Krueger, 1992).

The findings of this study suggest that the claims regarding the linear relationship between the reusability and abstraction (while in theory being very sound) might not necessarily be true in practice. Comparison between the amount of the reused programming artefacts of the DP and CBD groups show that, a systematically organised and reasonably large case-base of specific (not abstract) algorithmic solutions can provide means for an efficient reuse method. It also shows that the high level of abstraction of the reusable artefacts does not automatically ensure their high reusability. It should be noted that the use of Design Patterns (DP group) and the use the Case-Based solutions (CBD group) was highly encouraged, but not strictly compulsory. Designers of both test groups were free to decide for themselves whether to reuse the respective DP/CBD solutions in their designs or to create their programming algorithms from scratch.

Prior to the design tasks, participants of the DP group were explained the 'why', 'when', and 'how' of each Design Pattern; went through the step-by-step tutorials of the corresponding examples; and were provided with the print-outs describing and illustrating the patterns (See Methodology Section for more detail regarding the experimental set-up). It was also suggested to participants that they should give it a try, and use patterns as thinking and working design tools (Woodbury, 2010), because it would help them with the development of their design solutions. Yet, to use or not to use patterns was entirely up to designers. The CBD group participants were given the access to the online case-base of algorithmic solutions; and were shown how to use the tag search (case selection based on the assigned labels).

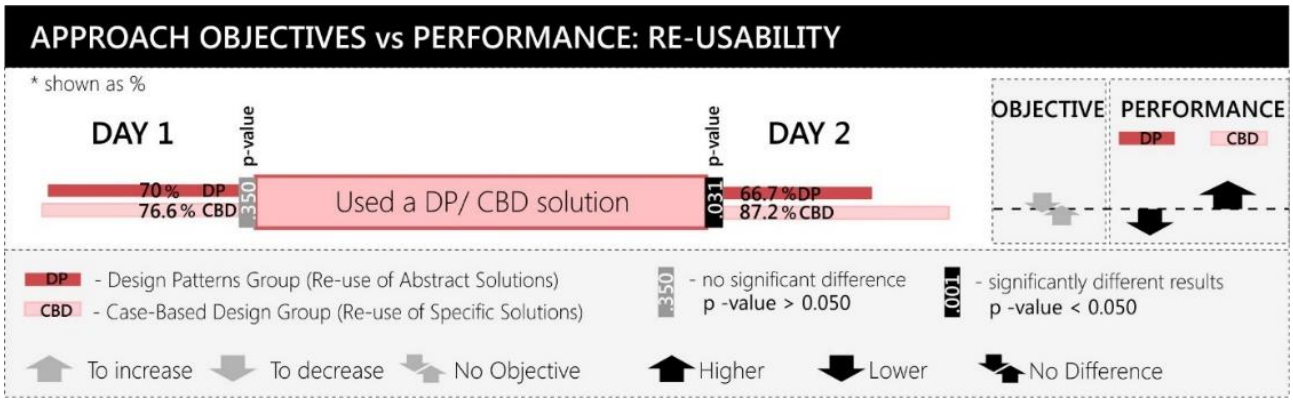


Exhibit 3.10 Reusability of abstract and case-based solutions [Also refer Appendix B, pages B64, B65]

Exhibit 3.10 illustrates that 70% of the DP group participants (on day 1) and 66.7% (on day 2) reported that they used a Design Pattern (or several patterns) in their algorithmic design solutions, while working on their individual design tasks (See Methodology Section for more detail). 'The use of a DP solution' implies that participants have either 1) explicitly identified the name of at least one of the thirteen patterns for parametric design (Ibid), or 2) that they have described a pattern using their own words. In some cases, instead of using the actual pattern names, such as 'Jig', 'Projection', or 'Point Collection' participants used words describing:

- design's geometry, such as *'Spiral'* (which can be referred back to a 'Spiral' example of the 'Increment' pattern) (Woodbury, 2010);
- modelling actions/programming components, such as *'Project'* and *'Select'* (which are not strictly speaking the patterns names but they could potentially be interpreted as corresponding, 'Projection', and 'Selector' patterns);
- in some cases participants of the DP group substituted patterns with such terms as: *'Panelling'* (which can potentially be traced back to the 'Place Holder' pattern), *'Cloud of points'* ('Point Collection' pattern), '*Gradual Repetition'* ('Increment' pattern), or other

descriptions some of which still could be traced back to the original Design Patterns: *'Reiterating pattern', 'Twists projecting up', 'Perforation', 'Lift surfaces', 'Size based on the distance', 'Weave', 'Attractor'* etc.

A part of these descriptions could be easily referred to the original pattern names, for example: *'Project'* to 'Projector' pattern or *'Size based on the distance'/'Attractor'* to 'Reactor' pattern. These cases were counted as 'Used as a DP solution' (Exhibit 3.10). With other descriptions it was harder (next to impossible) to affirmatively trace back to one of the thirteen Design Patterns, such for example as: *'Weave', 'Rotate' or 'Perforation'*. These cases were not counted as the Design Pattern use.

At times, it was almost as if designers have identified (invented) their own generalised solutions and reported them as canon patterns.

The final figures in Exhibit 3.10 show the total percentage of the DP group participants who reused patterns in their own designs: **70%** on day 1, **66.7%** on day 2. This includes cases when participants have identified original pattern names: **56.6%** on day 1, **60%** on day 2. The total percentage also includes cases which have been traced back to the original Design Patterns: **13.3%** on day 1, **6.6%** on day 2 (such cases as: *Select'* to Selector patterns, *'Project'* to Projection Pattern etc.). This means that only 56.6-60% (slightly more than a half) of the participants reusing abstract algorithmic solutions reported the use patterns, using their proper (canon) names. Other 6.6%-13.3% of the DP group participants (most probably) did use Design Patterns in their designs (as they described the core idea

corresponding to a particular pattern), but they failed to recall a proper pattern name.

Reusability: *'Used a DP/CBD solution in my own design'*
Day 1 DP **70%** CBD **76.6%** (p-value = **0.350**)
Day 2 DP **66.7%** CBD **87.2%** (p-value = **0.031**)

On day 1, the total percentage of the reuse of abstract solutions is very similar to the percentage of the reused case-based solutions (Exhibit 3.10). However, on day 2, the CBD group participants have reused significantly more (case-based) solutions (**87.2%**) compared to the DP group, reusing abstract algorithms (**66.7%**). It should be noted that a number of the CBD group participants have reused parts of algorithms that were shown during tutorials: **2.2%** on day 1, **6.4%** on day 2. Even though these reused solutions were not taken from the online case-base (CBD repository), technically speaking these solutions were still reused cases. They involved both the actual reuse of the existing algorithms and case-based reasoning. That is why the 2.2%-6.4% were included in the total percentage of the 'Used a CBD solution' criterion.

The comparison between the reused abstract and case-based solutions indicates that participants of the CBD group were reusing programming solutions more often than participants of the DP group. This might imply that specific programming artefacts can be as reusable and in some cases even more reusable than abstract programming artefacts. Therefore, in contrast the opinion expressed in literature stating that the effective reuse technology implies the use of high level of abstraction (Krueger, 1992), (Winn, Calder 2002), the evidence from this comparison indicates that the higher level of abstraction does not automatically imply the higher reusability of solutions.

3.3 The reuse of case-based solutions in algorithmic design

Case-based reasoning as a method to support algorithmic design in architecture

The reuse of Case-Based programming solutions (Case-Based Design (CBD)) proved its capacity to be a helpful method aiding the use of algorithmic design tools in architecture. The use of case-based reasoning is often discussed in the literature on both software programming and architectural design. It is claimed to be a highly effective method to solve design problems, and is argued that solutions from past design cases help architects think by analogy and solve their current design problems (Pearce, 1992) (Riesbeck, Schank, 2013). The results of this study indicate that the CBD approach (the use of case-based reasoning in design) can be as effective when applied in the context of parametric design in architecture, supporting the arguments that it is a promising 'intelligent design support' method (Heylighen, Neuckermans, 2001). Participants who used the CBD approach as a part of their algorithmic modelling process reported that they found it to be most helpful.

Both the DP and CBD test groups reported a median answer of 4 ('Agree') when responding to the question about the utility of the approaches (how helpful). This means that the majority of designers in both test groups 'agree' that the respective approaches are helpful. Therefore it can be stated that designers who learn and use algorithmic modelling for their designs find both reuse approaches to be helpful. When comparing the degree of utility of the approach, the reuse of programming solutions from the Case-Base is identified to be the more

helpful medium to learn and use algorithmic design (as reported/judged by participants).

Statistical analysis of these means shows that the CBD group reported the Case-Based Design approach to be more helpful compared to the DP group. The 0.007 p-value level means that there is 99.3% chance that the difference in results is statistically significant.

On a five point scale from 1 Strongly Disagree to 5 Strongly Agree
I find the use of the DP/CBD approach to be a helpful medium to learn and use algorithmic modelling' (Mean, std. deviation):
CBD 4.30 ± 0.507
DP 3.93 ± 0.640,
p-value = 0.007;

Outside the architectural design context, the CBD problem solving paradigm (design support approaches based on the reuse of previous experiences/case-based reasoning) is widely used in computer research and practice such as: software engineering, artificial intelligence etc. In programming, case-based reasoning has proven its high efficiency as a tool for design support, helping software developers to find solutions for their current problems by reusing past experiences (Maher, de Silva Garza, 1997) (Riesbeck, Schank, 2013). In design fields dealing with geometry, such as design, engineering and architecture, it has also been suggested that the CBD approach is a promising method (Hua, Fairings, Smith, 1996). Implementation of Case-Based Design approaches in architectural education (using non-computational design methods) has shown that students benefit from the inclusion of case-based reasoning (exposure to cases) in the design process (Heylighen, Verstijnen, 2000).

One of the objectives of this study was to test the CBD approach in the context of algorithmic modelling in architecture, which equally relates to the fields of computer programming and architectural design. Results of this research suggest that the use of Case-Based Design can be as helpful for learning and using visual programming in architecture. This is some of the feedback illustrating participants' experience with the use of the online Case-Base of algorithmic solutions (CBD approach) and the important role of the examples/thinking by analogy:

- '*I was introduced to design processes through following the examples shown and then referring to them to help me apply them to my own designs, this design approach allows a good reference and understanding of how Grasshopper works';*
- *'It is extremely helpful to have so many examples' (commenting on the role of design-cases);*
- *'It allowed me to see how it was supposed to be done' (solving problems by analogy/learning from examples)*
- *'The way we were introduced to the parametric modelling was the best and quickest way for me to learn the programming'*
- *'The examples were fantastic, so easy to follow and understand.'*

It is hard to overestimate the role of examples in education and design practice. This research shows that examples play a vital role in both understanding the theory and methods of visual programming, as well as in practical implementation of the technology. Even though the Design Patterns (DP) and the control (No Approach (NA)) group participants did not have the same access as CBD group participants to a systematically organised case-base of programming algorithms, they (in one way or another) still utilised case-based reasoning. For example, when developing their own algorithmic designs a number of the DP group participants were

more inclined to reuse a specific example (programming algorithm) illustrating the pattern rather than a pattern itself. In all the test groups some of participants chose to reuse parts of the algorithms shown during tutorials, others tried to find specific solutions online. In practice, it is almost impossible to completely avoid the use of examples (specific design solutions) when learning/implementing a new design approach or technology. Therefore, acknowledging that both reuse approaches utilise examples/case-based reasoning, the challenge (of this research) is to determine whether it is more effective to focus on the reuse of generalised solutions (abstractions) or the reuse of specific examples. In the DP approach the balance is shifted towards the maximal use of abstract solutions, while the CBD approach concentrates on the systematic reuse of specific examples.

Although, according to participants' opinion, Design Patterns were less useful than the Case-Base of algorithmic solutions, in some aspects the use of generalised (abstract) solutions had a better effect on the designers' modelling performance, such as their ability to overcome programming difficulties and the increase of the explored solution space. Even though patterns (abstract solutions), unlike specific algorithms from the case-base, did not prove to be as easy to reuse as claimed (Winn, Calder 2002), (Krueger, 1992) (See 'The relationship between the level of abstraction of algorithmic solutions and their reusability' section). Perhaps, the biggest strength of the DP approach was to give participants a broader and more structured understanding of a 'big picture' of programming methods, thus helping designers to put their mind on 'when', 'why' and 'how in principle' to use this newly acquired technology. This higher (abstracted) level of understanding might have been the reason why the novice users, who are familiar with Design Patterns, are able to apply programming logic more effectively (and consequently have

significantly less programming difficulties). Abstractions gave participants an opportunity to 'zoom out' from particular details and see/understand the underlying logic, which seems to be especially important for programming novices. That is why the use of abstract solutions, such as patterns for parametric design (Woodbury, 2010), seems likely to be more useful for educational purposes: teaching and learning of algorithmic modelling tools (compared to the CBD approach).

Relationship between examples and abstractions

There are two distinct positions identified in regard to the role of examples and abstractions (patterns). One position states that patterns' examples can be seen as elements of secondary value, while the importance is stressed on the use of abstractions (patterns) (Woodbury, 2010). The other position argues that in practice this viewpoint is not valid, because when using pattern languages, users tend to search for specific solutions rather than rely completely on abstractions (Dearden, Finlay, Allgar, McManus, 2002). The findings of this study support the arguments claiming that examples are as important as abstractions (Ibid). The participants in the DP group reported that examples played an important role in their design processes. It seems likely therefore that the abstract approach could benefit from the more systematic approach of case based reasoning to the provision and classification of examples.

The feedback from the participants, who took part in this experimental study, indicate that it is the case-based reasoning (thinking by analogy) that designers find to be the most helpful (based on the evaluation of the approach utility and participants' comments). This is very similar to analyses of the role of examples in Alexander's design pattern

language. Hua et al. claim that in practice it is extremely hard to identify the general principles which outline an abstraction. It is pointed out that, while Alexander attempted to interpret and organise design knowledge in an abstract way, what he ended up doing had little to do with generalisation, because each pattern actually refers to a set of specific buildings within specific environments (Hua, Fairings, Smith, 1996).

These arguments also seem valid in regards to Woodbury's design patterns. The Thirteen Patterns for Parametric Design are defined as a method representing the reuse of abstract solutions in design and architecture (a generalised solution which can be applied to a shared problem). Woodbury states that in order to use design patterns successfully it is essential to think with abstraction. The primary role is given to the abstraction which omits 'inessential details' (Woodbury, 2010). However, in practice the book 'Elements of Parametric Design' (which describes the patterns) is mostly comprised of carefully selected and systematically organised sets of specific examples (pattern samples). On one hand, these examples can be viewed as elements of secondary value, serving as a mere illustration and explanation of a general concept or idea (pattern). On the other hand, it can be argued that in practice it is the examples that make the whole method work. If we take all the 'inessential details'/examples away, the patterns will most likely be hard (or next to impossible) to communicate and explain to other designers. If we take away the pattern's identification and description (Name, What, When, Why, How) and leave only the subsets of examples, it is still highly possible that designers would be able to understand and reuse their overall logic.

Supporting this general conclusion is the most common response of the DP group participants to the question of how to improve the method: *'more examples'*. It seems that, even when using the abstract

constructs as a primary reuse method, it is important to acknowledge and address the significance of examples (and the reuse of specific cases). This study shows that in practice, the actual examples (case-based reasoning) provide a necessary *'reference'** and help designers to figure out how things are *'supposed to be done'** and *'how to apply them'** to their own designs (*participants' opinions regarding the role of examples).

The use of CBD as a method to reduce programming barriers

The reuse of specific examples (CBD approach) did not prove to be as effective as the DP method in aiding users to overcome programming difficulties, especially during the initial stages of learning and implementing algorithmic modelling techniques.

On a five point scale, with 1- 'Never', 2 - '1-3 times', 3 – '4-6 times', 4 - '4-6 times', 5 – '10 times or more'; *'How often have you come across insurmountable programming difficulties, while developing your design model'*, Day 1 (mean, std. deviation): NA **2.88** ± 1.053, CBD **2.91** ± 1.039, p-value = **0.981**; Day 2 (mean, std. deviation): NA **2.71** ± 0.890, CBD **2.53** ± 0.776, p-value = **0.467**;

The data suggests that, initially, (on the first workshop day) the CBD group had even more problems than the control group. However, when designers gained more experience with algorithmic modelling and the use of the case-base (on day 2) the CBD group has improved its ability to overcome programming difficulties (from day 1 median = 3: **'4-6**

problems' to day 2 median = 2: '1-3 problems'), while the control group failed to improve and remained having on average '4-6 problems' on both days. Statistical analysis (p-value levels) indicates that the differences in programming difficulties between the CBD and control groups are not statistically significant. Therefore there is no solid statistical evidence supporting the idea that the reuse of Case-Based algorithmic solutions is an effective method, which provides a way to easily generate solutions (Kolodner, 1991) or that it helps users to reduce programming barriers, even though the CBD group has improved its performance on the second day and had on average less problems compared to the control group (NA).

One of the significant (statistically determined) positive effects of the CBD approach was its capability to substantially reduce problems associated with the implementation of programming components (*use barriers*). The CBD group had two times less problems with the practical implementation of components compared to the control group (NA) and significantly less problems compared to the DP group.

Syntax Problems/Problems with implementation of functions and components:

Day 1: NA 44.8%/DP 33.3%/CBD 21.3% (p-value = 0.049/comparing all groups)

Day 2: NA 48.9%/DP 43.3%/CBD 23.4% (p-value = 0.029/comparing all groups)

These results can be easily explained, because the use of actual solutions (case-based reasoning) gives designers an opportunity to understand *'how exactly'* a certain programming algorithm (logic) can be done. The 'use barriers' (knowing what to use, but not knowing how to

use it) (Ko, Myers and Aung, 2004) are the second most common type of programming difficulties among the participants, topped only by the difficulties with the 'idea-to-algorithm translation' (I do not know what I want a computer to do). The use (implementation) barriers often occurred straight after designers figured out 'how in principle' an algorithm can be built. In theory, both abstract and case-based solutions can help designers to translate their design idea into a programming logic: Design Patterns by providing an abstract framework (construct) defining the core principles of a new solution; Case-Based Designs by giving an existing example, which a designer can reuse thinking by analogy. The advantage of the CBD approach is that it also can (and, as the results of this study show, does) help designers with practical implementation of these algorithmic solutions ('how exactly' to build a certain programming algorithm). Whereas the DP approach, by its definition, does not provide this type of information, because patterns are abstract and the sole role of samples is to illustrate this abstract idea.

The correlation analysis shows that the CBD group participants were not consistent in their ability to overcome programming difficulties on day one and day two. Exhibit 3.11 illustrates that '*Programming Difficulties*' on day 1/day 2, and '*Change in design idea due to programming difficulties*' on day 1/day 2 do not have any significant correlation. That means some participants could have faced a considerable number of insurmountable programming barriers on the first day, but on the second day they managed to perform much better and have only a few problems that they could not solve on their own. It also means that some of those who used the CBD approach and did well on the first day, on the second day faced considerably more difficulties. This might imply that the use of CBD can work really well for some designers (certain design problems/cases), but for other designers (other design problems) the

reuse of case-based algorithms can cause additional difficulties, instead of reducing them. By comparison, participants of both the DP and control (NA) groups were likely to perform similarly on both days: either having a lot of programming difficulties on day 1 and day 2, or being able to effectively overcome programming difficulties on both days. (See 'The reuse of abstract constructs as a method to reduce complexity and aid design performance' section). '*Programming Difficulties*' on day 1/day 2 have statistically significant correlations for both Design Patterns and control group participants.

Exhibit 3.11 also illustrates that the ability to overcome programming barriers of the CBD group has almost no correlation with the participants' feedback regarding the use of the approach (See the description below the diagram). For example: on both days the programming difficulties criteria (Exhibit 3.11 '*Programming difficulties: how often', 'Changed design due to programming difficulties' – red colour blocks*) had no significant correlation with how intuitive, easy to use and helpful participants found the CBD approach (Exhibit 3.11 *'Approach: helpful', 'Approach: easy to implement', 'Approach: intuitive', grey colour blocks with no connection wires to the red blocks).* Unlike the DP group participants, who were likely to have substantially less difficulties when they effectively used the Design Patterns, the CBD group participants have shown almost no dependent relationship between their ability to overcome programming barriers and their ability to find and reuse the algorithms from the case-base.

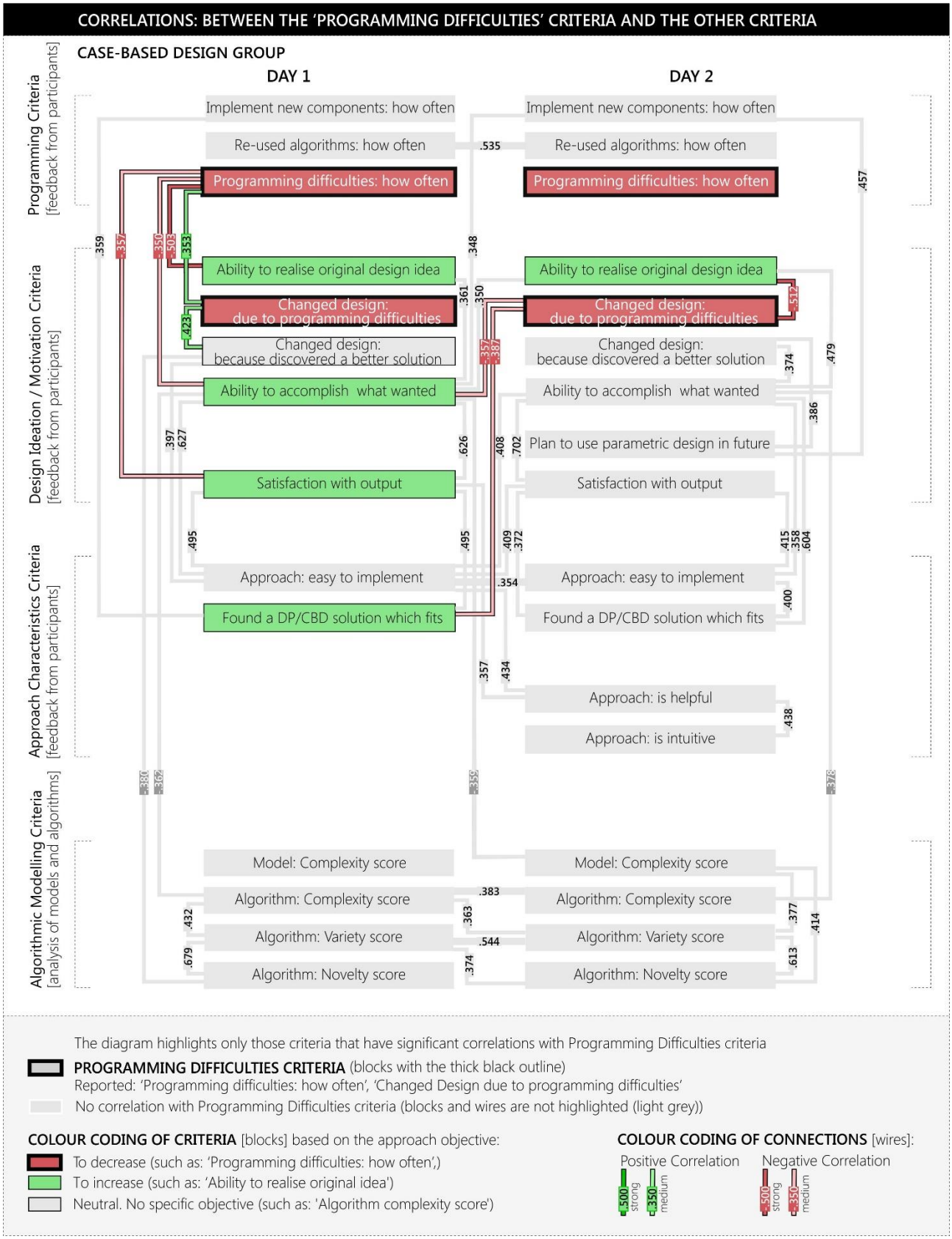


Exhibit 3.11. Case-Based Design group. Correlations between 'Programming Difficulties'/'Change in design idea due to programming difficulties' and the other criteria. [Also refer Appendix B, pages B71-B82]

It is hard to judge whether the non-consistent design performance (ability to overcome programming barriers on day one and day two) is a positive or negative influence of the approach. On one hand, it could be a positive thing that the use of the CBD approach can help designers to solve their current design problems regardless of how well they performed in the past. On the other hand, there is a chance that the reuse of the algorithmic solutions from the case-base can cause additional difficulties for those designers who previously managed to effectively use algorithmic design. This contradictory effect of the CBD approach can be therefore regarded as a potential weakness of the reuse method.

Case-based design is intuitive and easy-to-use approach

Case-Based Design proves to be an intuitive and easy-to-use support medium for algorithmic modelling in architecture. The surveys show that designers find the use of the case-base (online repository of programming solutions) to be very easy-to-use and understand. Statistical comparison between the results of the DP and CBD groups indicates that the reuse of specific solutions is significantly more intuitive than the use of abstractions (Design Patterns) (Exhibit 3.12).

On a five point scale from 1 – Strongly Disagree to 5 – Strongly Agree:
The use of the approach is intuitive‘, (Mean, std. deviation)
DP **3.37** ± 0.718,
CBD group **3.81** ± 0.851,
p-value **0.021**;

The median value (middle number in a range of values) for the intuitiveness of the DP group is 3 - *'Neither Agree nor Disagree',* the CBD median is 4 - '*Agree'.* This means that on average designers who used Design Patterns do not find the use of algorithmic abstractions to come too naturally (easily) to them. The group that used Case-Based Design, on the other hand, tend to 'agree' that the reuse of case-based solutions via online repository is intuitive. The p-value 0.021 indicates that the difference in means between the DP and CBD groups is statistically significant (as it is below the 0.05 threshold) (Exhibit 3.12).

These results were anticipated prior to conducting the experimental stage testing the DP and CBD approaches. The CBD approach was expected to be highly intuitive (easy to understand). It is often discussed (Carbonell, 1986) (Riesbeck, Schank, 2013) that problem-solving by analogy (the use of experiences from the past when solving new problems) is a default and natural way for people to solve problems. It is also pointed out that it is usually much easier to learn from a specific problem solving algorithm, than to 'generalise from it' (Aamodt, Plaza, 1994). Abstractions (Design Patterns) are in fact generalised solutions, and the use of abstract concepts often requires more intellectual effort (abstract reasoning) than the use of past cases. This happens because instinctively, humans tend to rely on specific, previously encountered situations when solving new problems (Ross, 1989), (Schank 1982), (Anderson, 2013). Reasoning by reusing past cases (case-based reasoning/CBD approach) appears to be a natural very intuitive and powerful method to solve problems for designers (Aamodt, Plaza, 1994) (Riesbeck, Schank, 2013).

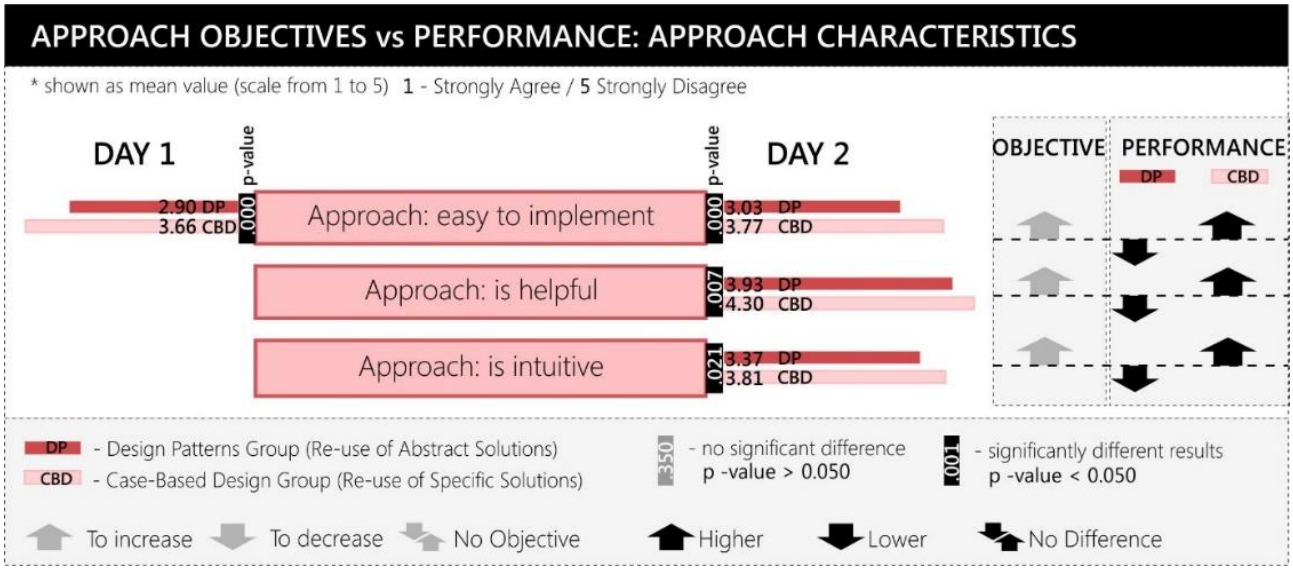


Exhibit 3.12. Approach characteristics criteria. How easy to implement, helpful and intuitive the DP and CBD approaches are. [Also refer Appendix B, pages B64-B65, B68]

It seems likely that this is why it is easier to understand and implement the CBD approach compared to the DP approach (Exhibit 3.12). Participants of the CBD and DP groups reported:

> On a five point scale from 1 – Strongly Disagree to 5 – Strongly Agree: *'It was easy to implement the Design Patterns/Case-Base of algorithmic solutions in my own design'* (day 1 and day 2 design assignments) (mean, std. deviation)
>
> Day 1: DP **2.90** ± 0.885/CBD **3.66** ± 0.668, p-value = **0.000;**
>
> Day 2: DP **3.03** ± 0.809/CBD **3.77** ± 0.666, p-value = **0.000;**

Both p-values (0.000) suggest that statistically there is almost 100% chance that the difference in results of the DP and CBD groups did not happen by chance. This empirical evidence indicates that the use of specific algorithmic solutions is considerably easier for designers then the use of abstractions (Design Patterns).

The median value for the DP group (*It was easy to implement the approach*) on both days is 3 '*Neither Agree nor Disagree'*; the CBD median is 4 '*Agree*'. Exhibit 3.12 illustrates that the results for: how easy it was to understand the approach ('Approach: is intuitive'), the ease of approach implementation ('Approach: easy to implement') and the approach usefulness ('Approach: helpful') follow a similar pattern. The CBD group mean value is always higher than the DP group mean value (Exhibit 3.12). It seems likely that the level of approach intuitiveness influences the ease of its implementation and consequently effects its usefulness (how helpful the approach is, as reported by participants). The CBD approach is easier to understand, since the use of case-based reasoning is naturally more intuitive for people than generalisation (abstraction) (Aamodt, Plaza, 1994).

It can be assumed that the use of abstract algorithmic solutions requires designers and architects to make a bigger intellectual effort (compared to the CBD approach) in order to use Design Patterns as 'thinking and working tools' (Woodbury, 2010). The correlational study shows that the reported intuitiveness of the DP and CBD approaches and their ease of implementation have a positive dependent relationship with 'how helpful' participants find each of these approaches (See Statistical Analysis section). Dependent relationship means that the two criteria have a statistically significant correlation. Positive correlation means that when one of the criteria increases the other (dependent) criteria is likely to increase as well and vice versa.

For example, the correlations to the answer to *'Approach is helpful'* and the responses to '*Approach is helpful'/ easy to implement'* were:

(Pearson's correlation coefficient - r):

'Approach: is helpful' with *'Approach: is intuitive'*

DP group r = **0.355,** CBD group r = **0.438;**

Approach: is helpful' with *'Approach: is easy to implement'*

DP group r = **0.397,** CBD group r = **0.434;**

It seems likely that in algorithmic architectural design the reuse of case-based programming solutions is considerably more intuitive and easy-to-use compared to the reuse of abstract solutions (pattern approach). Those participants who found the CBD approach to be highly intuitive and easy to use, also found it to be more helpful. It should be noted, however, that this 'helpfulness' of the approach was reported by participants themselves; it was not determined by the measured effect of the approaches (such criteria as: the ability to overcome programming barriers, explored solution space, ability to accomplish what was wanted, etc. (See methodology section)).

Relationship between participants' experience with the CBD approach and their design performance

The correlational study helps to understand and interpret the dependent relationship (correlation) between the use of case-based programming solutions (Case-Based Design approach) and participants' design performance (measured effect of the approach), such as their ability to realise their original idea or satisfaction with output. Correlation is a statistical relation between two variables. For example, in all test groups the

systematic changes in the value of 'Ability to realise original design idea' are accompanied by the systematic changes in the 'Ability to accomplish what was wanted' (correlation coefficient (r) equals 0.519) (See Statistical Analysis section for more details). Correlations can be 'positive': mutual relationship between two variables, when the value of one variable increases the other is likely to increase as well; or they can be 'negative': reciprocal relationship between two variables, when the value of one variable increases the other is likely to decrease. For example, when looking at the whole test population (participants of all groups/on day 2), the *'Ability to realise original design idea'* has a negative correlation with the *'Change in design: due to programming difficulties'* (correlation coefficient (r) equals - 0.389), meaning that when one of these variables increases the other is likely to decrease.

Exhibit 3.13 illustrates (statistically significant) correlations inside the CBD group (See Statistical Analysis section for more details). For example, the *'Ability to accomplish what was wanted'* and *'Satisfaction with output'* depend on (are positively correlated with):

- Participants' ability to find a CBD solution that they can reuse in their own designs *('Found a DP/CBD solution which fits'*) (Exhibit 3.13);
- How easy participants find the implementation of the CBD approach *('Approach: easy to implement'*) (Exhibit 3.13);
- How helpful the CBD approach is *('Approach: is helpful'*) (Exhibit 3.13);
- Participants who find an interesting CBD solution (and change their original design because of this discovered solution) are more inclined to use parametric design in future *('Changed design because discovered a better solution'/'Plan to use parametric design in future'*) (Exhibit 3.13).

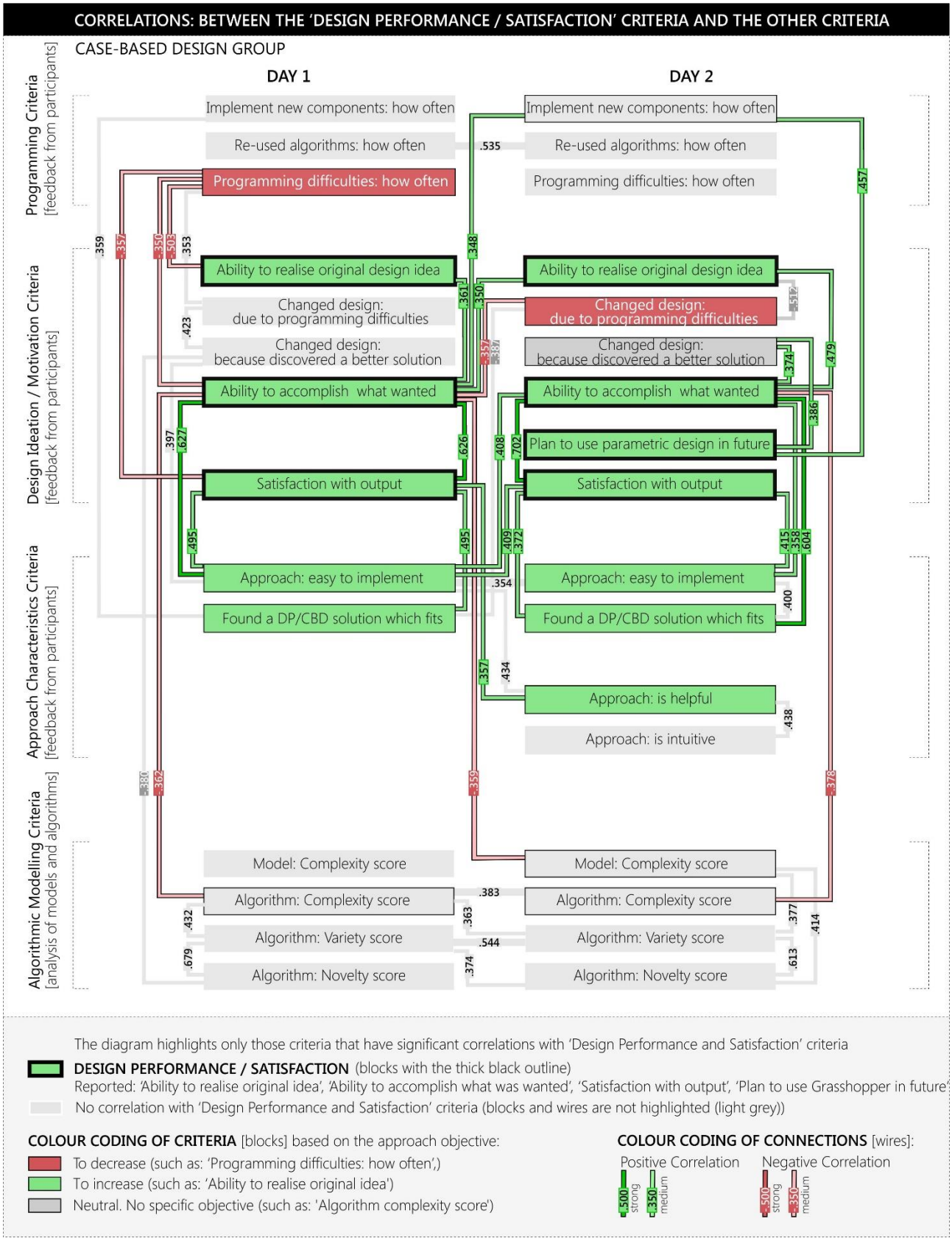


Exhibit 3.13. Case-Based Design group. Correlations between 'Ability to realise original design idea', 'Ability to accomplish what was wanted', 'Satisfaction with output', 'Plan to use algorithmic design in future' (Design Performance/Satisfaction) and the other criteria. [Also refer Appendix B, pages B71-B82]

These results suggest that participants' experience with the CBD approach is positively correlated with their design performance and satisfaction with output. Thus, those participants who could better understand and successfully use the Case-Based Design approach were also more capable to accomplish their design objectives and be more satisfied with their output designs.

When CBD group participants could find a programming solution in the case-base that they chose to reuse in their own designs, they were more likely to have a better design performance (Exhibit 3.13).

Pearson's correlation coefficient (r)
'*Found a CBD solution which fits' is correlated with 'Satisfaction with output'*
Day 1: r = **0.495**, Day 2 r = **0.372**;
'Found a CBD solution which fits' is correlated with 'Ability to accomplish what was wanted'
Day 2 r = **0.604**

The correlation coefficients (r) 0.495/0.372/0.604 indicate a positive medium-to-strong dependency between each pair of criteria (See Statistical Analysis section for more details). When one of the criteria increases the other criterion is likely to increase as well. For example, when designers are able to select a fitting reusable solution in the repository (case-base), they are more likely to accomplish their design objectives and be more satisfied with the design outcome. It also suggests that when participants, using the CBD approach, are not able to find a solution (case)

they want to reuse, they are less likely to accomplish the intended algorithmic design and are less likely to be happy with the output.

The challenge of the CBD system in this respect is to contain enough programming solutions, covering as many design problems as possible, and to be structured (organised/indexed) in such a way that the case selection process is intuitive and effective. The CBD system used in this study contained over 150 programming solutions. 76.6% of participants on day 1/87.2% of participants on day 2 reported that they successfully located and reused solutions from the repository. These CBD group percentages are significantly higher compared to the DP group who implemented Design Patterns in 70%/66.7% of cases (See 'The Relationship between the Level of Abstraction of Parametric Solutions and Their Reusability' Section).

These results indicate that the amount and range of programming solutions used in the CBD repository was sufficient to provide a base to test the CBD approach. In practice, it is next to impossible to cover all the possible solutions to all future design problems. A case in Case-Based Design can be viewed in different ways. It can be seen as a resulting solution (particular programming algorithm), or as a record of a method suggesting how to solve a problem (design strategy), or it could be seen as a lesson (design knowledge). In all of these definitions the purpose of a case in CBD is to help designers and architects to solve a similar design problem (Maher, de Silva Garza, 1997). It seems probable that the larger systematically indexed repository (containing thousands of cases) can provide a better design support, simply because it can cover more design cases. It seems reasonable to assume that when designers have more cases to select from they are more likely to be able to find what they are searching for, and (as this research shows) designers who can find a

reusable solution that fits their design idea are likely to have a better design performance *('Ability to accomplish what was wanted' and 'Satisfaction with output'*) (Exhibit 3.13).

Exhibit 3.13 also illustrates that 'Ability to accomplish what was wanted' and 'Satisfaction *with output'* are positively correlated with the ease of the approach implementation and how helpful participants find the CBD approach. These further suggest that participants' design performance is connected to their ability to use the Case-Based Design approach.

Pearson's correlation coefficient (r)

'*Approach: is easy to implement*' is correlated with *'Satisfaction with output'*

Day 1: r = **0.495/0.409**, Day 2 r = **0.415**

'*Approach: easy to implement*' *is correlated with 'Ability to accomplish what was wanted'*

Day 1: r = **0.627/0.408,** Day 2 r = **0.358)**

'*Approach: is helpful' is correlated with 'Satisfaction with output'*

Day 1: r = **0.357**

This dependent relationship between the criteria suggests that those designers (architects) who were able to understand and easily implement the CBD approach (use the repository of parametric solutions) were more likely to accomplish their design objectives and produce better* designs (*as judged by participants themselves). However, it should be noted that, even though there is a significant statistical dependency

(positive correlation) between the reuse of programming solutions and participants' design performance, this dependency can be interpreted in different ways. The first interpretation could be that the reuse of case-based programming solutions helps designers to use algorithmic modelling systems and be more capable of realising their design ideas (Exhibit 3.13). This interpretation is supported by the arguments that case-based reasoning is an effective method, helping people to solve problems by reusing previous solutions and experiences (Kolodner, 1991). Therefore, it can be reasoned that it was the successful use of the CBD approach that affected the design performance (in this case the design performance is affected by the use of the CBD approach). The second interpretation could be that there are people who are naturally (or due to previous experiences) more inclined to understand and use programming languages. These people might be more capable of mastering algorithmic design systems to realise their design ideas (*'Ability to accomplish what was wanted'*), therefore producing more satisfactory design outcomes (*'Satisfaction with output'*), and they also could be more capable of case-based reasoning in algorithmic design: finding the CBD approach *helpful* and *easy-to-use*. Regardless of the interpretations there is a dependent relationship (statistically significant correlation) between the reuse of programming solutions (CBD approach) and design performance. When one improves the other is likely to improve as well and vice versa (Exhibit 3.13).

The reuse of case-based algorithmic solutions induces more focused reasoning

Comparison of the design objectives in each test group indicates that the use of Case-Based Design (CBD) in architecture (using algorithmic

modelling tools) induces more focused design reasoning and less design experimentation (and potentially less innovative designs) (See 'The Reuse of Abstract Parametric Solutions Helps to Explore and Experiment' section). These findings correspond to the similar conclusions expressed by Peace regarding the use of case-based reasoning (CBR) in design, stating that CBR involves focused thinking, which is often applied to a narrow context of a design problem (Pearce, 1992). Statistical analysis of the experimental results shows that the use of the Case-Based Design approach affects the way participants reason and develop their designs. Comparisons between the three test groups shows that the CBD group is significantly more focused on realisation of the initial design ideas (significantly different from the control group (NA) and the Design Patterns group (DP) on the second day of the workshop) (Exhibit 3.14, See the Methodology Section).

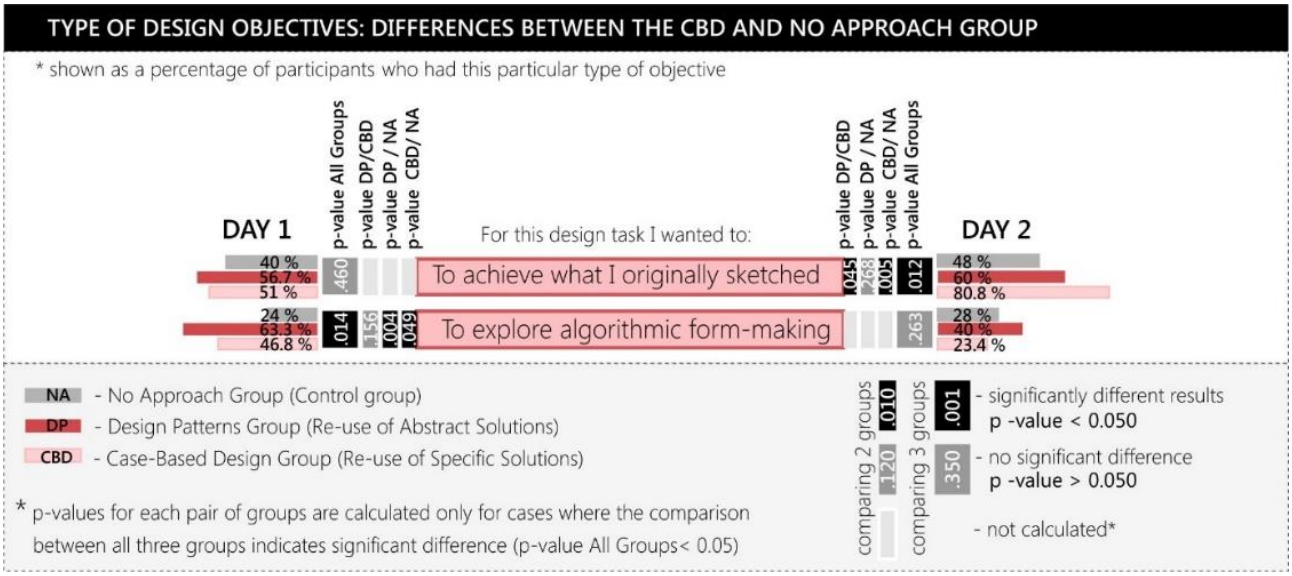


Exhibit 3.14. Design Objective criteria. Differences between the CBD and control (NA) groups. [Also refer Appendix B, pages B64, B67]

On the second day of the workshop, when designers gained more experience with visual programming and the use of the CBD system (online repository of programming solutions) the shift in design objectives becomes evident and statistically significant (Exhibit 3.14). Designers using

the CBD approach were more intent on realising their original design concepts, compared to both the DP and control groups. Statistical comparison of the objective '*To achieve what I originally sketched*' between all three groups gives the p-value, which equals 0.012.

Design Objective: *'To achieve what I originally sketched'*
Day 1: NA 40%/DP 56.7%/CBD 51% (p-value 'All Groups' = 0.460);
Day 2: NA 48%/DP 60%/CBD 80.8% (p-value 'All Groups' = 0.012);

This level (0.012) is below the 0.05 level of significance, meaning that, statistically speaking, there is at least a 98.8% chance that the difference in the results did not happen by chance (See Statistical Analysis Section). Further (Post Hoc) comparisons between each pair of test groups show that the difference between the control group (NA) and the Design Pattern group (DP) is not significant (p value DP/NA = 0.268. That suggests that a similar percent of the DP and control group participants were intent on realising their original design idea (Exhibit 3.14). The Post Hoc comparison also shows that the Case-Based Design group was significantly more focused on realising their original design idea compared to both the DP and control (NA) groups. The CBD/DP and CBD/NA p-values are both below the 0.05 level (significance level) (p-value DP/CBD = 0.045, p-value CBD/NA = 0.005), indicating significant difference in results (Exhibit 3.14). These findings suggest that the reuse of case-based parametric solutions in architecture induces more focused (narrow) thinking and design reasoning. The 'more focused' thinking and reasoning implies that it is oriented on the realisation of a particular design concept, rather than an open-ended design experimentation and exploration (See 'The Reuse of Abstract Parametric Solutions Helps to Explore and Experiment' Section).

Even though in the longer run (second day of the workshop) the CBD approach induced more focused design thinking, as opposed to abstract 'design experimentation' of the DP group; in the initial stages of learning (first day of the workshop) the use of the Case-Base also induces the 'exploration of algorithmic ways and form making logics' (Exhibit 3.14). Note: there is a difference between the 'design experimentation' (design objective: *'To experiment with parameters/model'*) and the 'exploration' of algorithmic form-making (design objective: *'To explore algorithmic form-making'*). The 'design experimentation' refers to the modification of design the model, such as changing the parameters of the programming algorithm or changing the programming logic itself to see the how the model responds (design objective focused on the experiments with the design model). The 'exploration' of algorithmic form-making refers not to the experiments with the design model itself but to finding out what are the capabilities and limitations of the algorithmic modelling system (design objective focused on the exploring technology).

Both abstract (DP) and case-based (CBD) reuse methods seem to encourage a more profound investigation of algorithmic design logic and techniques. '*To explore algorithmic form-making*' is one of the five most common categories of design objectives (identified by this study) (See Methodology Section), which refers to the exploration of the computational technology, its form-making logic and capacity: what it can and cannot do (note that it does not refer to the experimentation with the design output itself).

Design Objective *'To explore algorithmic form-making'*
Day 1: NA **24%**/DP **63.3%**/CBD **46.8%;** (p-value 'All Groups' = **0.014,**)
(p-value DP/CBD = 0.156, p-value DP/ NA = **0.004,** p-value CBD/NA = **0.049**)
Day 2: NA **28%**/DP **40%**/CBD **23.4%;** (p-value 'All Groups' = **0.263**)

These results indicate that on the first day of the workshop both DP and CBD group participants were more interested in the 'exploration of the algorithmic modelling technology (form-making)' than participants of the control group (NA) (Exhibit 3.14). The p-value comparing the percentages of participants who wanted '*to explore algorithmic form-making'* between all three test groups equals 0.014, which is below 0.050 level, meaning that results are significantly different. Further comparison between each pair of test groups shows that the DP (**63.3%**) and CBD (**46.8%**) groups had more or less similar percentages (p-value = 0.156 is above the significance level). Compared the control group, who only had **24%**, both DP and CBD group were more intent to explore the capabilities of an algorithmic design system (p-value DP/ NA = 0.004, p-value CBD/NA = 0.049 are both below the 0.050 threshold). This might indicate that during initial learning stages the reuse of abstract and case-based solutions encourages designers to explore algorithmic design technology.

Relationship between the reuse of case-based algorithmic solutions, innovation and design complexity

The shift towards more focused design reasoning in the CBD group has affected the way designers (who reused case-based solutions) built their

programming algorithms, which consequently affected the design outputs. The analysis of the algorithmic modelling criteria, such as: model complexity, algorithm complexity, and explored space of the programming solutions (algorithm Variety and Novelty), shows that the use of CBD approach has affected various aspects of the designs. Exhibit 3.15 illustrates that *'Algorithm Variety score'* on day 1 and '*Model Complexity score'* on day 2 have significantly different results, when compared between all three test groups (See a description below the diagram). The complexity levels of programming algorithms seem to be relatively similar in all groups *('Algorithm: Complexity score')*, both days p-values comparing means of the NA, DP, and CBD groups are above the significance level: day 1 p-value 'All Groups' = 0.136/day 2 p-value 'All Groups' = 0.898 (See Methodology Section) (Exhibit 3.15).

There is an ambiguity of opinions regarding the relationships between the use of Case-Based Design and explored solution space in design (Novelty and Variety) (See Methodology Section). One end of the spectrum of opinion suggests that innovative ideas often occur through the reuse of existing design solutions (Sun, Faltings, 1994), especially when two or more solutions are combined together (Hua, Fairings, Smith, 1996). Therefore, the hypothesis is that the CBD group might be expected to have a higher Novelty (original/not typical) of programming solutions, and therefore increased explored solution space. The hypothesis of those at the other end of the spectrum of opinion is that the disadvantage of the Case-Based Design approach is that 'the solution space is not fully explored', and there is no guarantee that the reused case leads to the optimal solution (Kolodner, 1991). Following this latter hypothesis the Variety (range of explored design options) of CBD group might expected to be lower compared to other test groups. The results of this

experimental study show no evidence that the reuse of case-based solutions led to innovative programming solutions. In fact, the Variety (explored solution space) of programming algorithms is consistently lower in the CBD group compared to the DP group (Exhibit 3.15).

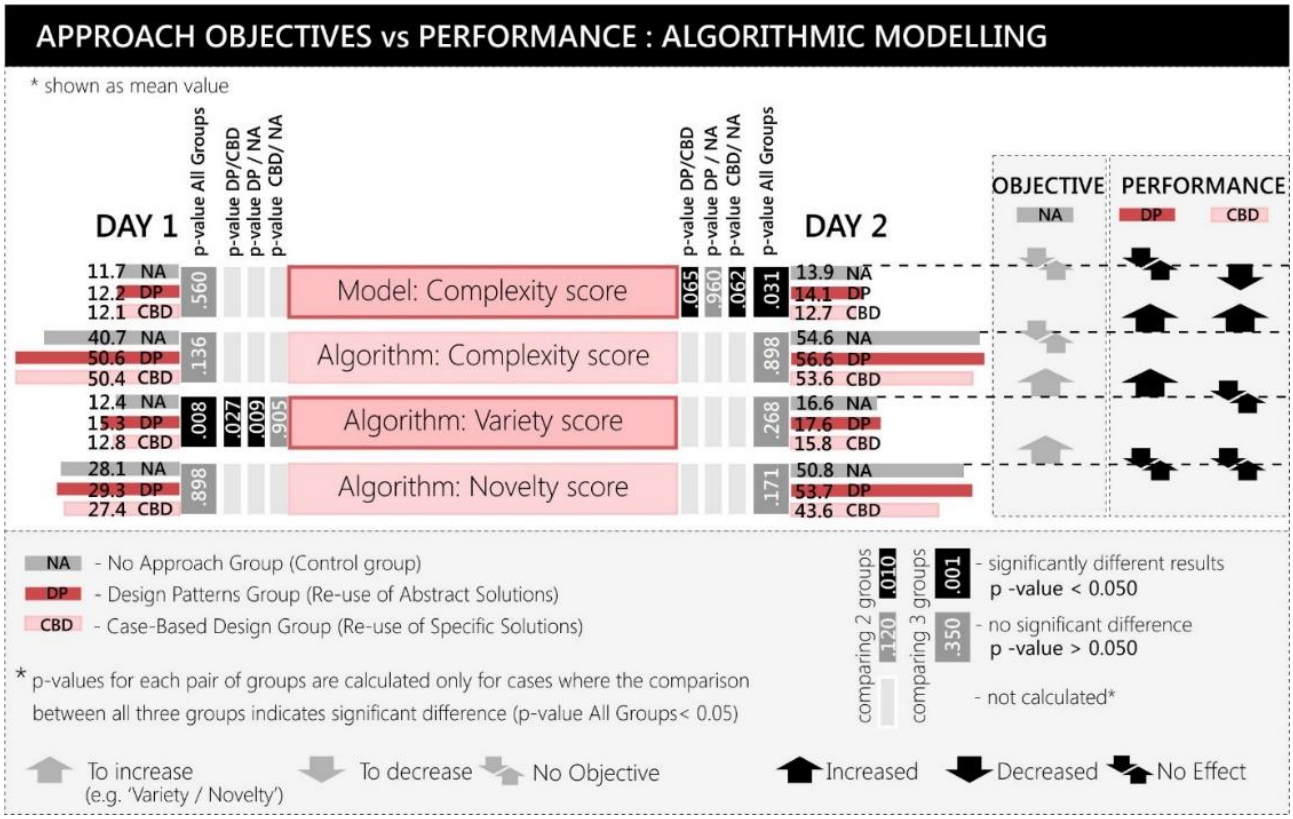


Exhibit 3.15. Algorithmic Modelling criteria: Model complexity score, Algorithm complexity score, Algorithm Variety score, Algorithm Novelty score. [Also refer Appendix B, pages B64]

The evaluation of programming solutions show that in terms of explored space of algorithmic solutions, the CBD group had a very similar range of use programming components (Algorithm Variety score) and innovation (Algorithm Novelty score) as the control group (No Approach group). On both days, the p-values comparing 'Programming Algorithm Novelty' between all groups are above the significance threshold level (0.898/0.171 both are larger than 0.050 level). Therefore, statistically there is no significant difference in the 'Novelty scores' of algorithmic solutions between the group that used no approach and the group that reused

case-based designs. This study found no evidence supporting the claims that the adaptation and combination of existing algorithmic solutions (cases) can lead to innovative designs. In fact, on the second day of the workshop the average Novelty score (evaluating the degree of how unusual/not typical the solution is) of programming algorithms developed by the CBD group is seemingly lower (**43.6**) compared to the control group (NA) (**50.8**) and to the group reusing abstract solutions (DP) (**53.7**) (Exhibit 9.5).

Programming Algorithm Variety (range of programming components): Day 1 (mean): NA - **12.4**/DP – **15.3**/CBD – **12.8** (p-value 'All Groups' = **0.008**) Day 2 (mean): NA – **16.6**/DP – **17.6**/CBD – **15.8** (p-value 'All Groups' = **0.268**) Programming Algorithm Novelty (The degree of how unusual/not typical a programming algorithm is.): Day 1 (mean): NA – **28.1**/DP – **29.3**/CBD – **27.4** (p-value 'All Groups' = **0.898**) Day 2 (mean): NA – **50.8**/DP – **53.7**/CBD – **43.6** (p-value 'All Groups' = **0.171**)

The correlational analysis shows that in the CBD group there is a negative dependency between the *'Novelty'* of programming algorithms and the reuse of case-based algorithms, which altered the original design concepts (*'Changed design: because discovered a better solution'*) (Pearson's correlation coefficient (r) equals – **0.380**) (Exhibit 3.16). This correlation might suggest that the resulting algorithmic solutions tend to be less innovative (more typical) when participants abandon or

significantly change their original design idea in favour of reusing an existing solution that does not really fit their intended design concept, but seems to be more interesting and worth changing the original plan. These findings indicate that the use of Case-Based Design in architecture can actually lead to the decrease of the explored solution space. This effect of the CBD approach might not be the most desirable especially during conceptual design stages where the experimentation and exploration of design options can make a significant difference and effect the further development of the project.

The CBD group participants' ability to accomplish their design objectives ('*Ability to accomplish what was wanted*') has a negative correlation with the complexity levels of resulting programming solutions (*'Algorithm: Complexity score'*) (Exhibit 3.16). This dependency is consistent and repeats on both days (day 1 r = - 0.362/day 2 r = - 0.378). Notice that the correlation is negative (reciprocal relationship between the variables), meaning that the CBD group participants were more likely to be satisfied when they managed to realise their design concepts using less complex algorithms. Accordingly they reported that they were able to accomplish less when they had to develop more complex programming algorithms in order to generate the intended outcome (design model). Interestingly, the situation in the DP is the opposite: participants who reused abstract solutions are likely to be more satisfied when the complexity of programming algorithms and output models is higher (See 'The Reuse of Abstract Parametric Solutions Helps to Explore and Experiment' Section).

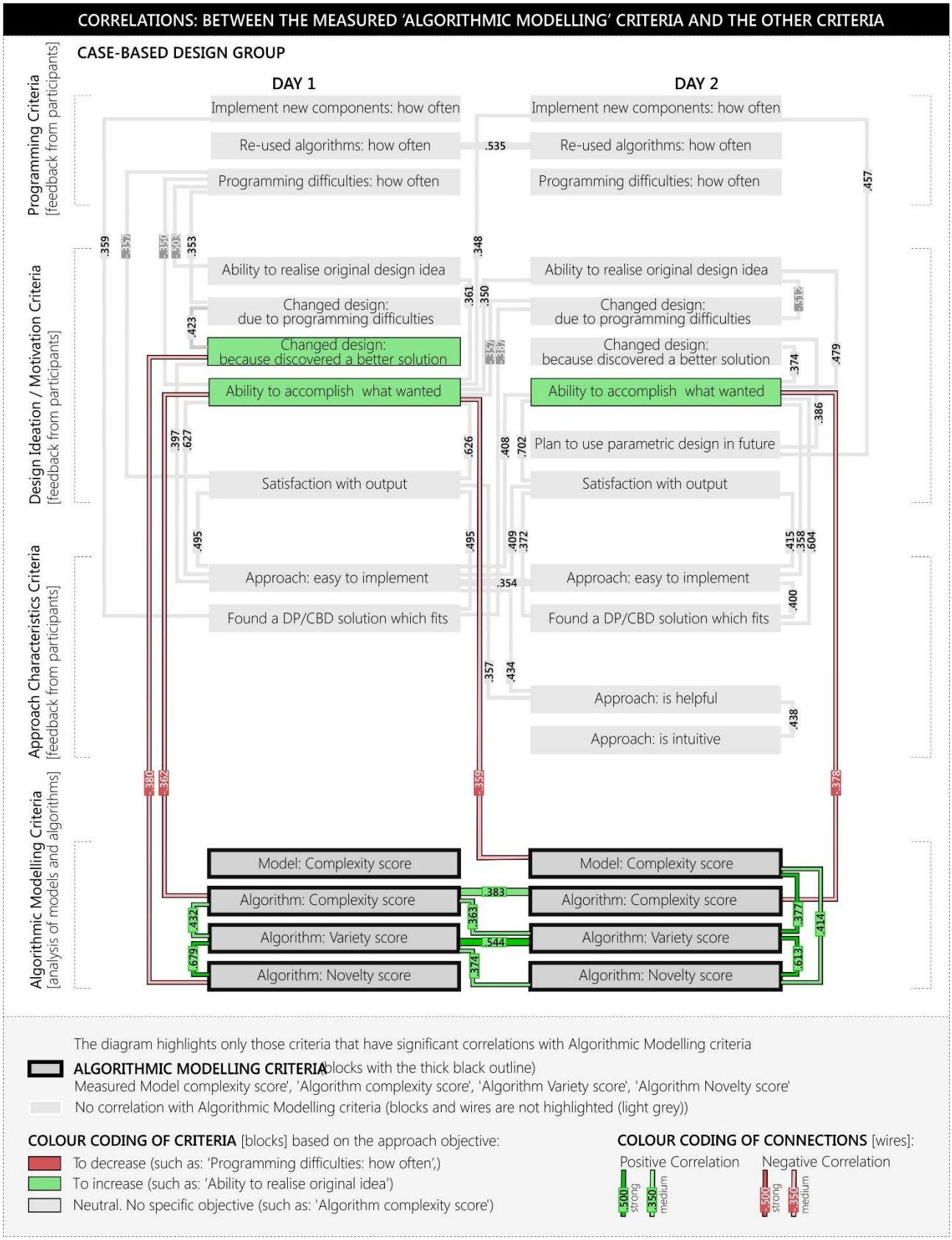


Exhibit 3.16. Case-Based Design group. Correlations between 'Model complexity score', 'Algorithm complexity score', 'Algorithm Variety score', 'Algorithm Novelty score' (algorithmic modelling performance) and the other criteria. [Also refer Appendix B, pages B71-B82]

As discussed in section 'The reuse of case-based algorithmic solutions induces more focused reasoning', the CBD group tend to be more committed to a particular design goal (more focused reasoning) then the other test groups. Participants using the Case-Based Design approach were less interested in experimentation with the design models or the exploration of the alternative options, and more focused on the realisation of the original design concepts. Observations indicate that participants of the CBD group were not particularly interested in creating a more developed (complex) design model. Exhibit 3.16 illustrates that there is a reciprocal relationship (negative correlation) between the *'Ability to accomplish what was wanted'* (on day 1) and *'Model complexity score'* (on day 2), r = -359. Exhibit 3.16 shows that on the second workshop day the average complexity of the output design models of the CBD group (12.7) was significantly lower (p-value between all groups equals 0.031), compared to the control group (13.9) (p-value CBD/NA equals 0.062) and compared to the Design Patterns group (14.1) (p-value CBD/DP equals 0.065). Even though, technically, the 0.062/0.065 are above the significance level (0.050), they are still very close to it. These p-values mean that statistically, there is at least 93% chance that the differences in results between the CBD and DP/NA groups have not happened by chance. Therefore, it seems likely that the CBD approach induces not only the development of the more simple programming solutions, but also more simple design outputs (Exhibits 3.15, 3.16).

Indexing issues in case-based design systems

Designers and architects who participated in this study often used metaphors and descriptive attributes when describing their algorithmic designs and when applying key words (indexes/tags/labels) for their

models, concepts and programming algorithms (Exhibit 3.17). The issue with abstract indexes (metaphors and attributes) is that they are rarely repeated (or searched for) by others, due to participants' individual backgrounds and associations. For example the same design solution one participant would label as a 'Cloud', the second as a 'Blob', the third as a 'Smooth Curvilinear Surface'. When browsing a repository of programming solutions these people are likely to search for some specific indexes, which express their own understanding and associations with the design characteristics, which may not match the indexes assigned by others. As a result, the retrieval of a case is likely to be unsuccessful. Keywords that work only for few people (limited population with matching associations) are not particularly effective key words. The main function of indexes in Case-Based Design systems is to provide a mechanism to navigate through the data-base of solutions, to identify and retrieve cases that a designer can potentially reuse.

Indexing (tagging/labelling) in algorithmic design can refer to various aspects of the solution, they can be: contextual indexes, visual indexes, association (metaphors/emotions) indexes, conceptual indexes, indexes describing the output geometry (forms) or programming solutions. Effective indexing is a very challenging task, because one has to predefine the features that will be relevant and helpful for future reuse and that will be understood and searched by others (Maher, de Silva Garza, 1997). In practice, even when a Case-Base contains a set of suitable algorithmic solutions for a particular design problem, there is no guarantee that any user can easily find and retrieve the appropriate cases, due to the mismatch in the thinking patterns of a person who applied the indexes and a person who searches for them. It is likely that this issue is going to be more relevant for the large scale case-bases. That is why

indexing and finding the cases is one of the main difficulties of designing a Case-Based system and using a CBD approach (Zimring, 1995).

To investigate how designers and architects tend to label their algorithmic solutions in architecture, on each day of the workshops all 126 participants of this study were asked to describe their designs by writing their own key words (applying indexes for their designs). Participants were asked to address three different aspects of their design solutions (use three categories of key words): 1) key words describing the form and geometry of their output design model; 2) association key words, describing design with abstractions, metaphors and attributes; 3) algorithmic modelling key words, describing programming solution. The key words in each category then were analysed and sorted into categories of most re-occurring indexes, which were (See Methodology Section):

- Geometry: Standards/Primitives (x example: point, circle, polygon, line etc.)
- Non-Standard geometry (index example: spiral, curves, surface etc.)
- Metaphors/Abstractions (index example: atom, ripples, wave etc.)
- Descriptive Attributes (index example: sharp, spiky, smooth, twisting, etc.)
- Programming Commands and Components (index example: divide surface, project, loft, extrude, rotate etc.)

Results of this investigation show that when describing **form and geometry of the output models**, designers mostly use 'geometry' related indexes (**53%** (276 key words)) or 'metaphor/descriptive attributes' related indexes (**40%** (209 key words)). 'Programming' related indexes were rarely used when reasoning about shape and geometry of a generated model (**7%** (39 key words)) (Exhibit 3.17).

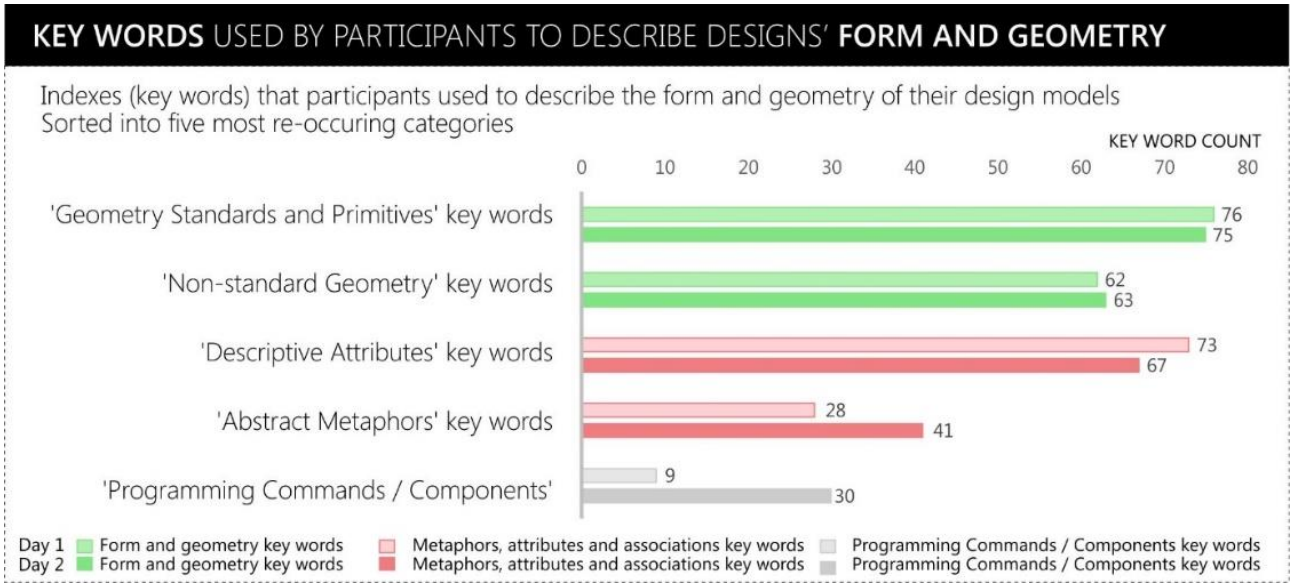


Exhibit 3.17. Indexing form and geometry of designs, (all groups) day 1/day 2 key words count. [Also refer Appendix B, pages B83-B84]

When using association key words, describing design with abstractions, metaphors and attributes designers and architects used predominantly 'metaphors or descriptive characteristics' (90% (423 key words)) and rarely referred to 'geometry' (4.5% (21 key words)) or programming (5.5% (25 key words)) (Exhibit 3.18). It seems likely that architects and designers tend to think (reason) about their design solutions with abstraction and it is relatively easy for them to describe their designs with associations, metaphors and characteristic attributes. However the major part of these abstract key words are not universal, due to individual experiences and backgrounds of participants. Many of the key words that participants used as associations (metaphors) seem unlikely to be considered the most helpful or effective attributes (indexes) for future reuse. For example: 'aesthetics', 'light rhythm', 'jittery', 'organic', 'slumping', 'drawn' etc. These key words might work for some people, and not work for others. Due to the differences in their personal experiences and associations, individual designers 'see' features in a design solution differently. These results seems to confirm that it is very hard to define the

characteristics and distinctive attributes representing a design solution (case) universally (Dave, 1994). This study shows that finding universal 'abstract' attributes is especially difficult, because even though designers tend to use a lot of metaphors and attributes when describing their designs, these descriptions are often too individual and far from being universal (Exhibits 3.18, 3.19).

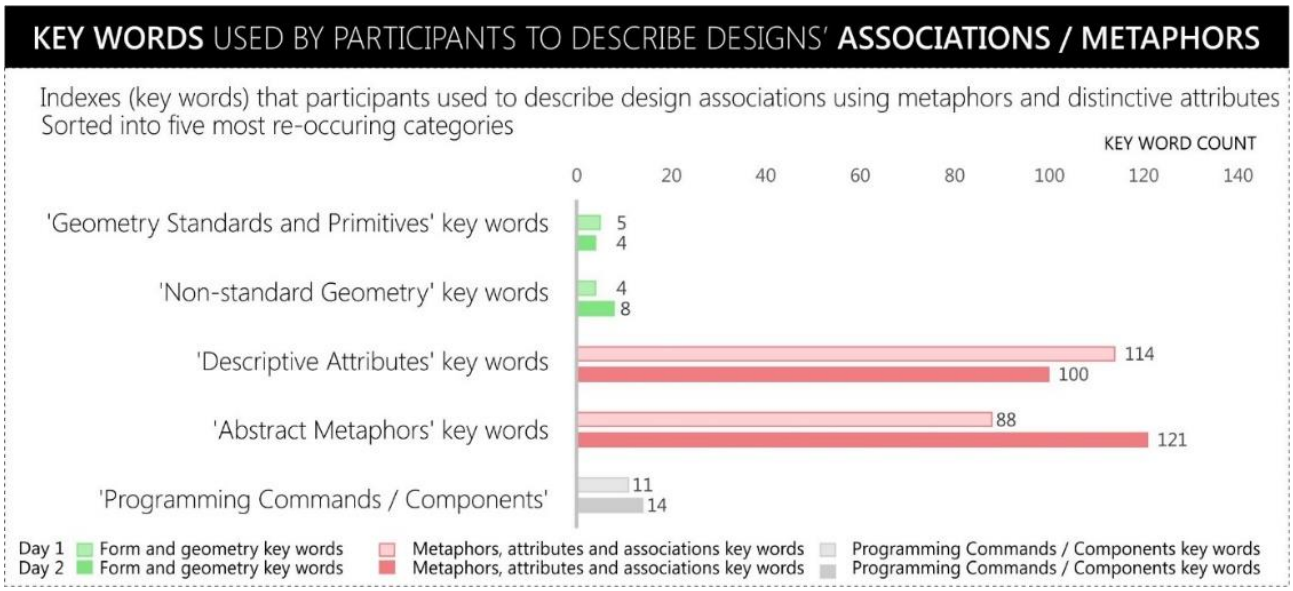


Exhibit 3.18 Indexing design associations using metaphors and distinctive attributes, (all groups) day 1/day 2 key words count. [Also refer Appendix B, pages B83-B84]

Participants used 704 key words describing their design with abstract metaphors, attributes and associations, which is 40% of all the key words (indexes). 380 (21%) of the key words were related to 'geometry' of output models and 686 (39%) were related to parametric algorithms (programming solutions) (Exhibit 3.18, 3.19). Out of all 1770 key words used by participants to describe different aspects of their designs 30 key words were repeated more than three times (counted for cases when different participants used the same index (key word) to describe their design solution). Out of these 30 top repeated keywords only 6 were related to 'abstract' design features. This happened because the majority

of metaphors and associations (abstract key words) were individual, whereas the 'form and geometry' (10 out of 30 top repeated key words) and programming related key words (14 out of 30 top repeated key words) were more universal (Exhibit 3.19).

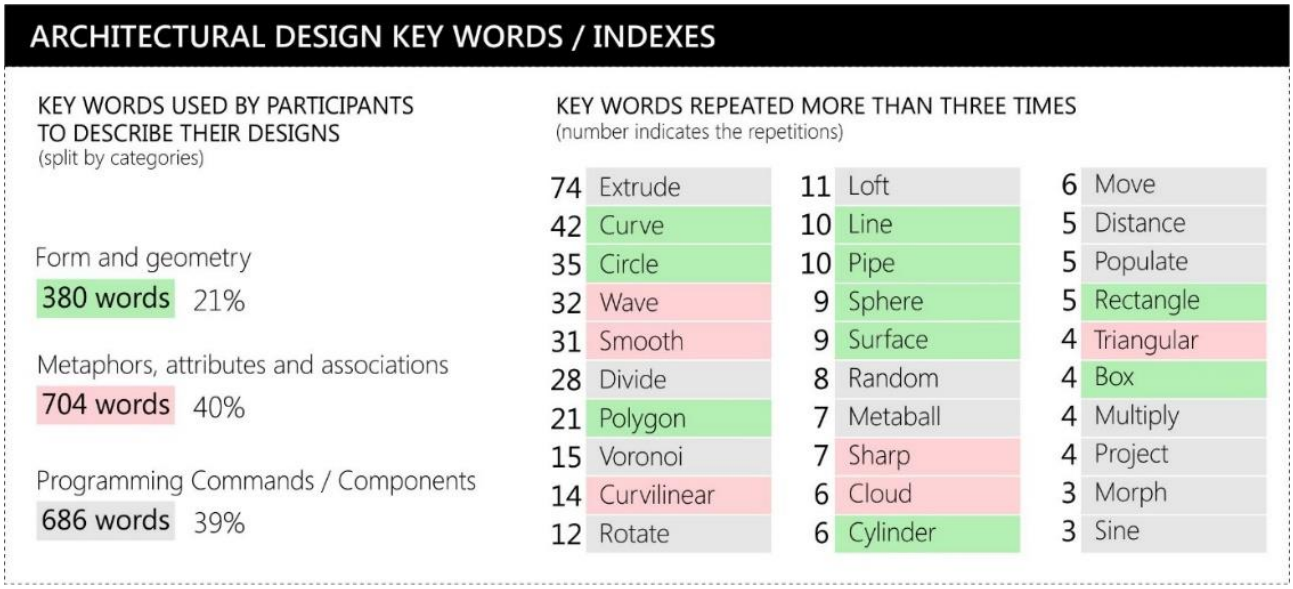


Exhibit 3.19. Key words used to describe parametric designs. Indexing in Case-Based Design, [Also refer Appendix B, pages B83-B84]

As a part of Case-Based Design approach evaluation the CBD group participants were asked to suggest how the online repository (used to test the reuse of case-based algorithmic solutions in architecture) could be improved. It was often suggested by participants that in addition to having the animated images of output geometry and the mechanism to search and retrieve design cases (programming solutions), based on specific indexes, a CBD system should have an established (pre-defined generalised) set of categories or *'groupings'*. The suggestion to *'potentially split cases into generalised categories'* was explained by one of the participants using the following argument: '*with such a wealth of information on the screen and even in refined searches it can be hard to remember what you are trying to find/looking for.'* The organisation

(classification and indexing) of cases is one of the main challenges of developing a Case-Based Design system (Dave, 1994). There are various aspects according to which, algorithmic design cases in architecture can be potentially sorted into categories. For example, it could be:

- design problem (what a particular design solution is trying to achieve/goals/objectives);
- design context (when and why this particular solution is relevant/conditions/limitations /scope),
- design output (what a resulting solution produces as output/building or structure typology/description of forms and geometry);
- programming solution (how a design problem is solved/logic of the algorithm/algorithmic solution);

All of these aspects of algorithmic solutions can help users of a CBD system to navigate through a database of cases in order to find and retrieve suitable solutions. Experience with the development and the feedback from participants who used a Case-Based Design system (testing the CBD approach in this study) shows that some of the potential categories can be more useful than the others. The preliminary results of the key words (indexing) investigation show that indexes, describing geometry features of an output model ('design output' indexes *such as: lines, curves, circles, polygons, pipes etc.*), can be useful but only to a certain extent. Practice shows that in algorithmic modelling it is often relatively easy to change the type of output (generated) geometry by minimal alterations to the input parameters or replacement of some programming components in the algorithm. For example, in an algorithm populating circles on a grid of points (or subdividing a surface into panels), a modeller (user) can change only one component to switch from circles to polygons, or to spheres or boxes. Similarly, it is often relatively easy to

change an algorithm from using lines to using polylines, or from polylines to curves, and so on. Therefore, despite their popularity, the use of geometrical features tags in some cases is not the most effective way to index (categorise) an algorithmic solution.

Indexes describing a **'design problem'** (*such for example as: tower, canopy, pavilion, urban furniture etc.*), as practice shows, also can be rather limiting (and therefore inefficient). For example, it was observed that some participants dismissed a potentially fitting reusable algorithm just because the index stated that it was a *'table'* and they needed to create a *'tower'*. The same programming logics can potentially be reused (applied) to model large architectural objects, or to design urban furniture, or to create fine jewellery items. This makes it possible that in practice an algorithm can be reused and applied to a variety of design problems: it is often only a matter of scale and material affordances. The **'design context'** indexes, describing the conditions, limitations, and scope of a design problem (or solution) were only partially addressed by this study as the CBD system (used to test the Case-Based Design approach) contained mainly simple algorithms which were applicable to a wide range of design contexts (See Methodology Section). Even though this 'design context' category was outside the scope of this study, this generalised case category, identifying the features of '*when and why each particular solution is relevant',* can potentially be useful. It should be further investigated by the studies dealing with more complex design cases.

Observations and the feedback from participants using the CBD approach show that the categorisation principles based on the aspects of the **'programming solutions'** seem to be among the most promising ones (used in the context of architectural algorithmic design). Indexing based on the aspects of a programming solution refers to the features of a

programming algorithm: what is the program or how exactly a design problem is solved. Exhibit 3.19 illustrates that key words describing programming commands and components (*such as divide, rotate, move, project, morph etc.*) were often used and repeated by other participants. The vocabulary (range of indexes) of programing commands and components is significantly narrower and clearly defined especially compared to the use of associations, descriptive attributes and metaphor indexes. Out of the 30 most repeated key words 14 relate to the features of a programming solution. It should be noted that terms related to programming and algorithmic modelling are not universal for all modelling platforms (software). However these types of indexes are likely to be effective when solutions are written using the same programming language.

When describing the features of their programming solutions participants mostly used the key words referring to programming commands and components (80%) rarely using references to 'geometry' (11%) or association indexes (attributes and metaphors) (9%) (Exhibit 3.20).

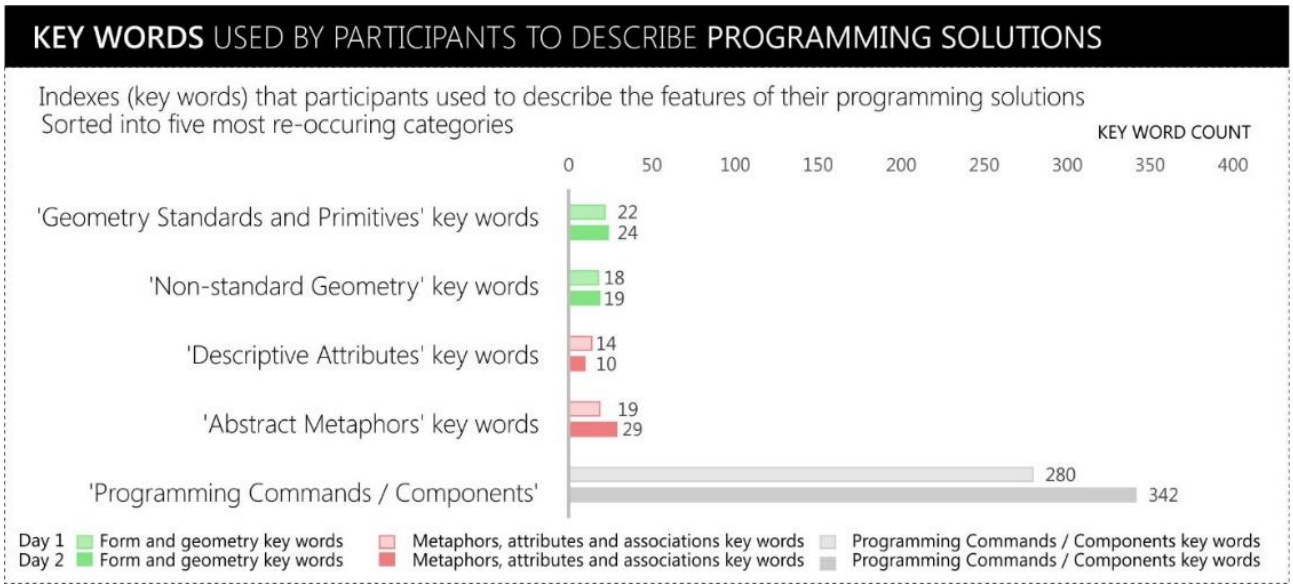


Exhibit 3.20. Indexing programming solutions/algorithmic modelling (all groups) day 1/day 2 key words count.

[Also refer Appendix B, pages B83-B84]

It was observed that those participants who knew *what* they wanted a program to do, were able to more easily find a fitting algorithm using indexes referring to programming commands. For example, when designers wanted to create a pavilion *surface* made of rotating panels, they could search for such indexes as: *'divide', 'surface divide', 'sub-surfaces', 'rotate'* etc., and there they were likely to find what they were looking for. However, in order to know which indexes to use, they had to have an understanding (knowledge) that a surface can be divided into sub-surfaces. Some participants reported that they could easily and effectively use the index search related to programming, while the others had difficulties with it. Most of index search difficulties occurred on the first day of the workshop when participants were still not too familiar and confident with the use of Rhino and Grasshopper and their modelling logic and commands. Here is how some of the CBD group participants expressed their issues with the index search:

- *'I struggled to know what key words to type since I do not use Rhino and Grasshopper and the relevant jargon.'*
- *'I found myself unsure of what key words to search for when using the search tool.'*
- *'Seems hard to connect visual ideas with word commands. Often I know what I want to achieve but do not know how to achieve it!'*

It seems unlikely to expect that designers and architects (who are amateur programming users) will know exactly which particular programming command or component index they need to use when searching through the case-base of algorithmic designs. The more experienced designers get the easier it is for them to identify the relevant key words (indexes) describing the reusable solutions, which can

potentially help them to translate their design idea (concept) into a programming algorithm.

In the fields of architecture and design, the information is often interpreted not textually, but visually in the form of diagrams and images. When using the CBD system participants often relied on the case related images provided (See Adaptation of the CBD approach Section for more detail). It was observed that after locating (narrowing down) a set of solutions using the textual indexes, designers preferred to rely on visual information (animated images of models). These images somewhat helped to overcome the lack of knowledge in programming terminology. When asked *'Which was the most helpful way to find information in the Case-Base of algorithmic solutions?'* participants of the CBD group reported (choosing on three options):

- Key words (10%)
- Visual diagrams/Images (38.2%)
- A combination of keywords and visual diagrams (51%)

This further proves the point that in architecture and design the visual representation of cases is an important indexing of cases (or, in some cases, is even more important). Visual representation of cases gave participants an opportunity to get the general understanding of what each algorithmic solution is producing as output and how the design model responds to the changes in parameters (which was possible due to the fact that the images were animated). However, the visual search (evaluation of cases) is only effective when there is a reasonable amount of cases displayed. It is likely to work for a dozen or a couple of dozen cases, but it seems unrealistic to expect that the user of a CBD system will be able to visually scan through hundreds or thousands of cases. The

observations show that that the visual evaluation of cases often happens as the method for final selection after the preliminary index search is done.

Generalised/abstract indexes and groupings (classification of cases) are important, because (due to the specifics of the human cognitive model) knowledge in our memory exists as both generalisation (abstraction) and as a collection of specific cases (solutions) (Heylighen, Neuckermans, 2001). The pre-defined classification of cases can help designers not only to narrow down the range of specific relevant cases, but also to help them to understand what they should look for, so they can effectively navigate through the database. The issue with the index search is that, in theory, the search/finding a case in a Case-Base suggests that designers already know that they are looking for. This implies that a design problem: '*What I want to do and how I want to do it*' is fully defined. However, in practice defining the problem (and therefore knowing what key words to search for) is actually a part of a design process. That is one of the reasons why designers often find it very difficult to clearly identify the relevant search indexes (Maher, de Silva Garza, 1997). In design, especially in conceptual design, a design problem is a task (algorithm) without a clearly defined specification, because a part of the problem is to identify what the problem is (Domeshek, Kolodner, 1992). Nevertheless, it is possible to assist designers in their search by providing visual information, clear indexes and (as this study suggests) generalised categories. As suggested by the observations and the feedback from participants there can be several strategies of how to improve the future algorithmic case-base systems for architectural design:

1) A dictionary of indexes – to help user navigate through the repository;

2) A search engine that keeps track/analyses the relationship between the indexes (key words) and can suggest solutions that are associated with each index. For example, when a user searches for an abstract (metaphor) index, such as 'cloud', and cannot find a fitting solution, a program will suggest solutions with 'similar' (related or synonymous) indexes, such as 'swarm' or 'cluster'. However, the implementation of this strategy can potentially be rather complicated. This study shows that the abstract indexes (key words) are not universal and substantially vary from person to person. That is why the 'similar index' suggestions that will work for one user could be absolutely useless for the other person.

3) The generalised categories that can be related to algorithmic modelling (programming solutions). These algorithmic modelling categories can potentially be based on Patterns for Parametric Design, developed by Robert Woodbury (2010), as these abstract (generalised solutions) proved to be an effective method of explaining and utilising the principles and logic of algorithmic modelling in architecture (See The Reuse of Abstract Solution section). It is planned to continue this study in future (and explore/test this strategy of using the Design Patterns as a grouping principle for a Case-Base of algorithmic solutions.

3.4 Comparison between reuse approaches: abstraction versus case-based

Effect of the approaches on the design thinking

This research tested whether the reuse of knowledge (tested by the reuse of abstract and case-based algorithmic solutions) can help designers and architects overcome barriers associated with programming and can improve algorithmic modelling performance. Compared to the control

group participants (No Approach), participants in both abstract and case-based reuse approach groups demonstrated improved performance. The differences in results were statistically significant (at 95% certainty level), including the ways designers' think and perform; and in what they ultimately produce. One of the most statistically significant differences is the major shift in the design objectives, caused by the use of approaches. The differences in objectives manifest themselves when designers gain more experience in algorithmic modelling. This can be seen in Exhibit 3.21. It illustrates the measured differences in the design ideation criteria between the abstract and the case-based study approaches on each day of the workshops. For three of these five criteria the differences were statistically significant: for these the p-value is highlighted in black, not grey, and most of these differences showed themselves to be statistically significant on day 2 of the workshops. Interpreting the measured responses, we can see that those designers who reused abstract solutions (the Design Patterns group) were more focused on experimenting with parameters (Exhibit 3.21).

Design Objective: 'To experiment with parameters'
Day 2: NA **12%**, DP **46.7%,** CBD **8.5%,**
(p-value All groups = **0.000,** p-value DP/CBD = **0.000,** p-value DP/NA = **0.006,** p-value CBD/NA = 0.463)
Design Objective 'To achieve what I originally sketched'
Day 2: NA **48%**, DP **60%,** CBD **80.8%,** p-value All groups = **0.012,** p-value DP/CBD = **0.045,** p-value DP/NA = 0.268**,** p-value CBD/NA = **0.005.**

Participants of both approach groups were much more likely to explore algorithmic form-making and to try out new programming logics

compared to the participants of the control group (No Approach (NA)). It should be noted that the group using the Case-Based Design approach was also more invested in the investigation of the capacity of algorithmic modelling (46.8%) compared to the control group (24.4%); however, the DP group showed the biggest interest *'To explore algorithmic form-making'* (63.3%) (Exhibit 3.21). Those who reused algorithmic solutions from specific design cases (Case-Based Design group) were more committed to realise the originally sketched design ideas and were less interested in explorations and experimentations (Exhibit 3.21).

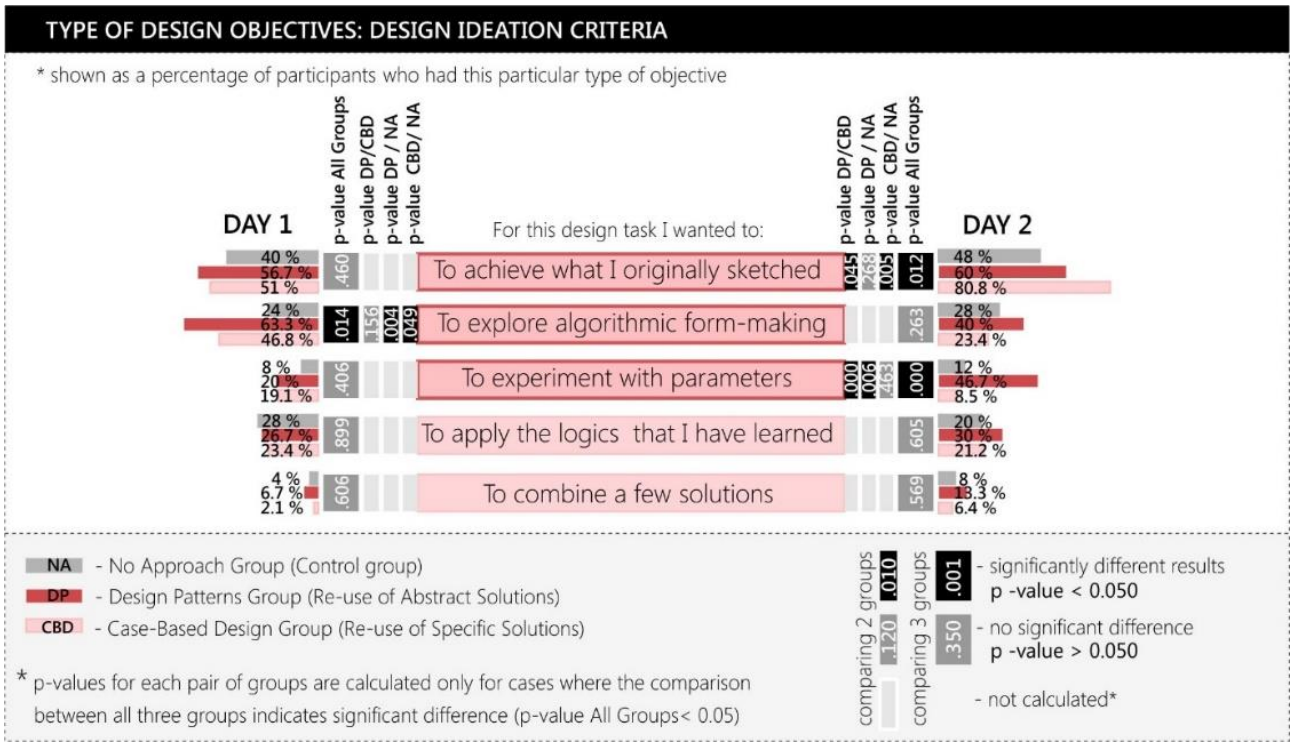


Exhibit 3.21: Typology and distribution of design objectives. [Also refer Appendix B, pages B64]

The shift in design objectives and modelling priorities appeared to have a significant influence on the design process and, as a result, on the final design output. The test group who reused abstract solutions (DP group) were less committed to a particular design goal. This is illustrated in Exhibit 3.22 by two designs from the DP group where the two participants reported a score of 2 (out of a maximum 5) on their ability to

model their original design idea. The figure shows the original hand sketch and the output model from their Day 2 DP workshop. These two participants also reported a 4 (out of 5 again) on their ability to find a Design Pattern that fitted their idea and a 4 on their ability to accomplish what they wanted. As shown in Exhibit 3.21, participants in this group were more likely to experiment and try alternative options of programming logic and components. This in turn has apparently influenced the way designers created their programming algorithms. Analysis of the programming algorithms showed that those who reused abstractions had a significantly greater explored solution space of the algorithms, compared to the group who reused specific design solutions.

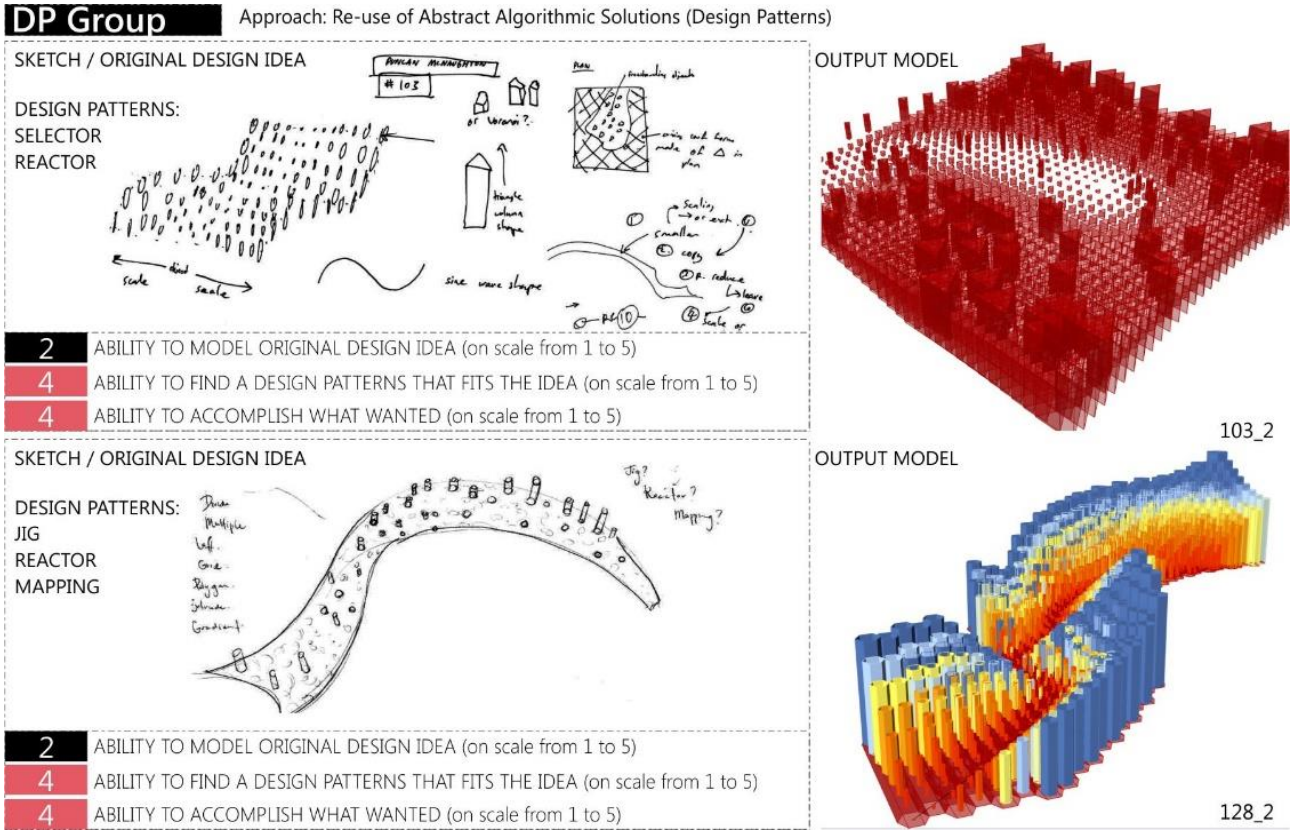


Exhibit 3.22: Examples of sketches (original design ideas) and corresponding output models, designed by participants using Design Patterns. Typical cases where designers have significantly changed their original idea and still reported that they were able to find a Design Pattern(s) that fit and were able to accomplish what they wanted.

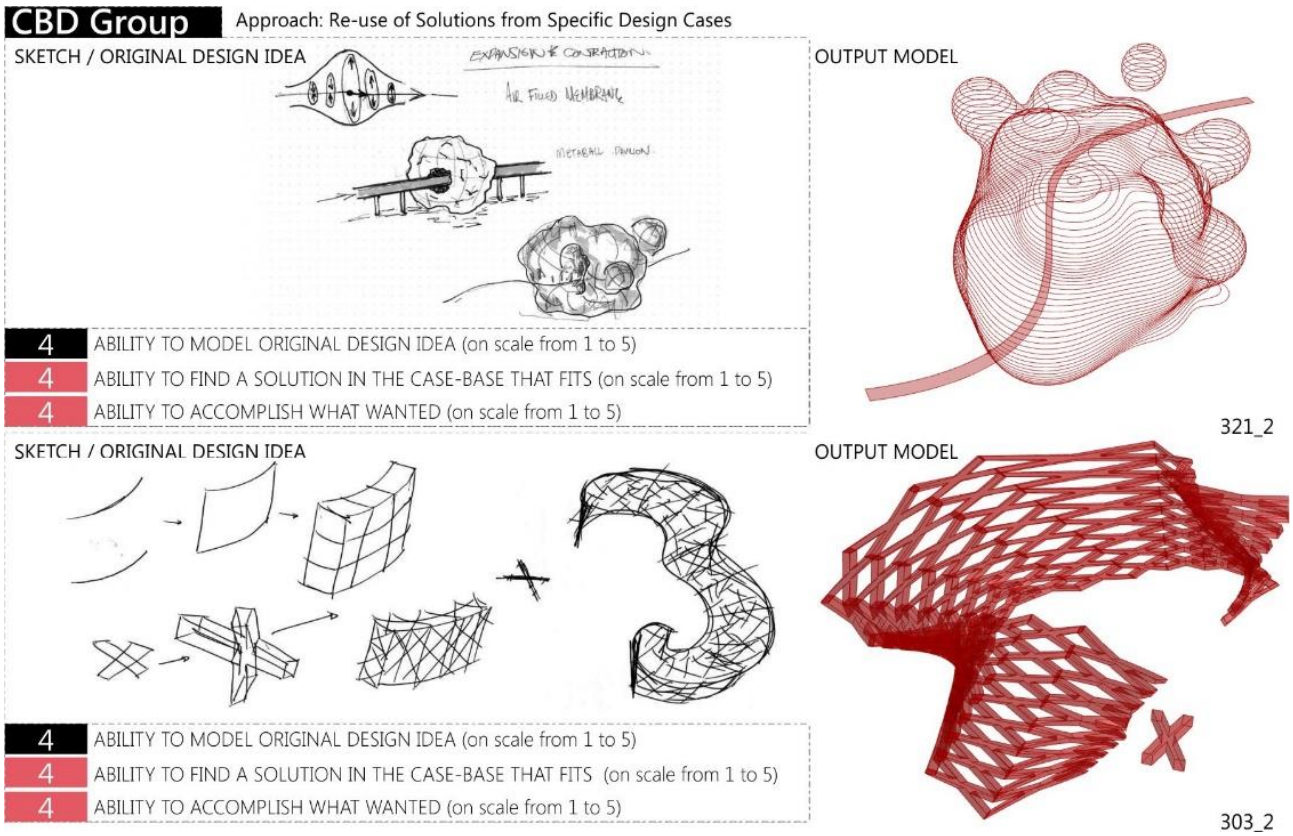


Exhibit 3.23: Examples of sketches (original design ideas) and corresponding output models, designed by participants using Case-Based Design approach. Typical cases where designer managed to develop an output model that was close to their original idea and reported that they were able to find a Design Pattern(s) that fit and were able to accomplish what they wanted. [Also refer Appendix B, pages B64]

Statistical testing indicates that designers who used case-based reasoning while developing their algorithmic solutions tended to focus on modelling a particular design outcome. This is shown in Exhibit 3.22 by two designs from the Case Based Design (CBD) group where the two participants reported a score of 4 (out of a maximum 5) on their ability to model their original design idea. As a group, the analysis in Exhibit 3.22 suggests they were less interested in exploring different programming options and new strategies. Instead, those who used CBD tended to implement components that they already knew (and which were explained during the workshop tutorials). When browsing the online case-base, these workshop participants predominantly used key words associated

with already familiar (used in the past) programming components, rather than using abstract key words, thus reducing the likelihood of developing alternative programming solutions.

The evidence suggests that use of case-based reasoning in parametric design will most likely decrease the variety of programming components used to create parametric models. Designers who use CBD also tended to produce less novel (more typical) programming solutions. However, it should be noted, that while the CBD group did use a substantially smaller range of programming components and developed less novel programming solutions compared to both DP and control groups, they reported higher overall satisfaction with the design model and their ability to accomplish their design objectives than with the abstract approach (Exhibits 3.22-3.23). These conclusions further confirm the findings reported in the earlier research on the implementation of CBD tools in design, stating that:

'The major disadvantage of the case-based method is that the solution space is not fully explored and as a result, there is no guarantee of an optimal solution' (Kolodner, 1993) (See Reuse of Case-Based Solutions Section).

Change in modelling speed/model complexity

The shift in design strategies caused by the use of abstract and case-based algorithmic solutions had a significant effect on the complexity of produced designs. Designers who reused specific programming solutions (CBD group) were likely to develop less complex output models, compared

to both the abstract (DP) and No Approach groups (Exhibit 3.24, 3.25). It would appear that the 'abstract' group's greater interest in experimenting with forms and parameters produces designs less restrained by the limitations of the original design concept. Four example designs from this abstract group are shown in Exhibit 3.24. The suffix to the participant ID number shows that three of these are from day 1 of the workshop, and one from day 2. The score highlighted in black under each design has been developed as a means of systematically ranking the complexity of the programming algorithm. All four of the participants whose work is illustrated reported high (5 out of 5) satisfaction with their output model, but were far less satisfied with their ability to model their original idea (a score of 2 or 3 out of 5).

Designers who reuse particular programming solutions, seem to be more focused on modelling a specific design outcome. Exhibit 3.25 shows four example outputs from this group laid out in the same manner as Exhibit 3.24. Two of the outputs are from Day 1 of the workshop and two from Day 2. The overall programming complexity of these examples is much lower than for the DP group in Exhibit 3.25. The four examples in each figure were selected to be clustered close to the average for each approach, but to all have a score of 5 on each workshop participant's satisfaction with the output model.

It is interesting that the No Approach workshop group were like the DP group in that they showed greater readiness than the CBD group to change their initial concepts, and to develop and experiment with their designs. The CBD group participants were more likely to try and develop a particular programming sequence, which would generate the form that they originally sketched, even though this might prove to be time-consuming.

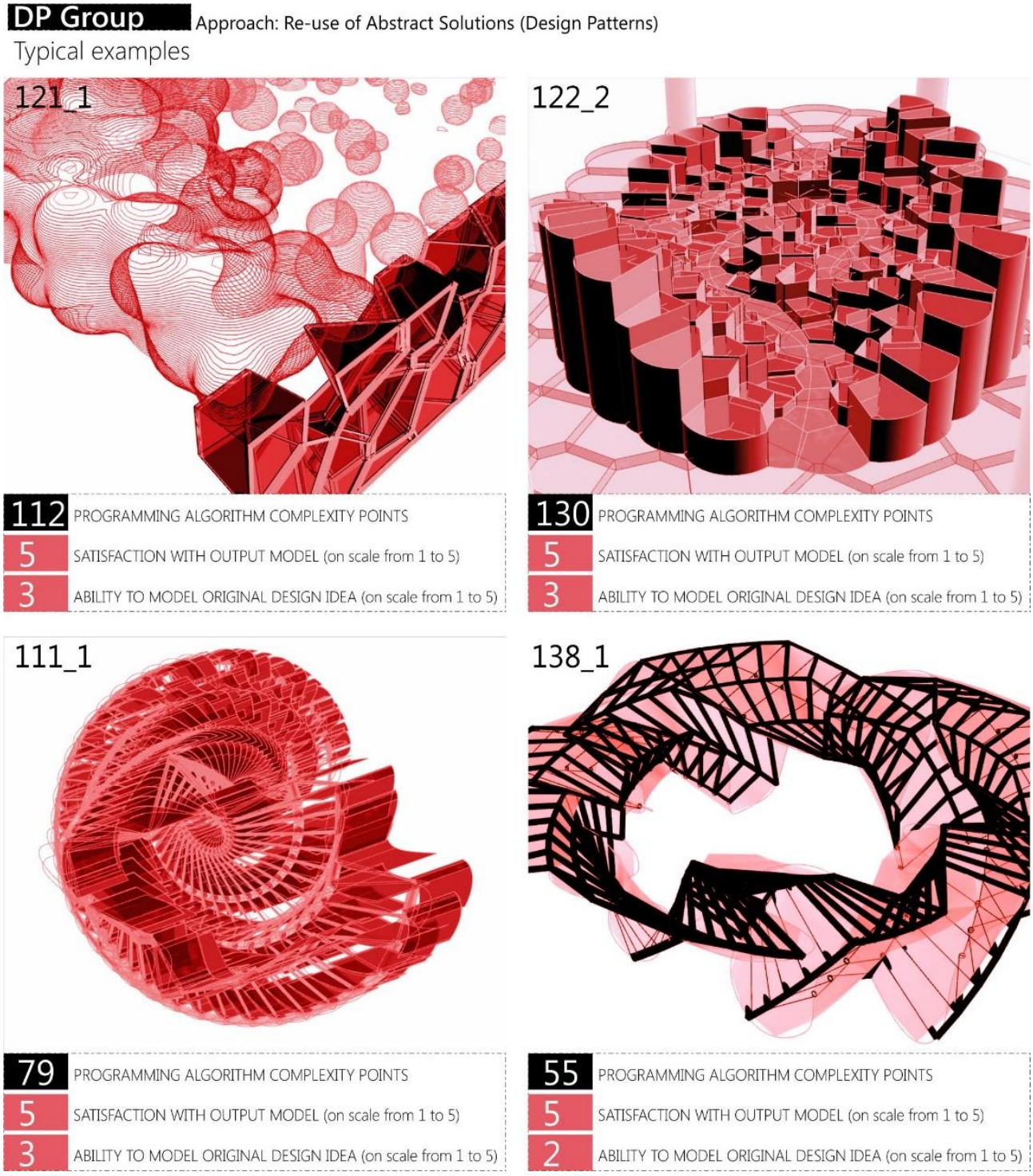


Exhibit 3.24. Examples of models, designed by participants, who used Design Patterns and were able to accomplish what they wanted; explored alternative design options; significantly changed the original idea; and developed more complex programming algorithms and output models. [Also refer Appendix B, pages B64]

There is likely another reason that the CBD group participants might be slower in modelling than the abstract and no-approach groups: it is

related to the time spent by users accessing the case-base examples looking for programming sequences that allow them to generate the form they originally sketched. Analysis of the screen recordings indicates that participants who reuse solutions from the case-base, tend to spend a considerable amount of time browsing the case-base and exploring various programming solutions. It was observed that designers rarely reuse the very first solution from the case-base which they chose to probe. Instead, they tend to compare several design options, before deciding which solution they actually want to reuse. Observation of the group which used case-based design shows that the search process for the most fitting specific solution can take a considerable amount of time, which inevitably slows down the overall speed of algorithmic modelling. Reuse of abstract solutions in this case has an advantage.

It seems likely that once designers and architects grasp the idea of a design pattern they do not have to re-learn it each time they implement it in a new design problem. Learning why and how to use a particular abstract solution (design pattern) is a one-time operation. In theory, when designers know a design pattern they might be expected to re-apply it to a new design task straight away. Designers who reuse specific solutions are likely to search the case-base of algorithms every time before they chose to reuse (copy/modify) (Woodbury, 2010). The 'modify' part of this copy/modify approach is very important as in most cases each reused solution has to be adapted to suit the new design context – to achieve the original sketch design outcome.

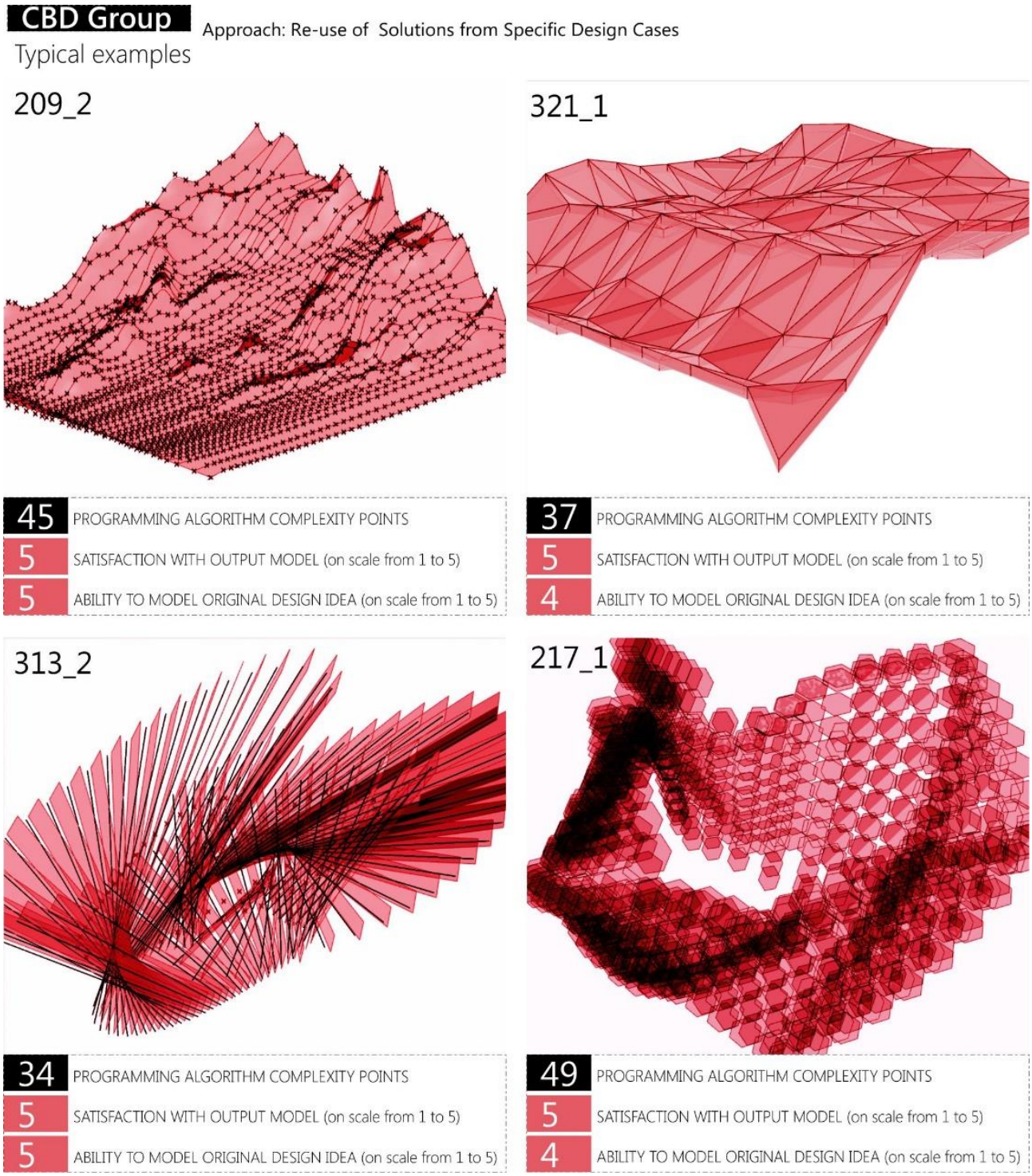


Exhibit 3.25. Examples of models, designed by participants, who used Case-Based Design approach and were able to accomplish what they wanted; managed to model the original idea; and developed more simple programming algorithms and output models. [Also refer Appendix B, pages B64]

Correlational analysis was used to study the reasoning of the designers in each group. Higher complexity levels of the output models

and of the programming algorithms are perceived positively by those who reused abstract solutions (Design Pattern (DP) group) (See 'The reuse of abstract solutions' section). The more complex the design models that DP participants produced, the higher their satisfaction with the output (correlation coefficient 0.463). Those DP designers, who managed to develop more complex programming algorithms also found the DP approach more helpful (correlation coefficient 0.417). Model and programming algorithm complexity are seen by these designers in a positive light.

In contrast to the abstract DP group, designers who reused algorithmic solutions from specific cases (CBD group) preferred to avoid complexity and tended to settle for the more simple programming algorithms. On both workshop days 'algorithm complexity' has a negative correlation (correlation coefficients -0.362/-0.378) with 'satisfaction with the design outcome'. When CBD group participants managed to come up with more simple programming solutions, they were apparently more satisfied with the outcome (See Reuse of Case-Based Solutions Section).

In summary, those who reuse specific solutions see complexity in a negative light, which is the exact opposite of what the group who reused abstract solutions tended to think.

Overcoming barriers associated with the use of programming

Many designers find it difficult to integrate algorithmic thinking and programming into the design process (Woodbury, 2010). Understanding and learning the programing framework syntax rules can be very

frustrating, particularly to novice users (Celani, Vaz, 2012). This study tested whether the reuse of abstract and specific algorithmic solutions can help designers and architects to overcome these barriers. The participant designers were asked to indicate the overall amount of difficulties that they had while developing their design assignments and also to specify what type of difficulty it was.

Analysis of their responses identified the five most common categories of difficulty:

- Idea-to-algorithm translation (*design barriers*, figuring out how to get from a sketched idea to a programming algorithm, which generates a model);
- Problems with specific components (*use barriers*, when participants knew which programming component they need, but struggled with how exactly to use it);
- Knowing what programming component to use and when (*selection barriers*);
- Logic Connections (*coordination barriers*, what is the correct sequence of programming logic, for example should 'vector' go before or after 'move');
- Valid Parameters, unexpected errors (*use and understanding barriers*, for example, incorrect inputs or domains of numbers). (Ko, Myers and Aung, 2004) (See Barriers associated with the use of algorithmic tools in architecture section for more detail)

The diagram in Exhibit 3.26 illustrates the degree to which all five of these parameters were a problem for each approach. The length of the pairs of bars either side of the central list of difficulties represents the percentage of workshop participants who reported each difficulty. The most common difficulty for people learning to use algorithmic modelling

tools is immediately clear: 'Idea-to-Algorithm Translation' was reported as a problem for 43-60% of workshop participants.

The second most common type of difficulty was problems with actual implementation of a particular programming component (Exhibit 3.26): 21-49% of participants. The reuse of solutions from the case-base proved to be an effective approach to overcome these types of difficulties. There were significantly less problems with particular programming components in the CBD group, compared to both the DP and the control group. The difference in the average number of participants reporting difficulties in day 2 workshops was the only statistically significant difference observed on these particular criteria in the bottom (Type of Difficulty) of the Exhibit 3.26. The top (How Often) portion of Exhibit 3.26 shows an overall analysis of the number of programming difficulties encountered by workshop participants.

Assigning a score of 1 for no difficulties, a score of 2 for 1-3 difficulties and so on to a score of 5 for 10 or more difficulties produced the three bars to the right for 'No Approach', "Abstract Approach' and 'Case Based Design Approach'. The average score (number of difficulties) on day 1 and on day 2 is significantly less for the reuse of abstract solutions (Design Patterns) approach. Reuse of abstract solutions is therefore an effective method to help designers reduce difficulties associated with use of algorithmic modelling tools. The DP group participants had significantly less programming difficulties compared to both the CBD and No Approach groups. Despite this clear difference, it is worth remembering the case-based (CBD) approach did help to overcome certain types of difficulties.

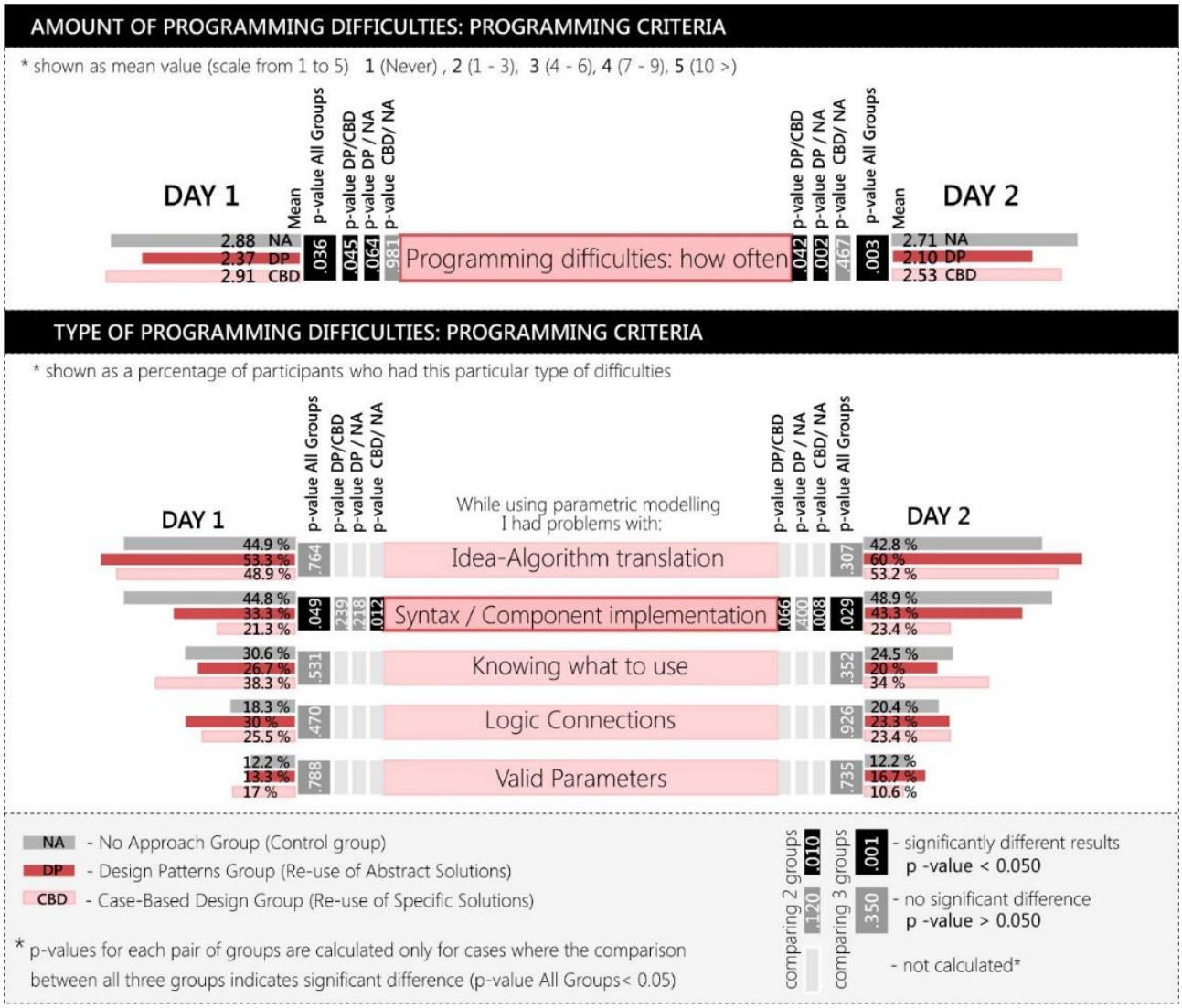


Exhibit 3.26: Overall amount of difficulties. Typology and distribution of programming difficulties. [Also refer Appendix B, pages B64]

As there were very few workshop participants with significant levels of experience with algorithmic modelling systems, it seems reasonable to conclude that in the initial stages of learning and using of these systems, the use of abstract solutions, such as Design Patterns, helps to reduce the overall amount of difficulties (See 'Expanding beyond the scope of this research' discussing design population: novices and experienced programmers). Abstractions help novices to better comprehend, in principle, 'when' and 'how' a design problem can be solved. However, in terms of initial impressions, rather than output produced, designers

themselves appear not to realise how helpful the use of abstractions (Design Patterns) is. When asked 'how easy to implement', 'helpful' and 'intuitive' each approach is, the Case-Based Design approach was identified by designers as significantly more intuitive, helpful and easy to use (For more details see 'The reuse of abstract solutions' and 'The reuse of case-based solutions' sections).

Summary of key findings

The primary observation to be made is that, when learning computational design methods, the use of a systematic approach to the reuse of algorithmic design solutions is more beneficial than having no approach.

In many aspects, such as for example the ability to overcome programming difficulties, the reuse of abstract (Design Patterns) solutions is more helpful than the reuse of solutions from a case-base (Case Based Design). The use of CBD proves to be mostly effective in overcoming difficulties associated with the implementation of specific programming components and commands.

The reuse of abstract solutions in algorithmic design helps to reduce the barriers that designers and architects have when they use algorithmic modelling systems and motivates designers and architects:

- to experiment more;
- to explore new programming solutions and commands;
- to produce algorithms and output models with higher levels of complexity.

The reuse of algorithmic solutions from specific cases (CBD), is an effective tool to reduce difficulties associated with the implementation of

specific programming components and commands. It is intuitive, helpful and easy to use; it promotes the development of more simple and less novel design solutions; and motivates designers:

- to focus on realising the initial design ideas;
- to be less invested in exploration of alternative solutions and experimentation with new programming logics.

Exhibit 3.27 illustrates the comparison of all the metrics (criteria), which were evaluated in this study through the analysis of the design models and programming algorithms, and using the feedback from participants.

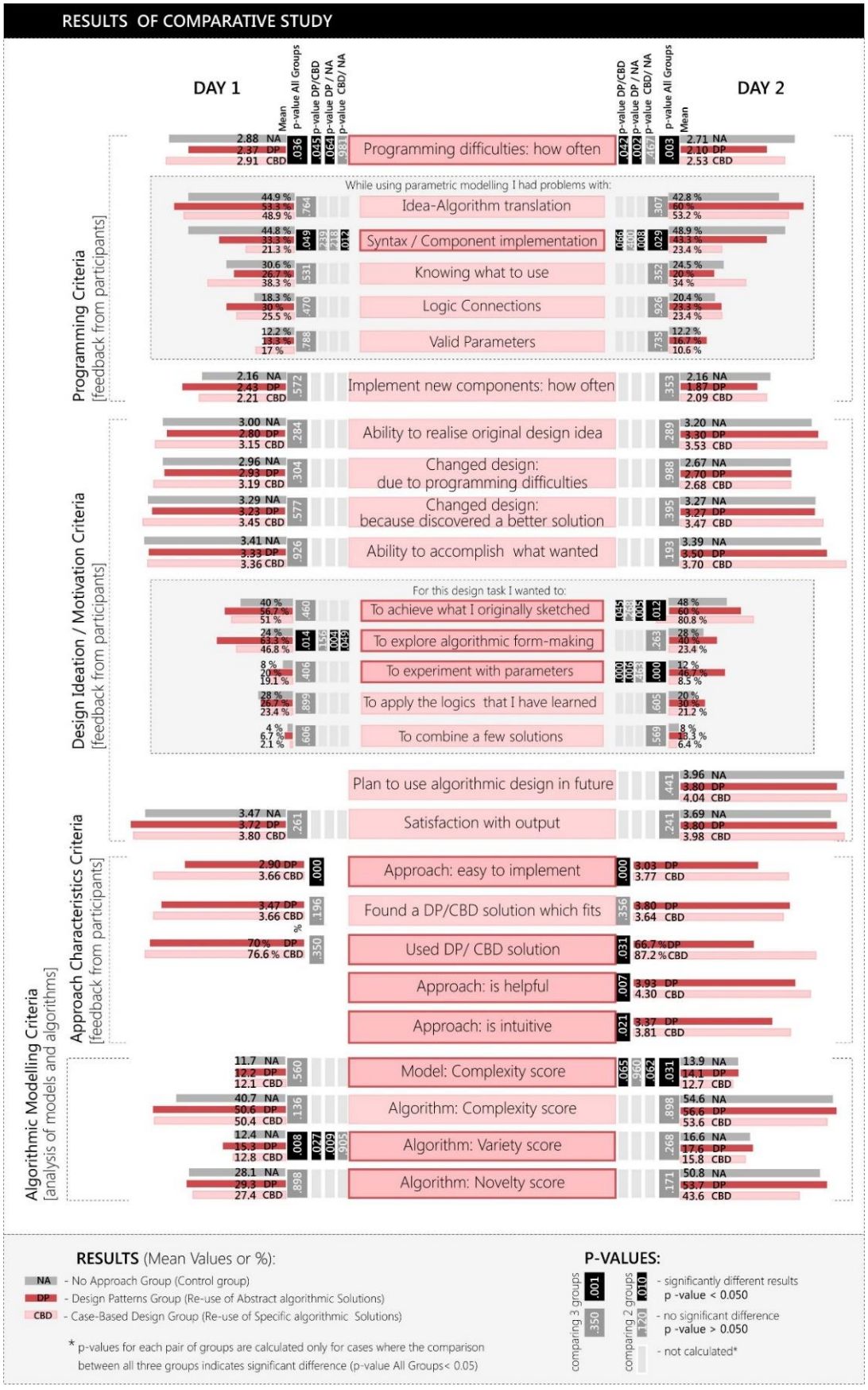


Exhibit 3.27: Results of comparative study (all criteria). [Also refer Appendix B, pages B63 – B84]

4. Expanding beyond the scope of this research

This thesis investigated the knowledge reuse as a design support method aiming to overcome programming challenges and help designers to adopt and to use algorithmic modelling tools more effectively. The study tested two alternative methods: Design Pattern approach and Case-Based approach. This investigation of ways to support learning and use of computation in architecture shows that the reuse of abstract and case-based algorithmic solutions helps designers overcome barriers associated with use of programming and improve their design performance (See 'The reuse of solutions as a method to support design' Section). Both of these knowledge reuse strategies are applicable for textual and visual programming environments. It was suggested that these approaches can be useful for a wide designer population, including both experienced and novice designers in programming. However, the framework of this study had a particular scope, such as using a visual programming environment and having a particular designer population.

This chapter discusses the boundaries of this study and talks about the future research aimed to expand beyond the current research scope. It outlines the potential of testing the Design Patterns and Case-Based Design approaches on a group of architects who are more advanced in

algorithmic design, and the potential of testing the DP and CBD approaches using textual programming languages. It also discusses the differences in performance between the male and female participants. Further this chapter will canvas the issues identified for the DP and CBD approaches. It is suggested that one of the ways to improve these knowledge reuse methods can be the development of a hybrid approach. This hybrid approach can incorporate the methods and techniques of both abstract and case-based solution reuse.

4.1 Design population: novices and experienced programmers

The target group of this study covered a wide design population, including architectural and design students as well as practicing professionals, who learn or routinely use algorithmic modelling systems in their design process. However, due to the constraints of the research scope, limited mainly by the availability of architects and designers (skilled in programming) who use algorithmic modelling in their professional everyday practice, this study focused on a learning environment, recruiting designers who are programming novices (See Design Population in Methodology section).

The findings and lessons of this study can be adopted and applied to the educational environments dealing with teaching and practical implementation of programming in architecture and design. For example, this research leads to the conclusion that the systematic use of algorithmic abstractions (Design Patterns) when learning (mastering) algorithmic design logic helps architects and designers to structure their computational thinking and subsequently helps to overcome barriers associated with the

implementation of programming. Learning and implementing algorithmic design through parametric abstractions (Design Patterns) helps to give a more profound understanding of the high-level (abstract) logic of programming processes. This understanding of abstract (high-level) logic seems to be most important especially in the initial stages of learning. The main challenges that algorithmic design imposes on architects and designers, is not that of acquiring programming skills but it is rather assimilating 'a mode of computational design thinking' (Menges, Ahlquist, 2011). The use of abstract patterns helps novices to adopt this new algorithmic thinking mode, explaining: when and why a particular programming logic can be used; and what 'in principle' an abstract algorithmic pattern can produce as output (Woodbury, 2010).

Research also highlights the weakness of this approach, related to the fact that patterns do not actually show 'how exactly' to solve a particular design problem. Because patterns are abstract solutions, they rather tell 'how in principle' a particular problem can be solved (giving generic guidelines instead of specific instructions). This research shows that the use of case-based reasoning can significantly reduce these implementation barriers, which are widely acknowledged as common for both among novice and more experienced programming users (Ko, Myers, Aung, 2004).

The question, which currently lays beyond the scope of this study, is: how effective the DP and CBD approaches can be when applied in the context of architects and designers experienced with coding skills. To go beyond the scope of this research requires a further study which tests the reuse of case-based and abstract algorithmic solutions (or a combination of those approaches) on more experienced programmers. This is the next planned focus of this research programme. The difference between these

advanced programming users and novice users is likely to be not only in distinct levels of coding skills, but also in the designers' ability to employ computational design thinking. It might be speculated that experienced programmers will not be as keen (flexible) to re-structure their well-established computational thinking mode, shifting from their own practically acquired algorithmic design constructs (abstractions) to patterns suggested by other people. However, it can also be reasoned that the reuse of high-level (abstract) solutions can be easier for advanced users, as they (unlike novices) are more skilled and usually know 'how' to solve/implement a particular programming algorithm.

Reuse of code (programming algorithms) is a common practice in software programming and to that extent algorithmic modelling in architecture should potentially benefit from algorithm reuse. To measure whether the reuse of a case-based programming algorithm is effective and worth using, it is necessary to test what would have been easier and faster to do: a) to reuse (copy/modify) an existing solution or, b) to create an algorithm from scratch. It can be assumed that for the more advanced algorithmic design users it might be easier to create an algorithm anew rather than spending time searching through the case-base and then modifying the original algorithm to fit the new design context. It can also be argued that more experienced users are usually dealing with more complex programming solutions that can be split into simpler subtasks. There is always a chance that there are existing solutions for some of these subtasks which can be recycled again and again. Therefore the reuse of algorithms can help to overcome the complexity of advanced algorithmic designs.

The analysis of experienced algorithmic designers' reactions to both Design Patterns and Case-Based Design approaches should examine

whether the trends detected amongst novice programmers persist. For example, would more experienced programmers who use the Design Pattern (DP) approach be showing more exploration and would those using the Case-Based Design (CBD) approach be more directly focused on realizing a single result reflecting original intentions? Moreover, would the DP approach still encourage satisfaction with complexity whereas CBD seems to discourage it? (See 'the Reuse of abstract parametric solutions' section for more detail).

4.2 Identified gender differences

Male and female participants showed similar results for most of algorithmic modelling performance criteria, which were identified and measured by this study. Results suggest that overall, participants of both genders performed evenly (statistically not significantly different) and had a similar response to the use of the DP and CBD approaches. 126 participants took part in this study. 55% of these participants were males (70) and 45% were females (56), with uniform distribution of genders in each test group. On average, male and female participants had a similar level of programming difficulties; similar ability to accomplish what was wanted; and both genders produces programming algorithms and models of similar levels of complexity. Only four out of thirty evaluated criteria were statistically different between the gender groups (Exhibit 4.1). Comparison between all male and female participants showed that statistically significant differences in results only occurred in: 'Algorithm Novelty score' and 'Ability to realise original idea' on day 1; and in: design objective 'To combine a few DP/CBD solutions' and 'Reuse of algorithms' on day 2 (Exhibit 4.1).

It was observed that in the initial stages of learning and using of algorithmic modelling system, male designers tend to explore more programming options compared to female participants. On the first day of the workshops male participants were keener to try new things and preferred to explore and test things on their own, rather than reuse existing solutions. Unlike the female participants, they initially tend to learn by 'trial and error', often using the 'guess and check' strategy. Comparison between the genders shows that on day 1 the average 'Novelty score' of programming algorithms is higher for male participants

Algorithm Novelty score (mean)

Day 1 Males 31. 4/Females 24.1, p-value = 0.017.

Day 2 Males 50. 7/Females 46.4, p-value = 0.335)

'I was able to realise my original design idea' (shown as mean values)

On a 5 point scale, from 1 Strongly Disagree to 5 Strongly Agree

Day 1 Males 3.20/Females 2.83, (p value = 0.047)

Day 2 Males 3.48/Females 3.24, (p value = 0.181)

However, the difference in novelty of explored solution space of the algorithms disappears, when female designers gain more confidence in programming (day 2) (Exhibit 4.1). On the second day of the workshop female participants started experimenting and exploring almost as much as male participants (no significant difference in results on day 2). It seems likely that in the initial stages this 'guess and check' approach to master a new algorithmic modelling software was rather effective, because on day 1 male participants had shown a higher ability to realise original idea. Again the difference in ability to model original design concept disappears on day 2 (Exhibit 4.1).

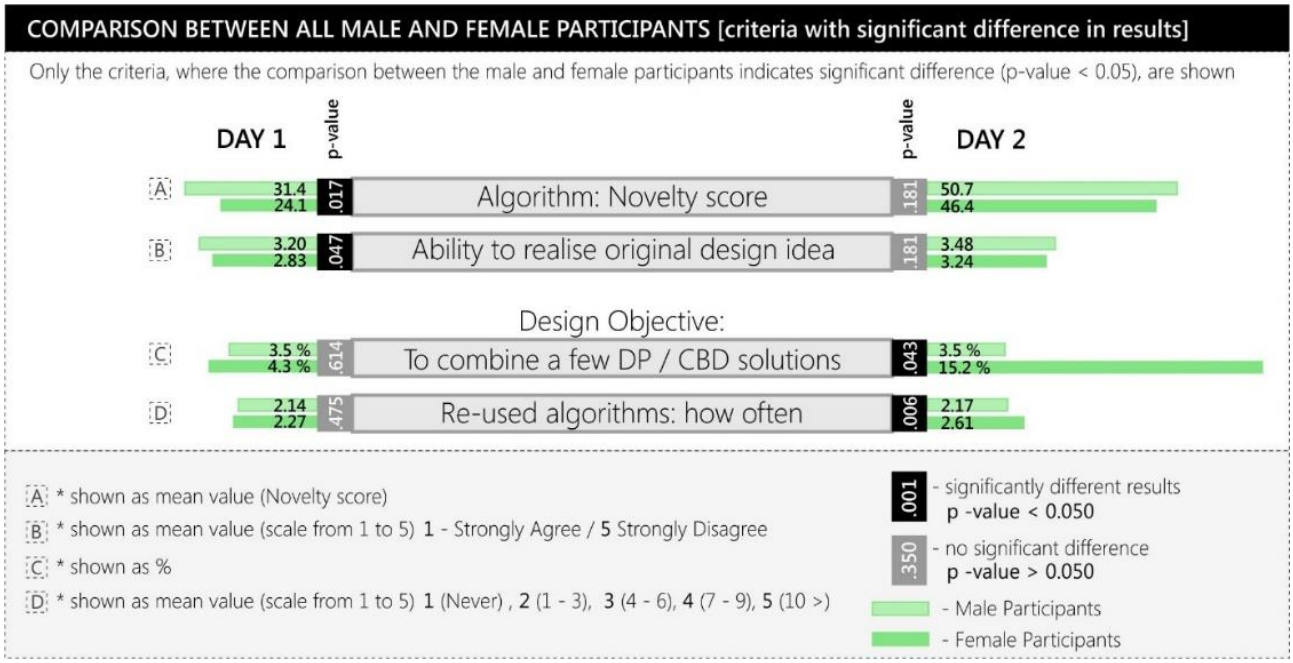


Exhibit 4.1. Comparison between the male and female participants, [Also refer Appendix B]

Statistical testing between genders also showed that female participants are more inclined to reuse existing programming algorithms, rather than search through the interface of a yet unfamiliar software by themselves and try to figure out how things can be done (significantly different on the second day of the workshop). The difference between the reported design objectives of male and female participants shows that on day 2 the objective: to combine several Design Patterns or Case-Based programming algorithms during the development of their design task, became significantly higher for female designers ('*To combine a few of DP/CBD solutions*': Males 3.5 %, Females 15.2 %, p-value= 0.043). This might suggest that on the second day of the workshop female participants were keener to engage the case-based reasoning and learn from existing solutions. It seem likely that as a result of this higher motivation to use case-based reasoning (learn from cases which worked for others in the past), the female participants have reused more programming algorithms, compared to male participants. Similar to design objectives the difference manifests itself on the second day of the workshop (Exhibit 4.1).

'Reused programming algorithms: how often'

On a 5 point scale with 1 Never, 2 (1-3 times), 3 (4 - 6 times), 4 (7-9 times), 5 (10 or more times): Males 2.17 (median = 2 (1-3 times), Females 2.61 (median = 3 (4 – 6 times)), p value = 0.006).

It is hard to speculate on the interpretation of these results. They might suggest that the case-based reasoning (and therefore Case-Based Design approach) can be a slightly more natural way for female designers to master the use algorithmic design systems. Male designers on the other hand seem to be more inclined to explore things on their own, applying the 'trial and error' approach (at least on the initial stages of learning). However overall, there is no indication that the CBD approach (and case-based reasoning) is a less effective support method for male designers. There is also no evidence suggesting that the DP approach (the use of abstract algorithmic patterns) is less effective for females. Therefore the stated above differences (Exhibit 4.1) can simply indicate that at some stages, female designers might prefer to reuse solutions, while male designers might tend to 'guess and check' things on their own (See Appendix for more details).

4.3 Algorithmic modelling [visual programming] platform

The DP and CBD approaches were tested using Grasshopper (Grasshopper3d, 2014) graphical algorithm editor tightly integrated with Rhino's 3-D modeling tools (Rhino3d, 2014). Grasshopper is a software platform, which provides a visual interface to programming (box-and wire interface). Visual programming is often considered to be more intuitive

and easier to use, causing less barriers associated with the use of programming in architecture and design, compared to the use of textual programming (scripting). Recent studies show that some programming barriers have significantly decreased with the development of visual programming software, such as Grasshopper (Celani, Vaz, 2012). Both visual and textual programming languages are currently used in computational design in architecture. Despite the differences, there are fundamental similarities between both programming languages. The use of both visual and textual programming methods require designers to adopt 'a mode of computational design thinking' (Menges, Ahlquist, 2011). This thinking mode implies that a designer has a deep understanding of algorithmic rules, methods, and behaviours of forces and forms (Woodbury, 2010). That is why among the objectives of both reuse approaches is to assist architects and designers with practical implementation of algorithmic modelling, as well as to help them understand how the form-making and programming constructs work together. An ability to switch-on the algorithmic thinking mode, which allows designers to translate their design concepts into programming algorithms, is often a greater challenge than mastering computational design techniques, such as the use of scripting (Menges, Ahlquist, 2011). That is why it can be expected that the use of both the DP and CBD approaches in the context of textual programming should not be dramatically different from the results obtained in the context of visual programming. However, it is also possible that the use of scripting can impose different challenges on users, due to the fact that the use of visual and textual programming languages require designers to have different sets of skills (techniques). For example, it is possible that the use of scripting can cause more problems with syntax (rules defining textual programming languages). This research shows that the CBD approach is

a more effective method to overcome or to reduce '*use barriers*' (Ko, Myers and Aung, 2004), which refer to problems with the implementation of programming components and syntax problems (See Reuse of Case-Based Solutions section). That is why, it is possible that the CBD approach (as well as the DP approach) can work somewhat differently when applied in the context of textual programming in architecture and design.

4.4 Similarities between the DP and CBD reuse approaches

While the methodology and principles of 'abstract' (DP) and 'case-based' (CBD) solutions adaptation differ, both approaches seek to make reuse of algorithmic design knowledge more effective. The core of this idea is that algorithmic design is not properly an invention or creation of something absolutely new, but is rather a process of rediscovery (Terzidis, 2006). This rediscovery can be directly drawn from existing design knowledge, for example though the reuse of programming artefacts, whether those reusable artefacts be abstract (Design Patterns) or specific (Case-Based Design). The objective of both the DP and CBD approaches is to re-cycle algorithmic solutions rather than creating each one anew. In practice, there is no actual need to create every single thing from scratch, because it is highly possible that 'someone, somewhere really did already invent the wheel you are about to reinvent' (Mann, 2005).

The fundamental difference between the DP and CBD approaches is the abstraction level of the reusable programming artefacts such as: patterns with a high level of abstraction for the DP approach and the specific programming algorithms for the CBD approach. The other principal difference between the approaches is the method by which the reusable artefacts are being selected, retrieved and reused. In order to use

Design Patterns one has to learn them first; to be aware 'when' and 'why' to use each pattern, 'what' each pattern does and 'how in principle' it can be done. Once a person knows patterns, they can be applied straight away for each new design problem. The use of the CBD approach does not require pre-acquired knowledge. However for each new design problem, the architect (or designer) has to browse a repository of the case-based solutions in order to locate and retrieve the fitting case. Observations of the CBD group participants show that this process can take a considerable amount of time.

None of these two approaches is either purely abstract or purely case based. There are abstract constructs utilised in the CBD approach and there are also sets of specific programming solutions used in the methodology of the DP approach. The pattern approach uses specific solutions (cases) to illustrate each Design Pattern. To explain patterns for parametric design Robert Woodbury uses the term 'samples' (Woodbury, 2010). On average six samples are used to illustrate each Design Pattern. Combined together this is over seventy specific solutions, which can be viewed as a case-base. There is a certain ambiguity between the relationship and role of patterns and their samples. Some authors state that pattern examples have only secondary value (should be used as illustrations) (Alexander, 1975) (Winn, Calder 2002) (Woodbury, 2010), others argue that samples are as important as the patterns themselves, because users tend to search for specific solutions rather than rely entirely on abstractions (Dearden, Finlay, Allgar, Mcmanus, 2002). There are arguments stating that the original design patterns, developed by Alexander, attempted to interpret design knowledge in an abstract and generalised way and the result had little to do with abstraction (Hua, Fairings, Smith, 1996). In reality, each pattern refers to a collection of specific buildings within specific environments (Ibid). The results of this

study show that when using patterns for parametric design, participants often referred to specific pattern examples, rather than to the abstract solution itself. This issue (of actual role of pattern samples) is likely to be relevant for any approach dealing with the reuse of abstract artefacts. That is why it seems reasonable to acknowledge that in practice, pattern samples are not being a mere illustration of a 'big idea', but that they perform a wider set of roles (functions).

The similarities between the DP and CBD approaches might suggest that, there could be a hybrid approach, which engages with the reuse of both abstract and specific programming artefacts. The CBD approach uses indexes, some of which have a certain level of abstraction, such as: 'distance', 'proximity', 'condition', 'panelling' etc. These indexes are used to sort and select cases from the repository, but they also can be seen as a grouping principle, or generalisation. The generalisation and abstraction of cases relate the CBD approach back to the patterns. Design Patterns can be used as the generalisation principles, grouping and indexing cases of a repository. Some of the CBD group participants suggested that additional to having the index search, an algorithmic Case-Base repository can be easier to navigate if the cases were organised into some sort of main pre-defined meta-group(s).

The thirteen patterns for parametric design can easily be used for organising the current and future solutions into the meta-groups. Potentially this could make the selection and retrieval of cases more efficient. However the use of this hybrid (DP/CBD) approach would imply that all users are already familiar with the concept of patterns. Alternatively, the Case-Base system can provide designers with the explanation of the patterns concepts and provides the description of all the thirteen Design Patterns. From the teaching perspective, the use of Design Patterns proved to be an effective way to systematically introduce

designers and architects to algorithmic modelling (See Appendix). The integration of a unified (as opposed to segmentation of pattern samples) and organised case-based repository can potentially make the hybrid DP/CBD method more intuitive, because participants found the CBD method to be significantly more helpful, intuitive and easy-to-use compared to the DP approach.

5. Recommendations

5.1 Recommendations for teaching programming in design, based on the lessons learned from this study

A range of practical lessons was learned throughout the course of this study, testing the reuse of design knowledge as a method to support learning and use of algorithmic design in architecture

From a teaching perspective, the systematic inclusion of Design Patterns and Case-Based reasoning into the learning narrative of programming in architecture and design proves to be highly beneficial. The use of these can improve the learners' ability to overcome programming barriers and help to enable computational (algorithmic) design thinking. Since the DP and CBD approaches were tested on the novice programmers, the findings of this study can be used to provide the basis for strategic teaching approaches, which utilise the reuse of programming artefacts. The lessons learned from this study can be applied to inform and (potentially improve) the methodology for teaching programming in architecture and design disciplines.

During the initial learning stages the use of abstract parametric patterns, described by Woodbury (2010), allows designers to better understand the underlying logic of programming design methods: learning through the systematic use of patterns assist designers to develop

and practically employ a computational thinking mode. Gaining this computational thinking mode is essential for 'idea-to-algorithm translation', which (according to the results of this study) is one of the biggest challenges among the novice programmers. Practice shows that even in cases when learners do not actually reuse any Design Patterns in the context of their current design solutions, knowing 'why' and 'how' these abstract algorithmic concepts work is still highly beneficial to them. The results indicate that being introduced to algorithmic modelling through patterns is likely to significantly reduce the overall number of programming difficulties and improve design performance (See the Reuse of Abstract Solutions Section).

The methodology for teaching programming by using Design Patterns proposed and tested in this study can be summarised as a following step–by-step program (as developed for algorithmic modelling workshops using visual programming with Grasshopper for Rhino) (See detailed 'Proposed curriculum of teaching programming in architecture using patterns for algorithmic design' in the Appendix). The general rule for organising the course was to gradually increase the complexity of used programing components and programming logic. 'Parametric Architecture with Grasshopper' (Arturo, 2011) and 'Grasshopper Primer' (Payne, Rajaa, 2009) were used to inform order and structure of the introduced concepts and programming components. Patterns that could be illustrated using very basic algorithms were introduced first and patterns that required more advanced programming skills – were introduced last. Patterns were also clustered according to their related patterns (Woodbury, 2010). In his book 'Elements of Parametric Design' Woodbury (Ibid) documents and explains all the patterns. This information can also be found online (Designpatterns, 2014). Both in the book and in the website, design patterns are sorted in alphabetical order, based on the first letter in the name of each pattern.

The proposed curriculum of teaching programming in architecture using patterns for parametric design is outlined below. It suggests the order in which patterns can be introduced to learners and specifies the content of programming tutorial topics, such as: *Lists, Data Management; Numerical sequences mathematical operations and functions, Paneling Tools, loops, etc.* The curriculum was structured to allow the combination of several design patterns in the later stages of the course to produce more complex programming algorithms and show how different programming logic can work together.

Note that prior to teaching these patterns it was necessary to make designers familiar with the interface and software use basics. For Grasshopper/Rhino this Introduction covered such topics as: *Working area (Interface); Components and data; Components' connection; Parameters and components; Direct import from Rhino (Linking geometry); Data Management; Data Stream Matching; Scalar Component Types; Operators Parametric control*.

1) 'Clear Names'. The first pattern to be introduced in the course is 'Clear Names'. It has actually nothing to do with algorithmic design per se. Its intent is to give each pattern a clear, meaningful and memorable name (Woodbury, 2010). The 'Clear Names' pattern can be used to illustrate the idea and organisational structure of design patterns (What (Intent), When, and How) (Ibid).

2) The 'Jig' pattern describes a concept of using simple abstract frameworks to isolate structure and location from geometric detail (Ibid). This pattern can be illustrated using an example of points that control the geometry of a curve (or a surface). 'Jig' can be explained using relatively simple programming logic. (See the full collection of pattern samples developed by Robert Woodbury on <http://www.designpatterns.ca>).

The following concepts can be introduced together with programming algorithms illustrating the 'Jig' pattern: *Numeric data; Coordinates; Points, Vector Basics; Point/Vector Manipulation, Curves; Creating Lines/Polylines/Curves from points; Surface Types; Creating Surfaces from Points and Curves.*

3) 'Mapping' is a pattern, which uses a function in a new domain and range (Ibid). 'Mapping' sample algorithms can include such programming concepts as: *Lists, Shifting Data, Mathematics; Functions (F(x); Sine/Cosine); Curve analysis; Evaluate Curve; Surfaces' analysis; Evaluate Surface; Reparameterize.*

4) The intent of the 'Point Collection' pattern is to organise collections of points or point-like objects (Ibid). This pattern can be used to create algorithms which illustrate the use of: *Points; Grids of points; Vectors; Translations (such as Move); Mathematical and logical functions; Numerical sequences.*

5) 'Increment'. The intent of the 'Increment' pattern is to drive change through a series of closely related values (Ibid). The 'Increment' and 'Point Collection' patterns can be easily combined together. The following concepts can be introduced using the 'Increment': *Lists; Data Management; Numerical sequences; Series; Range; Random; Fibonacci series; Data Tree; Flatten Tree; Merge; Graft Tree; Tree Branch; Explode Tree;*

6) 'Place holder' describes the logic of using a proxy object (for example a panel) to organise multiple inputs (multiple panels on a surface) (Ibid). This pattern is closely related to the 'Point collection' pattern and can be combined with 'Increment', which is why they are introduced close to each other. The programming algorithms illustrating the 'Place holder' pattern can include: *Paneling Tools; Surfaces' analysis; Divide Surface;*

Isotrim (SubSrf); Translations: Move; Rotations; Orient; Transformations with shape variation; Scale.

7) 'Projection' is a design pattern used to produce a transformation of an object in another geometric context (Ibid). This patterns can be illustrated using: *Curves, Surfaces; Vectors; Project; Graph Mapper; Deformations: Morphing*.

8) 'Selector' refers to conditional constructs ('If - Then – Else' type of programming algorithms). The intent of the 'Selector' pattern is to select particular items in a collection that have specified properties; for example the size of the objects or their index number. It can be presented using programming algorithms which introduce: *Lists; List Item; List Length; Reverse List; Shift List; Split List; Cull Nth; Cull Pattern; Dispatch; Conditional Statements, Range, Series, Interval*.

9) 'Reactor' is a design pattern, which is used to make an object respond to the proximity of another object (Ibid). Reactor can be easily combined with almost any previously introduced patterns, such as 'Selector' (select objects based on their proximity to the other object) and 'Point Collection' (change the location of the points depending on the proximity to an object). Reactor pattern can be illustrated using: *Conditional Statements, Distance, Attractors; Definitions; Attractor point; Attractor curve;*

10) The intent of the 'Controller' pattern is to control a more complex model (or a part of this model) through a simple separate model (Ibid). The use of this pattern implies that the main model has a relatively high degree of complexity. That is why it might be easier to control this model through the separate (simple) model. It is recommended to illustrate this pattern together with a couple of other patterns (for example with 'Point Collection', 'Place Holder', 'Reactor' or 'Selector'). The programming algorithms using the 'Controller' pattern can contain: *Curves, Surfaces;*

Vectors; Paneling Tools; Divide Surface; Translations: Move; Rotations; Orient; Distance, Attractors; etc.

11) 'Reporter'. The idea behind the 'Reporter' pattern is to take information from a model and to communicate it to the audience (re-present it) (Ibid). This pattern can be very useful in the later stages of the design (for example for the representation of elements properties using gradient colours.) It proved to be very effective when applied during the preparation of a digital model for fabrication. For example, 'Reporter' can be used to assign a certain number (index) to each panel or section of a model which is going to be laser-cut. The 'Reporter' pattern can be illustrated with: *Colours, Gradients, Text Display, Lists, Numeric data, Series, Analysis of the curves and surfaces.*

12) The 'Goal Seeker' pattern also refers to the conditional 'If - Then – Else' type of programming constructs. The idea of this pattern is to adjust inputs until a specific goal is reached. The illustration of this pattern will most likely require the use of scripting. The 'Goal Seeker' pattern can be illustrated using: *Script Component, Visual Basic, Variables; Arrays and Lists; Loops.*

13) The idea of the 'Recursion' pattern is to create a pattern by replicating a geometric object or motif (Ibid). Similar to the 'Goal Seeker' the illustration of the 'Recursion' will most likely require the use of scripting. 'Recursion' can be used to create fractals - repeating self-similar patterns. The 'Recursion' pattern can be explained using: *Script Component, Visual Basic, Variables; Arrays and Lists; Loops, Recursion, and Fractals.*

This systematic methodology for teaching programming in architecture using Design Patterns can provide the basis for strategic approach that can be applied for both long term algorithmic design courses as well as for the short term intensive workshops. This teaching framework was successfully tested on a series of algorithmic modelling

workshops using visual programming with Grasshopper/Rhino. This method allows novice programmers to activate computational thinking and gain practical skills (as tested on a diverse group of students, and practicing architects and designers). (See Detailed 'Proposed curriculum of teaching programming in architecture using patterns for algorithmic design' in the Appendix)

5.2 Lessons regarding the use of patterns for parametric design

- Learning the patterns for parametric design helps architects and designers to activate computational thinking mode. Learning programming through Design Patterns proved to reduce programming barriers that novice programmers often face when mastering algorithmic modelling systems.
- In many cases designers and architects tend to remember and refer to some specific pattern examples, rather than patterns themselves;
- In some cases designers may forget or replace certain pattern names, but still use the patterns. For example 'Reporter' was often referred to as a 'Proximity' or 'Distance' pattern; 'Place Holder' was sometimes referred to as 'Paneling', 'Increment' as 'Series'; 'Projector' as 'Project' etc.
- Participants who used Design Patterns were less committed to actually model their original (previously sketched) designs, compared to those participants who used the Case-Based Design approach.
- The use of the Design Pattern approach in the initial stages of learning of programming in architecture encourages exploration of the

software. It proved to help designers in getting familiar with the software. It also encourages the experimentation with forms and various design iterations which might be useful during conceptual design stages.

5.3 Lessons regarding the use of case-based design and the organisation of the CBD systems

- It is recommended to use the CBD approach after designers and architects, who are novice in programming, gain some experience with the tool. This means that they have already acquired basic programming skills and are familiar with the fundamentals of algorithmic design methods (if learners are taught design patterns, the use of the CBD approach is recommended only after they have learned design patterns). It was observed that the reuse of Case-Based Design solutions is likely to discourage the exploration of the software interface and available commands and options. In some cases designers might reuse algorithms to get a desired result (outcome) without clearly understanding 'how' this algorithm actually works, which defeats the whole purpose of learning.
- After designers get more familiar with the modelling tool and the use programming algorithms (when they can use computational thinking mode and are able to create simple algorithms on their own), the use of the Case-Based Design approach can be very effective. Unlike Design Patterns it can show designer 'how exactly' a particular problem can be solved. The CBD approach proves to reduce programming barriers associated with the syntax and implementation of programming components.
- The reuse of Case-Based programming solutions motivates designers to find simpler/more effective algorithms.

- Those who use the CBD approach when working on their own projects are likely to be less inclined to experiment with parameters and be more motivated to realise their original idea (design task).
- When organising a repository of parametric design solutions it is useful to:

1) Organise pre-defined 'meta-groups', based on the programming logic of algorithms. This could be done using Design Patterns typology;

2) Use consistent index dictionary, with the focus on programming commands or geometric characteristics of the output model, rather than using abstract indexes (associations/metaphor/descriptive attributes)

3) Visual representation of design output is very important. The feedback from the CBD group participants indicates that after initial index search they often relied on visual analysis of the output geometry when selecting a case to reuse.

4) Split complex programming solutions into parts: simple reusable artefacts.

6. Conclusion

The evidence presented in this thesis demonstrates that, in the context of algorithmic architectural design, the integration of knowledge reuse approaches, with learning and design processes, is beneficial. This thesis has been tested in empirical studies with groups of students and architects. Three different approaches were employed; two groups used an abstract and a case-based approach to knowledge reuse and a control group had no structural approach. Both extremes of the knowledge reuse approach reduced barriers to using programming in design and improved design performance. The group size and research design enabled these results to be established as statistically significant.

Design Patterns developed by Robert Woodbury (an example of the abstraction reuse) proved to be an effective design support and learning method, significantly reducing learning barriers associated with the use of algorithmic modelling systems and programming languages. The use of abstract solutions (patterns) helps architects to understand and adopt algorithmic design methods better. Even though most of the participating designers and architects found the use of patterns to be less intuitive and less easy-to-use compared with the reuse case-based algorithmic solution, overall the pattern approach proved to be a more effective design support method, particularly at the initial stages of learning.

The use of the Case-Based Design approach (reusing specific algorithmic solutions) helps to reduce problems associated with use barriers (the implementation programming components and syntax), which often occur when designers know 'what to use', but do not know 'how to use it'. However, the reuse of case-based solutions does not reduce the overall number of problems, and seems to discourage design exploration. It encourages more focused reasoning, oriented towards the realisation of the original design intention.

Bibliography

Aamodt, A., & Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. AI communications, 7(1), 39-59.

Abdelsalam, M. (2009). The Use of the Smart Geometry through Various Design Processes: Using the programming platform (parametric features) and generative components. pp. 297-304

Akerkar, R., & Sajja, P. (2010). Knowledge-based systems. Jones & Bartlett Publishers.

Alexander, C. (1979). The timeless way of building (Vol. 1). Oxford University Press.

Alexander, C. (Ed.). (1975). The Oregon experiment (Vol. 3). Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). Pattern languages. Center for Environmental Structure, 2.

Alreck, P. L., & Settle, R. B. (1995). The Survey Research Handbook: Guidelines and Strategies for Conducting a Survey, 2E.

Altshuller, G. S. (1988) TRIZ-88 Available from: Open Source Repository <http://www.altshuller.ru/engineering/engineering16.asp> (accessed 23 September 2014)

Altshuller G. S. (1975) Manuscript, Available from: Open Source Repository <http://www.altshuller.ru/triz/standards1.asp> (accessed 23 September 2014)

Altshuller, G. S. (1984). Creativity as an exact science. Gordon and Breach.

Altshuller, G. S. (1999). The innovation algorithm: TRIZ, systematic innovation and technical creativity. Technical Innovation Center, Inc.

Anderson, J. R. (2013). The architecture of cognition. Psychology Press.

Andia, A. (2001). Integrating digital design and architecture during the past three decades. In Virtual Systems and Multimedia, 2001. Proceedings. Seventh International Conference on (pp. 677-686). IEEE.

Arturo, T. (2011). Parametric Architecture with Grasshopper.

Autodesk, 3Ds Max. (2012). Available from: Open Source Repository <http://usa.autodesk.com/> (accessed 23 July 2012).

Baerlecken, D., Manegold, M., Reitz, J., & Kuenstler, A. (2010). Integrative Parametric Form-Finding Processes. In New Frontiers: Proceedings of the 15th International Conference on Computer-Aided Architectural Design Research in Asia CAADRIA (pp. 303-312).

Bakar, N. A., & Rahim, Z. A. (2014). Design-To-Cost Framework in Product Design Using Inventive Problem Solving Technique (TRIZ). Journal on Innovation and Sustainability. RISUS ISSN 2179-3565, 5(2), 3-17.

Ball, P. (2011). Shapes: nature's patterns: a tapestry in three parts (Vol. 1). Oxford University Press.

Bareiss, R. (1989). Exemplar-based knowledge acquisition: A unified approach to concept representation, classification, and learning. Boston, Academic Press.

Bayle, E., Bellamy, R., Casaday, G., Erickson, T., Fincher, S., Grinter, B. & Thomas, J. (1998). Putting it all together: towards a pattern language for interaction design: A CHI 97 workshop. ACM SIGCHI Bulletin, 30(1), 17-23.

Benton, S. (2007). Mediating between Architectural Design Ideation and Development through Digital Technology, Predicting the Future [25th eCAADe Conference Proceedings / ISBN 978-0-9541183-6-5] Frankfurt am Main (Germany), 26-29 September 2007, pp. 253-260.

Biggerstaff, T. J., & Richter, C. (1989, March). Reusability framework, assessment, and directions. In Software reusability: vol. 1, concepts and models (pp. 1-17). ACM.

Blogger. (2014) Available from: Open Source Repository <https://www.blogspot.com/> (accessed 1 October 2014).

Bonnardel, N., & Zenasni, F. (2010). The impact of technology on creativity in design: An enhancement?. Creativity and innovation management, 19(2), 180-191.

Borchers, J. O. (2001). A pattern approach to interaction design. AI & Society, 15(4), 359-376.

Branting, L. K. (1991). Exploiting the complementarity of rules and precedents with reciprocity and fairness. In Proceedings: Case-Based Reasoning Workshop (pp. 39-50).

Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. Computer networks and ISDN systems, 30(1), 107-117.

Buda, A., & Jarynowski, A. (2010). Life-time of correlations and its applications vol. 1, Wydawnictwo Niezalezne: 5–21, December 2010. ISBN 978-83-915272-9-0.

Burry, M. (2011). Scripting cultures: Architectural design and programming. John Wiley & Sons.

Cao, Q. and Protzen, J. (1999). Managing Design Information. Design Studies, 20(24), 343–362.

Carbonell, J. G. (1986). Derivational analogy; A theory of reconstructive problem solving and expertise acquisition. In R.S. Michalski,

J.G. Carbonell, T.M. Mitchell (eds.): Machine Learning - An artificial Intelligence Approach, Vol.II, Morgan Kaufmann, pp. 371-392.

Carifio, J., & Perla, R. J. (2007). Ten common misunderstandings, misconceptions, persistent myths and urban legends about Likert scales and Likert response formats and their antidotes. Journal of Social Sciences, 3(3), 106.

Celani, G., & Vaz, C. E. V. (2012). Cad scripting and visual programming languages for implementing computational design concepts: A comparison from a pedagogical point of view. International Journal of Architectural Computing, 10(1), 121-138.

Charlesworth, C. (2007). Student Use of Virual Physical Modeling in Design Development-An Experiment in 3D Design Education. The Design Journal, 10(1), 35-45.

Chen, Z. R. (2007). How to improve Creativity: Can Designers Improve Their Design Creativity by Using Conventional and Digital media simultaneously?, CAAD Futures 2007, Australia.

Christiaans, H. H. C. M., & Dorst, K. (1992). An empirical study into design thinking. Research in Design Thinking, N. Roozenburg and K. Dorst, eds., Delft University Press, Delft.

Coad, P. (1992). Object-oriented patterns. Communications of the ACM, 35(9), 152-159.

Coates, P. (2010). Programming. Architecture. Routledge.

Cohen, J. (2013). Statistical power analysis for the behavioral sciences. Routledge Academic.

Coplien, J. O., & Alexander, A. W. O. (1996). Software patterns.

Cui, J., & Tang, M. X. (2013). Integrating shape grammars into a generative system for Zhuang ethnic embroidery design exploration. Computer-Aided Design, 45(3), 591-604.

Dave, B., Schmitt, G., Faltings, B., & Smith, I. (1994, January). Case based design in architecture. In Artificial Intelligence in Design'94 (pp. 145-162). Springer Netherlands.

Davis, D. (2013). "Modelled on Software Engineering: Flexible Parametric Models in the Practice of Architecture." PhD dissertation, RMIT University.

Dearden, A. M., Finlay, J., Allgar, E., & McManus, B. (2002). Using pattern languages in participatory design.

Dearden, A., & Finlay, J. (2006). Pattern languages in HCI: A critical review. Human–computer interaction, 21(1), 49-102.

Designpatterns. (2014). Available from: Open Source Repository <http://www.designpatterns.ca> (accessed 1 October 2014)

Domeshek, E. A., & Kolodner, J. L. (1992). A case-based design aid for architecture. In Artificial Intelligence in Design'92 (pp. 497-516). Springer Netherlands.

Dorta, T. (2007). Augmented sketches and models: the hybrid ideation space as a cognitive artifact for conceptual design. Proceedings of Digital Thinking in Architecture, Civil Engineering, Archaeology, Urban Planning and Design: Finding the Ways, EuropIA, 11, 251-264.

Dorta, T. (2007). Implementing and assessing the hybrid ideation space: a cognitive artefact for conceptual design. Moon, 61, 77.

Dorta, T., Perez, E., & Lesage, A. (2008). The ideation gap:: hybrid tools, design flow and practice. Design Studies, 29(2), 121-141.

Dovey, K. (1990). The pattern language and its enemies. Design Studies, 11(1), 3-9.

Dowdy, S., & Wearden, S. (1983). Statistics for Research, 1983.

Eggenschwiler, T., & Gamma, E. (1992). ET++ SwapsManager: Using object technology in the financial engineering domain. ACM Sigplan Notices, 27(10), 166-177.

Engebretson, A., & Wiedenbeck, S. (2002). Novice comprehension of programs using task-specific and non-task-specific constructs. In Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on (pp. 11-18). IEEE.

Englebart, D. C. (2003, September). Improving our ability to improve: A call for investment in a new future. In IBM Co-Evolution Symposium.

Ericsson, K. A., & Simon, H. A. (1984). Protocol analysis. MIT-press.

Fadem, B. (2008). BRS Behavioral Science, Lippincott Williams & Wilkins

Feller, W. (1950). An Introduction to Probability Theory and Its Applications: Volume One. John Wiley & Sons.

Fisher, R. A. (1925). Statistical methods for research workers. Genesis Publishing Pvt Ltd.

Forrest, A. R. (1974). Computational geometry-achievements and problems. Computer Aided Geometric Design, 17-44.

Gabriel, R. P. (1996). Patterns of software (Vol. 62). New York: Oxford University Press.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design (pp. 406-431). Springer Berlin Heidelberg.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: elements of reusable object-oriented software. Pearson Education.

Garlan, D., & Delisle, N. (1990). Formal specifications as reusable frameworks. In VDM'90 VDM and Z—Formal Methods in Software Development (pp. 150-163). Springer Berlin Heidelberg.

Garland, M. (1999). Multiresolution modeling: Survey & future opportunities. State of the Art Report, 111-131.

Gay, L. R., & Diehl, P. L. (1992). Research methods for business and management. Macmillan Coll Div.

Generative components. (2012). Available from: Open Source Repository <http://www.bentley.com/> (accessed 23 July 2012).

Gentner, D. (1983). Structure-Mapping: A Theoretical Framework for Analogy*. Cognitive science, 7(2), 155-170.

Gero, J. S. (1996). Design tools that learn: A possible CAD future. Information Processing in Civil and Structural Design, Civil-Comp Press, Edinburgh, 17-22.

Glass, G. V., Peckham, P. D., & Sanders, J. R. (1972). Consequences of failure to meet assumptions underlying the fixed effects analyses of variance and covariance. Review of educational research, 237-288.

Grasshopper 3D. (2014) Available from: Open Source Repository <http://www.grasshopper3d.com/> (accessed 1 October 2014).

Groat, L., & Wang, D. (2002). Architectural research methods. New York.

Hamade, R. F., & Artail, H. A. (2008). A study of the influence of technical attributes of beginner CAD users on their performance. Computer-Aided Design, 40(2), 262-272.

Heisserman, L. (1994). Generative geometric design. Computer Graphics and Applications, IEEE, 14(2), 37-45.

Heylighen, A., & Neuckermans, H. (2001). A case base of case-based design tools for architecture. Computer-Aided Design, 33(14), 1111-1122.

Heylighen, A., & Verstijnen, I. M. (2000). Exposure to examples. In Artificial Intelligence in Design'00 (pp. 413-432). Springer Netherlands.

Howe, N. (2011). Algorithmic Modeling: Teaching Architecture in Digital Age. Do Not Print, 17.

Hua, H. (2014). A case-based design with 3D mesh models of architecture. Computer-Aided Design.

Hua, K., Fairings, B., & Smith, I. (1996). CADRE: case-based geometric design. Artificial Intelligence in Engineering, 10(2), 171-183.

Huang, L. (2009). Technology in Computer Aided Architectural Design, ICIC '09 Proceedings of the 2009 Second International Conference on Information and Computing Science - Volume 02 Pages 221-223

Hubbard, R., & Lindsay, R. M. (2008). Why P values are not a useful measure of evidence in statistical significance testing. Theory & Psychology, 18(1), 69-88.

IBM SPSS. (2014) Available from: Open Source Repository <http://www-01.ibm.com/software/analytics/spss/> (accessed 1 October 2014).

Iwamoto, L. (2013). Digital fabrications: architectural and material techniques. Princeton Architectural Press.

Jamieson, S. (2004). Likert scales: how to (ab) use them. Medical education, 38(12), 1217-1218.

Janssen, P., & Wee, C. K. (2011). Visual Dataflow Modelling: A Comparison of Three Systems.

Jonson, B. (2005). Design ideation: the conceptual sketch in the digital age. Design studies, 26(6), 613-624.

Kalay, Y. E. (1999). The future of CAAD: From computer-aided design to Computer-aided collaboration. In Computers in Building (pp. 13-30). Springer US.

Kaplan, S. (1996). An introduction to TRIZ: The Russian theory of inventive problem solving. Ideation International.

Karle, D., & Kelly, B. (2011). Parametric Thinking. In Proceedings of ACADIA Regional 2011 Conference (pp. 109-113).

King, I. (1993). Christopher Alexander and Contemporary Architecture. Special issue of Architecture and Urbanism, August 1993

Ko, A. J., Myers, B. A., & Aung, H. H. (2004, September). Six learning barriers in end-user programming systems. In Visual Languages and Human Centric Computing, 2004 IEEE Symposium on (pp. 199-206). IEEE.

Kolodneer, J. L. (1991). Improving human decision making through case-based decision aiding. AI magazine, 12(2), 52.

Kolodner, J. L. (1983). Reconstructive Memory: A Computer Model*. Cognitive science, 7(4), 281-328.

Kolodner, J. L. (Ed.). (1993). Case-based learning (Vol. 10, No. 3). Springer.

Krish, S. (2011). A practical generative design method. Computer-Aided Design, 43(1), 88-100.

Krishnamurti, R. (2011). Bridging parametric shape and parametric design. In SDC'10: NSF International Workshop on Studying Visual and Spatial Reasoning for Design Creativity.

Krueger, C. W. (1992). Software reuse. ACM Computing Surveys (CSUR), 24(2), 131-183.

Kwinter, S., & Davidson, C. (2008). Far from equilibrium: essays on technology and design culture. ACTA Press.

Lano, K. (2014). Design patterns: applications and open issues. In Cyberpatterns (pp. 37-45). Springer International Publishing.

Lea, D. (1994). Christopher Alexander: An Introduction for Object-Oriented Designers, Software Engineering Notes, 19 (1) 39-46

Leach, N. (2009). Digital morphogenesis. Architectural Design, 79(1), 32-37.

Leach N. (2010). Parametric and Algorithmic research in architecture, Definitions: Parametric and Algorithmic Design, University of Southern California, US <http://parasite.usc.edu/?p=443>.

Leach N., Schumacher P. (2012). On Parametricism'- A Dialogue between Neil Leach and Patrik Schumacher. Published in: T + A (Time + Architecture) Digital Fabrication, International Architectural Magazine in China

Leitão, A., & Santos, L. (2011). Programming Languages for Generative design: Visual or Textual?. In Zupancic, T., Juvancic, M., Verovsek., S. and Jutraz, A., eds., Respecting Fragile Places, 29th eCAADe Conference Proceedings, University of Ljubljana, Faculty of Architecture (Slovenia), Ljubljana (pp. 549-557).

Lubke, G. H., & Muthén, B. O. (2004). Applying multigroup confirmatory factor models for continuous outcomes to Likert scale data complicates meaningful group comparisons. Structural Equation Modeling, 11(4), 514-534.

MacLean, A., Young, R. M., Bellotti, V. M., & Moran, T. P. (1991). Questions, options, and criteria: Elements of design space analysis. Human–computer interaction, 6(3-4), 201-250.

Maher, M. L., & de Silva Garza, A. G. (1997). Case-based reasoning in design. IEEE Intelligent Systems, 12(2), 34-41.

Maher, M. L., & Pu, P. (Eds.). (2014). Issues and applications of case-based reasoning to design. Psychology Press.

Maher, M. L., Balachandran, M., & Zhang, D. M. (1995). Case-based reasoning in design. Psychology Press.

Mallasi, Z. (2007). Applying Generative Modeling Procedure to Explore Architectural Forms. In Proceedings of the ASCAAD Conference (pp. 335-342).

Mann, D. Someone, Somewhere Really Did Already Invent The Wheel You're About To Re-Invent., HKIVM 7th International Conference, 2005.

Marriott, K., & Chok, S. S. (2002). Qoca: A constraint solving toolkit for interactive graphical applications. Constraints, 7(3-4), 229-254.

Martens, B., Koutamanis, A., & Brown, A. (2007). Predicting the future from past experience. Predicting the future, 523-531.

Martin, P., & Bateson, P. (1986). Measuring BehaviourCambridge University Press. Cambridge UK.

Matcha, H. (2007). Parametric Possibilities: Designing with Parametric Modelling. In Predicting the Future: 25th eCAADe Conference Proceedings (pp. 849-856).

MaxScript. (2012). Available from: Open Source Repository <http://docs.autodesk.com/> (accessed 23 July 2012).

Maya. (2012). Available from: Open Source Repository <http://usa.autodesk.com/> (accessed 23 July 2012).

McCormack, J., Dorin, A., & Innocent, T. (2004). Generative design: a paradigm for design research. Proceedings of Futureground, Design Research Society, Melbourne.

Mehta, C. R., & Patel, N. R. (1998). Exact inference for categorical data. Encyclopedia of biostatistics, 2, 1411-1422.

Mel. (2012). Available from: Open Source Repository <http://download.autodesk.com/us/maya/2010help/files/Glossary_M_MEL.htm> (accessed 23 July 2012).

Menges, A. (2012). Material computation: Higher integration in morphogenetic design. Architectural Design, 82(2), 14-21.

Menges, A., & Ahlquist, S. (Eds.). (2011). Computational Design Thinking: Computation Design Thinking. John Wiley & Sons.

Meredith M. (2008). From Control to Design, Parametric/Algorithmic Architecture\ 2008-10-15.

Microsoft Excel. (2014) Available from: Open Source Repository <http://office.microsoft.com/en-us/excel/> (accessed 1 October 2014).

Mitchell, W. J. (1975). The theoretical foundation of computer-aided architectural design. Environment and Planning B, 2(2), 127-150.

Mitchell, W. J. (1990). The logic of architecture: Design, computation, and cognition. MIT press.

Mora, R., Bédard, C., & Rivard, H. (2008). A geometric modelling framework for conceptual structural design from early digital architectural models. Advanced Engineering Informatics, 22(2), 254-270.

Mosier, J. N., & Smith, S. L. (1986). Application of guidelines for designing user interface software. Behaviour & information technology, 5(1), 39-46.

Muller, M. J., Haslwanter, J. H., & Dayton, T. (1997). Participatory practices in the software lifecycle. Handbook of human-computer interaction, 2, 255-297.

Musta'amal, A. H. (2010). An empirical investigation of the relationship of CAD use in designing and creativity through a creative behaviours framework (Doctoral dissertation, © Aede Hatib Musta'amal).

Nielsen, J. (1994). Usability engineering. Elsevier.

Pane, J. F., Ratanamahatana, C., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. International Journal of Human-Computer Studies, 54(2), 237-264.

Paneling-tools. (2009) Available from: Open Source Repository <http://tips.rhino3d.com/2009/04/basics-of-paneling-tools.html> (accessed 24 July 2012).

Payne, Rajaa (2009). Grasshopper Primer. Second edition. LIFT Architects. Accessed September 2014.

Pearce, M., Goel, A. K., Kolodner, J. L., Zimring, C., Sentosa, L., & Billington, R. (1992). Case-based design support: A case study in architectural design. IEEE Expert, 7(5), 14-20.

Pérez, E., Dorta T. (2011). Assessment of design tools for ideation, Proceedings of the 16th International Conference on Computer Aided Architectural Design Research in Asia / The University of Newcastle, Australia 27-29 April 2011, pp. 429-438.

Porter, B. W., & Bareiss, E. R. (1986). PROTOS: An Experiment in Knowledge Acquisition for Heuristic ClassificationTasks.

Processing. (2012). Available from: Open Source Repository <http://processing.org/> (accessed 23 July 2012).

Python. (2012). Available from: Open Source Repository <http://www.python.org/>(accessed 23 July 2012).

Ramirez, A. J., & Cheng, B. H. (2010, May). Design patterns for developing dynamically adaptive systems. In Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (pp. 49-58). ACM.

Recio-García, J. A., González-Calero, P. A., & Díaz-Agudo, B. (2014). jcolibri2: A framework for building Case-based reasoning systems. Science of Computer Programming, 79, 126-145.

Reffat, R. M. (2006). Computing in architectural design: reflections and an approach to new generations of CAAD. Journal of Information Technology in Construction (ITCON), 11, 655-668.

Rhino3d. (2012). Available from: Open Source Repository <http://www.rhino3d.com/> (accessed 23 July 2012).

Rhinoscript. (2012). Available from: Open Source Repository <http://www.rhinoscript.org/> (accessed 23 July 2012).

Riesbeck, C. K., & Schank, R. C. (2013). Inside case-based reasoning. Psychology Press.

Robbins, R. J. (1994). Database Fundamentals. Johns Hopkins University, rrobbins@ gdb. org.

Roscoe, J. T. (1975). Fundamental research, statistics for the behavioral sciences (2nd Ed.). New York: Holt, Rinehart & Winston

Ross, B.H (1989). Some psychological results on case-based reasoning, Case-Based Reasoning Workshop, DARPA 1989. Pensacola Beach. Morgan Kaufmann. pp. 144-147.

Rouse, W. B., & Hunt, R. M. (1982). Human problem solving in fault diagnosis tasks. Georgia Inst. of Tech Atlanta Center for Human-Machine Systems Research Report no 82-3, 1982.

Saeed, G. (2000). Fundamentals of Probability.

Sargent (1991) Knowledge-based systems, Available from: Open Source Repository <http://home.klebos.net/philip.sargent/book/7_knowledge_based_systems.html> (accessed 1 October 2014).

Saunders, W. S. (2002). Book reviews: A pattern language. Harvard Design Magazine, 16, 1-7.

Schank, R. C. (1982). Dynamic memory: A theory of reminding and learning in computers and people. Cambridge University Press.

Schön, D. (1991). Teaching and learning as a design transaction. Research in design thinking.

Schön, D. A. (1983). The reflective practitioner: How professionals think in action (Vol. 5126). Basic books.

Schuler, D., & Namioka, A. (1993). Participatory design: Principles and practices. L. Erlbaum Associates Inc.

Serban, D., Man, E., Ionescu, N., & Roche, T. (2004). A TRIZ approach to design for environment. In Product Engineering (pp. 89-100). Springer Netherlands.

Shah, J. J., Smith, S. M., & Vargas-Hernandez, N. (2003). Metrics for measuring ideation effectiveness. Design studies, 24(2), 111-134.

Shih Y. T., Williams A., Gu N. (2011). A method to investigate differences of sketching before and during CAD modelling design process, Architecture @ the Edge: Association of Architecture Schools of Australasia 2011 International Conference (AASA 2011) The School of Architecture & Building, Deakin University.

Shikhare, D., Bhakar, S., & Mudur, S. P. (2001). Compression of large 3D engineering models using automatic discovery of repeating geometric features. Signal Processing, 19(20), 15.

Siemens, G. (2006). Knowing knowledge. Lulu. com.

Smith, A. C. (2004). Architectural model as machine: A new view of models from antiquity to the present day. Routledge.

Sobek, D and Ward, (1996). A Principles from Toyota's Set-Based Concurrent Engineering Process, Proceedings of ASME Computers in Engineering Conference, Irvine, CA.

Stigler, S. (2008). Fisher and the 5% level. Chance, 21(4), 12-12.

Stiny, G. (2008). Shape: talking about seeing and doing. The MIT Press.

Stoline, M. R. (1981). The status of multiple comparisons: simultaneous estimation of all pairwise comparisons in one-way ANOVA designs. The American Statistician, 35(3), 134-141.

Stratton, R., Mann, D., & Otterson, P. (2000). The Theory of Inventive Problem Solving (TRIZ) and Systematic Innovation-a Missing Link in Engineering Education?. TRIZ Journal.

Sun, K., & Faltings, B. (1994, January). Supporting creative mechanical design. In Artificial Intelligence in Design'94 (pp. 39-56). Springer Netherlands.

Sutcliffe, A. (2000). On the effective use and reuse of HCI knowledge. ACM Transactions on Computer-Human Interaction (TOCHI), 7(2), 197-221.

Terninko, J., Zusman, A., & Zlotin, B. (1998). Systematic innovation: An introduction to TRIZ (theory of inventive problem solving). CRC press.

Terzidis, K. (2006). Algorithmic architecture. Routledge.

Tidwell, J. (2005). Deigning Interfaces-Patterns for Effective Interaction Design. O'Reilly Media, Inc.

Toth, B., Salim, F., Drogemuller, R., Frazer, J. H., & Burry, J. (2011). Closing the loop of design and analysis: Parametric modelling tools for early decision support. In Circuit Bending, Breaking and Mending: Proceedings of the 16th International Conference on Computer-Aided Architectural Design Research in Asia (pp. 525-534). The Association for Computer-Aided Architectural Design Research in Asia (CAADRIA).

Tsatsoulis, C., & Alexander, P. (1997). Integrating cases, sub-cases, and generic prototypes for design. Issues and Applications of Case-Based Reasoning in Design, 261-300.

Tumblr. (2014) Available from: Open Source Repository <https://www.tumblr.com/> (accessed 1 October 2014).)

Turrin, M., von Buelow, P., & Stouffs, R. (2011). Design explorations of performance driven geometry in architectural design using parametric modeling and genetic algorithms. Advanced Engineering Informatics, 25(4), 656-675.

Tuthill, G. S., & Levy, S. T. (1991). Knowledge-based systems: a manager's perspective. TAB Books.

Van Berkel, B., & Bos, C. (2006). After Image. Design Models: Architecture, Urbanism, and Infrastructure. Thames & Hudson. London, UK, 370-379.

Wallenstein, S. Y. L. V. A. N., Zucker, C. L., & Fleiss, J. L. (1980). Some statistical methods useful in circulation research. Circulation Research, 47(1), 1-9.

Walther, J., Robertson, B. F., & Radcliffe, D. F. (2007). Avoiding the potential negative influence of CAD tools on the formation of students' creativity. Department of Computer Science and Software Engineering, the University of Melbourne.

Weisberg, H. F., & Bowen, B. D. (1977). An introduction to survey research and data analysis San Francisco. WH. Freeman and Company, 13, 59-62.

Wiig, K. M. (1999). What future knowledge management users may expect. Journal of knowledge management, 3(2), 155-166.

Winn, T., & Calder, P. (2002). Is this a pattern?. Software, IEEE, 19(1), 59-66.

Woodbury, R. (2010). Elements of parametric design.

Woodward, M. (2010). An Interpretation Design Pattern Language: A propositional conceptual tool for interdisciplinary team members working on interpretation design projects, Cumulus Working Papers, Aalto University School of Art and Design, Melbourne

Wordpress.com. (2014) Available from: Open Source Repository <https://wordpress.com/> (accessed 1 October 2014).

Zeid, I. (2005). Mastering Cad/Cam. McGraw-Hill, Inc.

Zimmerman, J. L. (1997). EVA and divisional performance measurement: Capturing synergies and other issues. Journal of Applied Corporate Finance, 10(2), 98-109.

Zimring, C., Do, E., Domeshek, E., & Kolodner, J. (1995). Supporting case-study use in design education: A computational case-based design aid for architecture. In Computing in Engineering: Proceedings of the Second Congress. New York: American Society of Civil Engineers.

Appendix A

Proposed Curriculum of Teaching Programming in Architecture Using Patterns for Algorithmic Design

This systematic methodology for teaching programming in architecture using Design Patterns can provide the basis for strategic approach that can be applied for both long term algorithmic design courses as well as for the short term intensive workshops. This teaching framework was successfully tested on a series of algorithmic modelling workshops using Grasshopper for Rhino. This method allows novice programmers to activate computational thinking and quickly gain practical skills.

Prior to introducing Design Patterns, there should be a basic introduction of the software interface and the structure of programming components. For Grasshopper this includes finding and selecting different types of programming components, connecting and disconnecting them; linking and modifying existing geometry and creating geometry from scratch. The first step involves making learners familiar with the concepts of domains of numbers, introduction of 'number sliders', mathematical functions and operations, coordinates. It should also include an overview of how to create geometry (2D and 3D primitives) and how to use some of the basic operations, such as: move, rotate, scale; and Boolean operations: intersection, subtraction, addition.

Tutorial Content:

Working area (Interface);

Components and data;

Components' connection;

Parameters and components;

Direct import from Rhino (Linking geometry);

Data Management;

Data Stream Matching;

Scalar Component Types;

2D and 3D Primitives (points, lines, curves, planes, circles, polygons, spheres, boxes etc.)

Operators (move, rotate, scale);

Parametric control;

1 Design Pattern: Clear Names

After designers gain an overall understanding of the software interface and basics of modelling methods, they can be introduced to the concept of patterns for parametric design.

See <http://www.designpatterns.ca> for details.

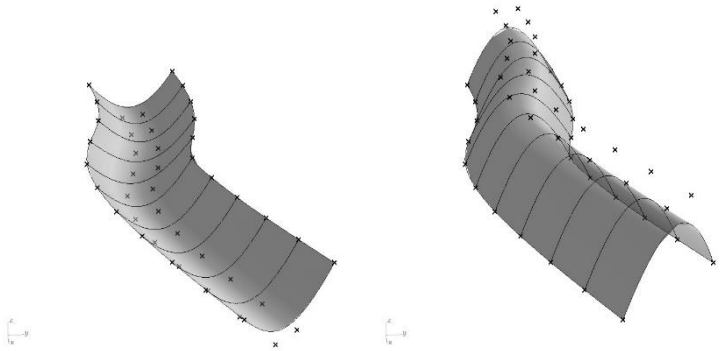
The first pattern to be introduced in the course is 'Clear Names'. It has actually nothing to do with parametric design per se. Its intent is to give each pattern a clear, meaningful and memorable name. The 'Clear Names' pattern can be used to illustrate the concept and organisational structure of design patterns (Intent, Use When, Why, and How) (Woodbury, 2010). Design Patterns can be understood as re-usable abstracted parametric design solutions. To better understand the concept of Design Patterns please refer to the following explanations:

- Design Pattern is an abstract solution, which can be applied to a shared problem (Woodbury, 2010).
- Interpretation of the design idea / concept (Woodbury, 2010);
- Pattern is a 'pre-formal construct' (Lea, 1994);
- Patterns emerge from repetitions of human behaviour (Coad, 1992);
- Pattern is a recurrent phenomenon or structure, 'didactic medium for human readers' (Borchers, 2001);
- Pattern describes a problem and then describes the core of the solution (Gamma, 1994 quote Alexander (1977));
- Pattern is a structured description of invariant solution. Invariant refers to a set of shared characteristics of the recommended solution (Winn, Calder, 2002)
- Patterns should capture 'big ideas' (Winn, Calder 2002) instead of covering every possible design decision.

- Pattern is an abstraction, which describes not some specific example, but it rather refers to a general concept or idea, which is often associated with vagueness. In computer science, an abstraction characterizes a class of instances which omits inessential details (Woodbury, 2010), (Gamma, Helm, Johnson, Vlissides, 1994).
- Design Patterns are the medium to understand and express the practice craft of parametric modelling (Woodbury, 2010)

**Patterns for parametric design used in this course were developed by Robert Woodbury (2010)*

2 Design Pattern: Jig



'Jig' pattern describes a concept of using simple abstract frameworks to isolate structure and location from geometric detail*.

Tutorial Content:

Numeric data;

Coordinates;

Points,

Vector Basics;

Point/Vector Manipulation,

Curves;

Types of Curves

Creating Lines / Polylines / Curves from Points;

Surfaces

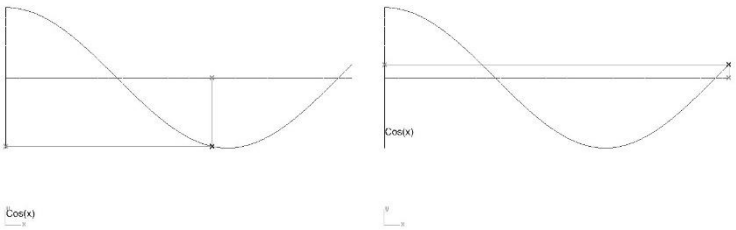
Creating Surfaces from Points and Curves

Notes:

The use of 'Jig' allows designers to learn how they can control an object using its isolated structure. 'Jig' is chosen to be the first design pattern introduced to learners, due to a number of reasons. Firstly, the concept of changing a geometry using, for example, control points is relatively easy to understand, even for novice modellers. Secondly the modification of a geometrical object (such as a curve or a surface) using

control points can be done through a very simple programming algorithm. The objective of the course is to use more simple algorithms and programming logic in the beginning and then gradually increase the complexity.

3 Design Pattern: Mapping



The intent of the 'Mapping' pattern is to use a function in a new domain and range*.

Tutorial Content:

Lists, Shifting Data, Mathematics;

Functions (F(x); Sine / Cosine);

Curve analysis; Evaluate Curve;

Surfaces' analysis;

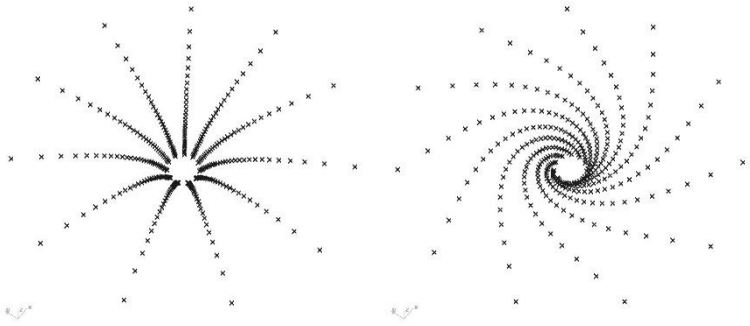
Evaluate Surface;

Reparameterize; 'Remap Numbers'

Notes:

'Mapping' can be combined with further (more detailed) introduction of the use of mathematical functions in parametric design, such as sine, cosine, x*x etc. The introduction of 'Mapping' and illustration of it using programming algorithms can be used to explain the 'Remap Numbers' components and 'Reparameterize' option. The 'Reparameterize' sets the domain from 0 to 1 instead of the real size, which can be really useful for the evaluation of curves and surfaces. Woodbury states that 'It is much, much easier to think about a function in its natural domain and range' (2010)

4 Design Pattern: Point Collection



The intent of the 'Point Collection' pattern is to organise collections of points or point-like objects*.

Tutorial Content:

Points;

Grids of points;

Vectors;

*Functions (F(x); Sine / Cosine / x*x);*

Translations (such as Move);

Mathematical and logical functions;

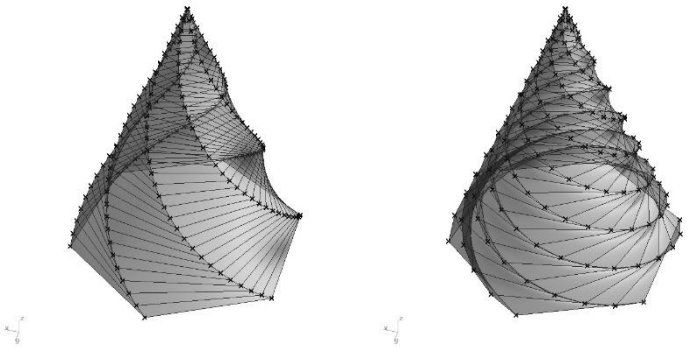
Numerical sequences.

Mathematics;

Notes:

Similar to 'Jig', the concept behind the 'Point Collection' pattern is relatively easy to grasp: locating the repeating elements using various organisational methods. The use of 'Point Collection' also allows the integration of mathematical functions, defining the distribution (location) of each point in the collection. The following examples can be used to illustrate the idea of this pattern: spirals, waves, random point clouds or specifies a position of points on curves and surfaces.

5 Design Pattern: Increment



The intent of the 'Increment' pattern is to drive change through a series of closely related values*.

Tutorial Content:

Lists; Data Management;

Numerical sequences;

Series; Range; Random; Fibonacci series;

Data Tree; Flatten Tree; Merge; Graft Tree; Tree Branch; Explode Tree;

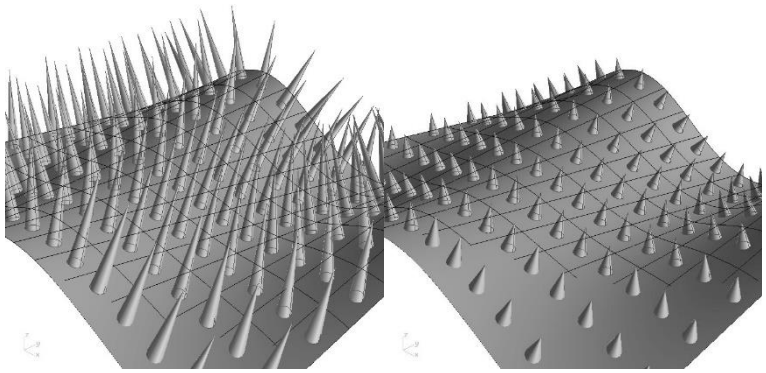
Notes:

'Increment' is one of the patterns that is often re-used by designers in their own design works. Observations show that novice programmers often get excited by the complexity of geometry that can be generated using gradual rotation or move of the objects. Some of designers might not (foresee) predict what kind of geometry can be created using programming algorithms that gradually transforming an object with incremental changes. The 'Increment' and 'Point Collection' patterns can be easily combined together.

It should be noted that, even though designers often use the logic of 'Increment' in their parametric projects, they may tend to forget the name of this pattern. 'Increment' is often referred to it as 'Series', which is a programming component in Grasshopper. Similarly the 'Projection'

pattern is sometimes referred as 'Project' or the 'Reactor' pattern is often called 'Distance' (both of which are programming components). This trend might indicate a couple things: a) these names pattern could be not the most universal, or b) designers and architects tend to remember and associate some specific programming commands (such as: project, series, distance) rather than use the original (more abstract) pattern name.

6 Design Pattern: Place Holder



'Place holder' describes the logic of using a proxy object (for example a panel) to organise multiple inputs (panels on a surface)*.

Tutorial Content:

Paneling Tools;

Surfaces' analysis;

Divide Surface; Isotrim (SubSrf);

Translations: Move; Rotations; Orient;

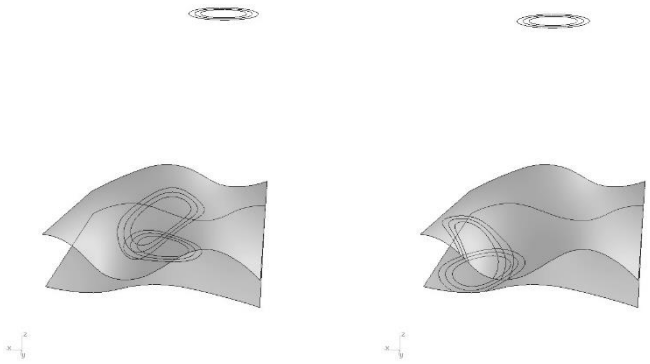
Transformations with shape variation;

Scale.

Notes:

The 'Place holder' pattern is related to the 'Point Collection' pattern. It can also easily be combined with 'Increment' (for example, by rotation or scaling of repeating elements) and with 'Jig' (for example, to control the surface). 'Place Holder' is often associated by designers with the concept of Paneling (however is only one of 'Place Holder's' possible applications). Here is an example how 'Point Collection' and 'Place holder' can be used together: a) use 'Point Collection' to define coordinates of the input objects; b) use 'Place holder' by creating a proxy object (for example 'spines') and referencing it to the locations.

7 Design Pattern: Projection



'Projection' is a design pattern, which is used to produce a transformation of an object in another geometric context.

Tutorial Content:

Curves,

Surfaces;

Vectors;

Project;

Image sampler,

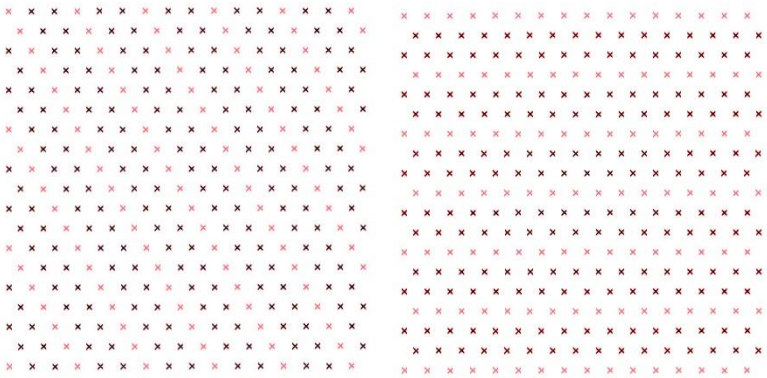
Graph Mapper;

Deformations: Morphing;

Notes:

Even though the concept and the application of the 'Projection' patterns is relatively simple, it allows to create very complex outcomes. One of the algorithms illustrating the 'Projection' pattern can be split it into two parts: creating a relatively complex and detailed 2D pattern using 'Increment', 'Point Collection' and 'Place holder' and then using 'Projection' logic transform this 2D pattern onto a different geometric context (for example project or morph it into a complex curvilinear surface (receiving object)). Alternatively the initial 2D pattern can be created using data from an image ('Image Sampler').

8 Design Pattern: Selector



The intent of the 'Selector' pattern is to select particular items in a collection that have specified properties (for example, their size or their index number).

Tutorial Content:

Lists;

List Item;

List Length; Reverse List;

Shift List; Split List;

Cull Nth; Cull Pattern;

Dispatch;

Conditional Statements,

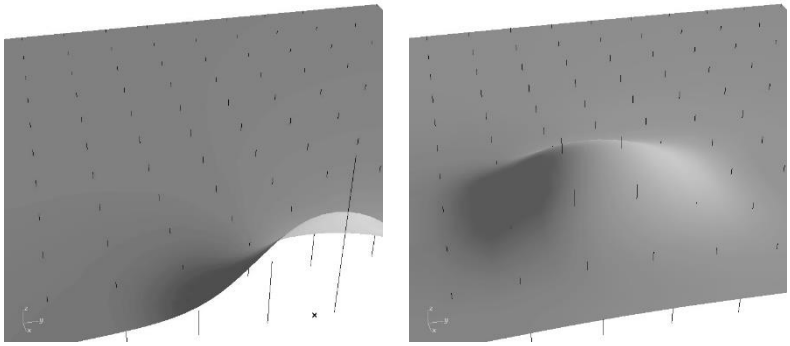
Range, Series, Interval.

Notes:

'Selector' refers to conditional constructs ('If - Then – Else' type of programming algorithms). From teaching perspective, the 'Selector' pattern can be used to give designers a better and more advanced understanding of how the lists of data work. Including the illustrations on how multiple numbers, objects and coordinates can be placed in lists and how this data can be organised and manipulated (data tree structure). 'Selector' can be illustrated with programming algorithms which introduce such concepts as splitting the lists of data, based on the

item's number (index); based on a specific pattern (true / false); or reversing / shuffling the order of data in the list etc. Study shows that designers can easily grasp the idea of the 'Selector' pattern. However practical implementation of conditional constructs and managing the lists of data is often frustrating for novice programmers. That is one of the reasons why this pattern was not introduced in the beginning of the course.

9 Design Pattern: Reactor



'Reactor' is a design pattern, which is used to make an object respond to the proximity of another object*.

Tutorial Content:

Conditional Statements,

Distance,

Attractors;

Definitions;

Attractor point;

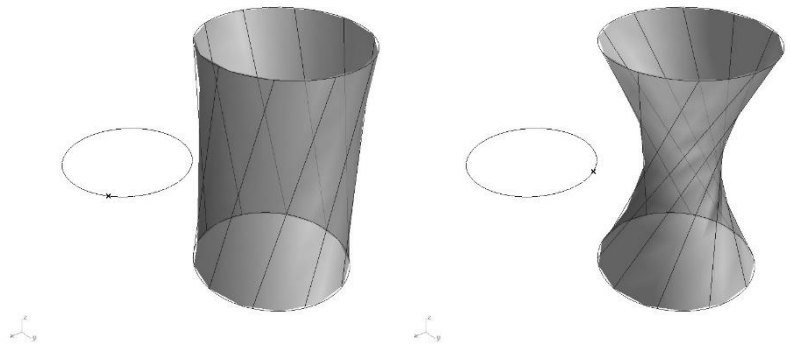
Attractor curve;

Notes:

'Reactor' can be easily combined with almost any previously introduced patterns, such as 'Place Holder' and 'Selector'. For example, selecting objects (sorting them into different lists) based on their proximity to a curve or an attractor points. 'Rector' can be paired with other introduced patterns to create proximity responsive designs. That is one of the reasons why it was introduced later in the course. Proximity is often used to create responsive (interactive) structures. Distance between the objects (for example between the attractor point and elements of the structure) can be used as a parameter that informs the size or a degree of elements' rotation. Some designers, who learned parametric

modelling using Design Patterns had a tendency to intuitively substitute the name ‘Reactor’ with such words as ‘Distance’ and ‘Proximity’. This might suggest that the name ‘Reactor’ might not be the most universal and memorable.

10 Design Pattern: Controller



The intent of the 'Controller' pattern is to control a more complex model (or a part of a model) through a simple separate model*.

Tutorial Content:

Curves, Surfaces;

Vectors;

Paneling Tools;

Divide Surface;

Translations: Move; Rotations; Orient;

Distance,

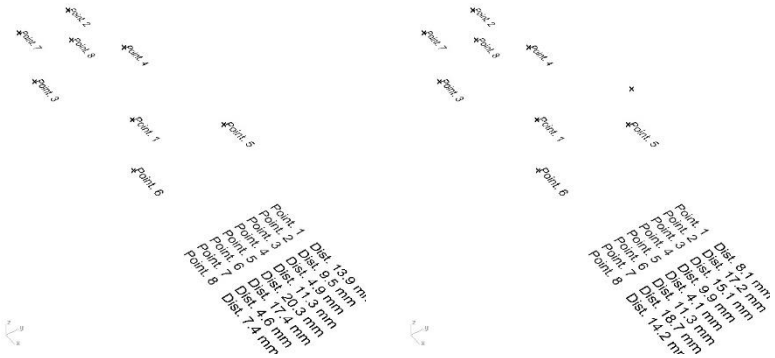
Attractors;

Notes:

The use of the 'Controller' pattern implies that the design model has a relatively high degree of complexity. Which is why it might be easier to control this model through a separate (more simple) model. It is recommended to illustrate this pattern together with a couple of other patterns (for example with 'Point Collection', 'Place Holder', 'Reactor' or 'Selector'). The idea of 'Controller' is closely related to the idea of 'Jig'. Similar to the 'Controller' pattern the objective of the 'Jig' pattern is to control an object using its isolated structure (using for example a set of control points). The difference between these patterns is that the

'Controller' description implies that a separate simple model should be used to control a more complex model (object). When creating their own algorithms, designers sometimes have a tendency to skip the creation of a separate model and instead use isolated structures (points or curves) to control their resulting models.

11 Design Pattern: Reporter



The idea behind the 'Reporter' pattern is to extract information from a model and to communicate it to the audience (represent this information)*.

Tutorial Content:

Colours,

Gradients,

Text Display,

Lists,

Numeric data,

Series,

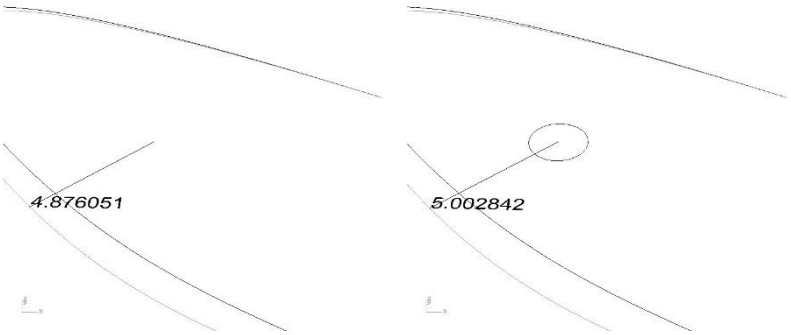
Analysis of the Curves and Surfaces.

Notes:

This pattern can be very useful on the later stages of the design (for example for the representation of elements' properties using gradient colours.) The representation of the information could be done through the use of colours / gradients (for example shading larger elements as red and smaller elements as green) or it could be represented with text (for example showing the area / volume of each element, their proximity to each other, or their index number). The use of the 'Reporter' pattern

has proved to be very useful for the preparation of a digital model for fabrication. For example, by showing an index number of each element (panel or section) of a model that has to be laser-cut.

12 Design Pattern: Goal Seeker



The idea of 'Goal Seeker' is to adjust inputs until a specific goal is reached*.

Tutorial Content:

Variables;

Arrays and Lists;

Loops.

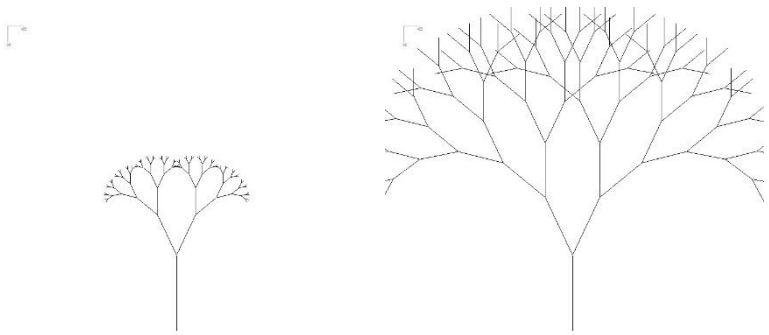
Script Components,

Visual Basic,

Notes:

The 'Goal Seeker' pattern refers to the conditional 'If - Then – Else' type of programming constructs. 'Goal Seeker' can be illustrated by gradually scaling objects in a collection until they reach a specific size (volume), or until a specific distance between the objects is reached. In this regard, the 'Goal Seeker' pattern is related to the 'Selector' pattern, which also employs conditional algorithms (sorting items in a collection according to specified properties). 'Goal Seeker' gives an opportunity to introduce designers to the idea of loops and iterations. It should be noted that in Grasshopper the implementation of conditional statements, and iterations: loops and recursions will most likely require the use of scripting or the use of additional plugins.

13 Design Pattern: Recursion



The idea of 'Recursion' is to create a pattern by replicating a geometric object or motif *.

Tutorial Content:

Recursion,

Fractals

Variables;

Arrays and Lists;

Loops.

Script Components,

Visual Basic,

Notes:

'Recursion' is also related to the concept of loops and iterations. Hence it is clustered with the 'Goal Seeker' pattern. 'Recursion' can be used to create fractals, which are often used as examples of recursions in programming. Similar to the 'Goal Seeker' pattern the illustration of the 'Recursion' pattern will most likely require the use of scripting.

Appendix B

Report of Results

Colour coding of diagrams:

**pink: the p- value indicates that there is a significant difference between the approaches (for this particular criterion)*

**grey: the p- value indicates that there is NO significant difference between the approaches (for this particular criterion)*

COMPARISON BETWEEN THE NO APPROACH, THE DESIGN PATTERNS APPROACH AND THE CASE-BASED DESIGN APPROACH GROUPS (ANOVA / CHI-SQUARE).

**Only the cases when the p-value is below 0.05 are shown*

Criteria	No App.(Mean / %)		DP (Mean / %)		CBD (Mean / %)		t (df) / X^2		p - value	
	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Model Complexity Score	11.73 +- 2.465	13.94 +- 2.585	12.23 +- 2.046	14.10 +- 2.551	12.15 +- 2.246	12.74 +- 2.246	.583 (125)	3.5 (125)	.560	.031
Algorithm Variety Score	12.43 +- 3.565	16.65 +- 5.851	15.13 +- 4.718	17.60 +- 5.137	12.77 +- 3.595	15.77 +- 3.218	4.99 (125)	1.3 (125)	.008	.268
How Often You Have Come Across Programming Difficulties	2.88 +- 1.053	2.71 +- .890	2.37 +- .669	2.10 +- .403	2.91 +- 1.039	2.53 +- .776	3.414 (2)	6.2 (2)	.036	.003
Programming Difficulties: Problems With Particular Components	44.8% (22/49)	48.9% (24/49)	33.3% (10/30)	43.3% (13/30)	21.3% (10/47)	23.4% (11/47)	6.02	7.11	.049	.029

It Was Easy To Implement DP/CBD Approach In My Design			2.90 +- .885	3.03 +- .809	3.66 +- .668	3.77 +- .666	-4.280 (75)	-4.326 (75)	.000	.000
I Find DP/CBD Approach - Intuitive				3.37 +- .718		3.81 +- .851		-2.357 (75)		.021
Used DP/CBD Solution			70% (21/30)	66.7% (20/30)	76.4% (35/47)	87.2% (38/47)	.414	4.706	.350	.031
I Find DP/CBD Approach - Helpful				3.93 +- .640		4.30 +- .507		-2.775 (75)		.007
Design Objective: To Achieve The Form I Originally Sketched	40% (10/25)	48% (12/25)	56.7% (17/30)	60% (18/30)	51% (24/47)	80.8% (38/47)	1.555	8.775	.460	.012
Design Objective: To Explore/Learn Algorithmic Form-Making Process	24% (6/25)	28% (9/25)	63.3% (19/30)	40% (12/30)	46.8% (22/47)	23.4% (11/47)	8.510	2.672	.014	.263
Design Objective: To Experiment With Parameters / Iterations / Variables	8% (2/25)	12% (3/25)	20% (6/30)	46.7% (14/30)	19.1% (9/47)	8.5% (4/47)	1.801	17.800	.406	.000

Comparison of Algorithmic Modelling Criteria

Model Complexity

Model Complexity Score.

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
MODELLING SPEED	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Model Complexity Score	11.73 +- 2.465	13.94 +- 2.585	12.23 +- 2.046	14.10 +- 2.551	12.15 +- 2.246	12.74 +- 2.246	.583 (125)	3.569 (125)	.560	.031

Day 1: F (125) = .583, p = 0.560; F ratio (F), the degrees of freedom (df) and the p-value are used.

Day 2: F (125) = 3.569, p = 0.031;

ANOVA Post-Hoc, Tukey's test

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		p – value NA with DP		p – value NA with CBD		p – value DP with CBD	
MODELLING SPEED	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Model Complexity Score		13.94 +- 2.585		14.10 +- 2.551		12.74 +- 2.246		.960		.062		.065

Categories of Model Complexity

Comparison of Model Complexity categories between the DP and CBD groups:

Criteria	DP (Mean)		CBD (Mean)		t		(df)		p - value	
CATEGORIES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Basic elements	4.53+-.507	4.43 +-.774	4.30 +-.883	4.28 +-.949	1.326	.758	75	75	.189	.451
Composition Space	.80 +-.407	.77 +-.430	.81 +-.398	.70 +-.462	-.091	.614	75	75	.928	.541
Arithmetic of Shapes	.27 +-.691	.43 +-.898	.28 +-.743	.36 +-.735	-.059	.382	75	75	.953	.703
Transformations	2.10 +-.548	2.27 +-.521	2.32 +-.556	2.13 +-.679	-1.697	.955	75	75	.094	.343
Number of Elements	2.40 +-.675	2.67 +-.844	2.38 +-.795	2.17 +-1.049	.097	2.179	75	75	.923	.032
Shape of the Element	1.30 +-.837	1.50 +-.900	1.49 +-.975	1.62 +-1.012	-.877	-.516	75	75	.383	.607
Colour	.83 +-.874	2.03 +-.890	.57 +-.773	1.49 +-1.081	1.361	2.302	75	75	.177	.024

Correlation between Model Complexity and the other criteria. ALL groups:

MODEL COMPLEXITY	Novelty	Variety
DAY 2	DAY 2	DAY 2
Pearson Correlation	.468**	.458**
Sig. (2-tailed)	.000	.000
N	126	126

Correlation between Model Complexity and the other criteria. NA group:

MODEL COMPLEXITY	Variety	Ability To Model Original Idea	MODEL COMPLEXITY	Novelty
DAY 1	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.504**	-.386	Pearson C correlation	.398**
Sig. (2-tailed)	.000	.057	Sig. (2-tailed)	.005
N	49	25	N	49

Correlation between Model Complexity and the other criteria. DP group:

MODEL COMPLEXITY	Algorithm Complexity	Re-Use Of Knowledge	Motivation Satisfaction With Outout	Motivation Satisfaction With Outout	Implemented A Dp/Cbd S-N That Fits	I Find Dp/Cbd Approach Helpful	MODEL COMPLEXITY	Algorithm Complexity	Change Idea, found Interesting Solutions	I Find Dp/Cbd Approach Helpful	Novelty	Variety	Algorithm Complexity
DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.377	-.482**	.463*	.441*	.629**	.355	Pearson Correlation	.413*	-.371*	.385*	.688**	.764**	.797
Sig. (2-tailed)	.040	.007	.010	.015	.000	.054	Sig. (2-tailed)	.023	.044	0.36	.000	.000	.000
N	30	30	30	30	30	30	N	30	30	30	30	30	30

Correlation between Model Complexity and the other criteria. CBD group:

MODEL COMPLEXITY	Ability to accomplish what was wanted	Novelty	Variety
DAY 2	DAY 1	DAY 2	DAY 2
Pearson Correlation	-.359*	.414**	.377**
Sig. (2-tailed)	.013	.004	.009
N	47	47	47

Dependent variable control (Experience / Gender):

Dependent Variable	Approach / p-value		Approach / F (df)		Design Experience / p		Design Experience / F	
Model Complexity Score	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach / Design Experience		.017		(1,67) 5.966		.538		(4,67) .786
	Approach / p-value		Approach / F		Gender / p		Gender / F	
Approach / Gender		.019		(1, 73) 5.797		.146		(1, 73) .704

Algorithm Complexity

Algorithm Complexity Score

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
ALGORITHM COMPLEXITY	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Algorithm Complexity Score	40.69 +- 18.275	54.61 +- 26.988	50.60 +- 33.14	56.57 +- 28.22	50.40 +- 30.11	53.59 +- 27.48	2.025 (125)	.107 (125)	.136	.898

Day 1: F (125) = 2.025, p = 0.136; F ratio (F), the degrees of freedom (df) and p-value are used.

Day 2: F (125) = .107, p = 0.898;

Categories Of Programming Components Implemented

Comparison of implemented components by category (input tubs) between the DP and CBD groups:

Criteria	DP (Mean)		CBD (Mean)		t		df		p - value	
COMPONENTS COMPLEXITY	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
0 - 1 Input Comp.	6.23 +- 1.675	6.67 +- 2.218	5.38 +- 1.895	5.96 +- 1.574	2.007	1.641	75	75	.048	.105
2 Input Comp	4.83 +- 1.895	6.80 +- 2.188	3.79 +- 1.473	6.02 +- 2.202	2.714	1.517	75	75	.008	.133
3 Input Comp.	3.47 +- 2.193	2.43 +- 1.455	2.57 +- 1.331	2.43 +- 1.347	2.005	.024	42.750	75	.051	.981
4 Input Comp.	.50 +- .820	1.57 +- .774	.85 +- .978	1.23 +- .937	-1.633	1.621	75	75	.107	.109
5 Input Comp.	.00	.13 +- .346	.02 +- .146	.09 +- .282	-.797	.670	75	75	.428	.505
6 Input Comp.	.10 +- .305	.00	.15 +- .360	.04 +- .204	-.616	-1.430	75	46.000	.540	.160

Correlation between Programming Algorithm Complexity and the other criteria. All groups:

ALGORITHM COMPLEXITY	Variety	Algorithm Complexity	ALGORITHM COMPLEXITY	Algorithm Complexity	Novelty	Variety
DAY 1	DAY 1	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2
Pearson Correlation	.498**	.401**	Pearson Correlation	.401**	.458**	.599**
Sig. (2-tailed)	.000	.000	Sig. (2-tailed)	.000	.000	.000
N	126	126	N	126	126	126

No Approach group:

ALGORITHM COMPLEXITY	Variety	Algorithm Complexity	ALGORITHM COMPLEXITY	Novelty	Variety	Algorithm Complexity	Satisfaction With Output	Novelty	Variety
DAY 1	DAY 1	DAY 2	DAY 2	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2
Pearson Correlation	.610**	.525**	Pearson Correlation	.478**	.352*	.525**	.363*	.614**	.675**
Sig. (2-tailed)	.000	.000	Sig. (2-tailed)	.001	.013	.000	.010	.000	.000
N	49	49	N	49	49	49	49	49	49

DP group:

ALGORITHM COMPLEXITY	Model Complexity	Novelty	Variety	Utility Approach Is Helpful	Model Complexity	Algorithm Complexity	ALGORITHM COMPLEXITY	Algorithm Complexity	Utility Approach Is Helpful	Model Complexity	Novelty	Variety
DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.377*	.403*	.511**	.434*	.413*	.374*	Pearson Correlation	.374*	.361*	.797	.583	.795
Sig. (2-tailed)	.040	.027	.004	.017	.023	.042	Sig. (2-tailed)	.042	.050	.000	.001	.000
N	30	30	30	30	30	30	N	30	30	30	30	30

CBD group:

ALGORITHM COMPLEXITY	Design Objectives Accomplish What	Variety	Variety	Algorithm Complexity	ALGORITHM COMPLEXITY	Algorithm Complexity	Design Objectives Accomplish What
DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 1	DAY 2
Pearson Correlation	-.362	.432	.363	.383	Pearson Correlation	.383	-.378
Sig. (2-tailed)	.013	.002	.012	.008	Sig. (2-tailed)	.008	.009
N	47	47	47	47	N	47	47

Explored Solution Space

Variety

Variety Score

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
VARIETY SCORE	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
ALGORITHM VARIETY Score	12.43 +- 3.565	16.65 +- 5.851	15.13 +- 4.718	17.60 +- 5.137	12.77 +- 3.595	15.77 +- 3.218	4.992 (125)	1.332 (125)	.008	.268

Day 1: F (125) = 4.992, p = 0.008; F ratio (F), the degrees of freedom (df) and p-value are used.

Day 2: F (125) = 1.332, p = 0.268;

ANOVA Post-Hoc, Tukey's test

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		p – value NA with DP		p – value NA with CBD		p – value DP with CBD	
VARIETY SCORE	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
ALGORITHM VARIETY Score	12.43 +- 3.565		15.13 +- 4.718		12.77 +- 3.595		.009		.905		.027	

Correlation between Algorithm (Programming Solution) Variety and the other criteria. All groups:

VARIETY	Novelty	Algorithm Complexity	VARIETY	Model Complexity	Novelty	Algorithm Complexity
DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.698**	.498**	Pearson Correlation	.458**	.766**	.599**
Sig. (2-tailed)	.000	.000	Sig. (2-tailed)	.000	.000	.000
N	126	126	N	126	126	126

No Approach group:

VARIETY	Ability To Model Original Idea	Satisfaction With The Output	Model Complexity	Novelty	Algorithm Complexity	Algorithm Complexity	VARIETY	Novelty	Novelty	Algorithm Complexity
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2
Pearson Correlation	.480*	.406**	.504**	.687**	.610**	.352*	Pearson Correlation	.471**	.809**	.675**
Sig. (2-tailed)	.015	.004	.000	.000	.000	.013	Sig. (2-tailed)	.001	.000	.000
N	25	49	49	49	49	49	N	49	49	49

DP group:

VARIETY	Ability To Accomplish What Was Wanted	Algorithm Complexity	Novelty	Approach Helpful	VARIETY	Model Complexity	Novelty	Algorithm Complexity
DAY 1	DAY 1	DAY 1	DAY 1	DAY2	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.350	.511**	.794	.357	Pearson Correlation	.764	.777	.795
Sig. (2-tailed)	.058	.004	.000	.053	Sig. (2-tailed)	.000	.000	.000
N	30	30	30	30	N	30	30	30

CBD group:

VARIETY	Novelty	Variety	Algorithm Complexity	Novelty	VARIETY	Novelty	Model Complexity	Variety	Algorithm Complexity
DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 1	DAY 1
Pearson Correlation	.679	.544	.432	.374**	Pearson Correlation	.613	.377**	.544	.363*
Sig. (2-tailed)	.000	.000	.002	.010	Sig. (2-tailed)	.000	.009	.000	.012
N	47	47	47	47	N	47	47	47	47

Dependent variable control (Experience / Gender):

Dependent Variable	Approach / p-value		Approach / F (df)		Design Experience / p		Design Experience / F	
TOTAL VARIETY SCORE	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach / Design Experience	.032	.134	(1,67) 4.797	(1,67) 2.304	.739	.495	(4,67) .496	(4,67) .856
	Approach / p-value		Approach / F		Gender / p		Gender / F	
Approach / Gender	.005	.056	(1,73) 8.441	(1,73) 3.760	.003	.575	(1,73) 9.526	(1,73) .318

Descriptive Statistics

Dependent Variable: DAY 1 Total Variety Score

Gender	DP Mean	DP Std. Deviation	DP N	CBD Mean	CBD Std. Deviation	CBD N	Total Mean	Total Std. Deviation	Total N
Male	16.87	4.984	15	13.57	3.510	30	14.67	4.301	45
Female	13.40	3.851	15	11.35	3.390	17	12.31	3.702	32
Total	15.13	4.718	30	12.77	3.595	47	13.69	4.203	77

Descriptive Statistics

Dependent Variable: DAY 2 Total Variety Score

Design Experience Groups	DP Mean	DP Std. Deviation	DP N	CBD Mean	BCD Std. Deviation	CBD N	Total Mean	Total Std. Deviation	Total N
0. - 1.9 years of experience	19.18	5.671	11	16.25	3.793	12	17.65	4.905	23
2.0 - 3.9 years of experience	15.57	3.207	7	15.47	3.623	15	15.50	3.419	22
4.0 - 5.9 years of experience	17.37	6.739	8	15.94	2.645	18	16.38	4.234	26
6.0 - 7.9 years of experience	17.00	1.414	2	13.00	.	1	15.67	2.517	3
8 or more years of experience	17.50	2.121	2	14.00	.	1	16.33	2.517	3
Total	17.60	5.137	30	15.77	3.218	47	16.48	4.141	77

Novelty

Novelty Score

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
NOVELTY SCORE	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
ALGORITHM NOVELTY Score	28.16 +- 16.697	50.82 +- 31.646	29.30 +- 19.193	53.67 +- 20.860	27.43 +- 16.562	43.57 +- 17.820	.108 (125)	1.791 (125)	.898	.171

Day 1: F (125) = 0.108, p = 0.898; F ratio (F), the degrees of freedom (df) and p-value are used.

Day 2: F (125) = 1.791, p = 0.171;

Novelty Categories of Implemented Components

Comparison between the DP and CBD groups

Criteria	DP (Mean)		CBD (Mean)		t		df		p - value	
NOVELTY CATEGORIES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
0 Novelty Points Comp.	21.87 +- 15.538	22.80 +- 13.850	21.45 +- 15.423	20.83 +- 12.908	.116	.635	75	75	.908	.527
1 Novelty Points Comp	1.87 +- 2.300	1.57 +- 1.305	1.53 +- 2.578	1.51 +- 1.586	.579	.162	75	75	.564	.872
2 Novelty Points Comp	1.07 +- 1.258	.93 +- 1.112	1.47 +- 2.063	1.21 +- 1.382	-1.061	-.931	74.891	75	.292	.355
3 Novelty Points Comp	1.27 +- 1.530	1.23 +- 1.040	.72 +- .994	1.17 +- 1.167	1.727	.241	44.675	75	.091	.810
4 Novelty Points Comp	1.10	.50	.77	.79	1.207	-1.195	75	75	.231	.236

	+- 1.185	+- .777	+- 1.183	+- 1.160						
5 Novelty Points Comp	.50 +-.900	1.97 +- 1.671	.85 +- 1.335	1.53 +- 1.653	-1.267	1.121	75	75	.209	.266
6 Novelty Points Comp	.53 +-.776	1.00 +- 1.017	.94 +- 1.275	1.26 +- 1.421	-1.554	-.853	75	75	.124	.396
7 Novelty Points Comp	.90 +- 1.185	1.37 +- 1.129	.79 +- 1.122	.79 +- 1.020	.421	2.331	75	75	.675	.022
8 Novelty Points Comp	1.27 +- 2.638	1.03 +- 1.189	.47 +-.654	.77 +- .983	1.626	1.072	31.292	75	.114	.287
9 Novelty Points Comp	.13 +-.346	1.13 +- 1.548	.19 +-.495	1.13 +- 1.825	-.561	.014	75	75	.576	.989
10 Novelty Points Comp	.27 +-.691	.90 +- 1.423	.51 +- 1.120	.72 +- 1.192	- 1.181	.587	74.94 7	75	.241	.559

Correlation between Algorithm (Programming Solution) Novelty and the other criteria. All groups:

NOVELTY	Variety	Novelty	NOVELTY	Model Complexity	Variety	Algorithm Complexity
DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.698**	.350**	Pearson Correlation	.468**	.766**	.458**
Sig. (2-tailed)	.000	.000	Sig. (2-tailed)	.000	.000	.000
N	126	126	N	126	126	126

No Approach group:

NOVELTY	Variety	Variety	Algorithm Complexity	Novelty	NOVELTY	Novelty	Model Complexity	Variety	Algorithm Complexity
DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.687**	.471**	.478**	.451**	Pearson Correlation	.451**	.398**	.809**	.614**
Sig. (2-tailed)	.000	.001	.001	.001	Sig. (2-tailed)	.001	.005	.000	.000
N	49	49	49	49	N	49	49	49	49

DP group:

NOVELTY	Variety	Algorithm Complexity	Programming Difficulties How Often	NOVELTY	Ability To Accomplish The Original Idea	Model Complexity	Variety	Algorithm Complexity
DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.794	.403*	-.374*	Pearson Correlation	.379*	.688	.777	.583
Sig. (2-tailed)	.000	.027	.041	Sig. (2-tailed)	.039	.000	.000	.001
N	30	30	30	N	30	30	30	30

CBD group:

NOVELTY	Changed the design idea, because you discovered new solutions	Variety	NOVELTY	Variety	Model Complexity	Variety
DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2
Pearson Correlation	-.380**	.679**	Pearson Correlation	.374**	.414**	.613**
Sig. (2-tailed)	.009	.000	Sig. (2-tailed)	.010	.004	.000
N	47	47	N	47	47	47

Dependent variable control (Experience / Gender):

Dependent Variable	Approach / p-value		Approach / F (df)		Design Experience / p		Design Experience / F	
TOTAL NOVELTY SCORE	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach / Design Experience		.145		(1,67) 2.178		.973		(4,67) 47.251
	Approach / p-value		Approach / F		Gender / p		Gender / F	
Approach / Gender		.024		(1,73) 5.333		.462		(1,73) .548

Descriptive Statistics

Dependent Variable: DAY 2 Total Novelty Score

Design Experience Groups	DP Mean	DP Std. Deviation	DP N	CBD Mean	CBD Std. Deviation	CBD N	Total Mean	Total Std. Deviation	Total N
0. - 1.9 years of experience	58.91	23.763	11	39.75	20.951	12	48.91	23.914	23
2.0 - 3.9 years of experience	49.00	17.117	7	46.53	17.517	15	47.32	17.019	22
4.0 - 5.9 years of experience	51.63	17.246	8	44.17	16.100	18	46.46	16.488	26
6.0 - 7.9 years of experience	61.50	19.092	2	20.00	.	1	47.67	27.502	3
8 or more years of experience	41.50	43.134	2	58.00	.	1	47.00	31.953	3
Total	53.67	20.860	30	43.57	17.820	47	47.51	19.565	77

Comparison of Programming Criteria

Programming Difficulties

How Often You Come Across Programming Difficulties

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
PROGRAMMING DIFFICULTIES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
How often you have come across programming difficulties	2.88 +- 1.053	2.71 +- .890	2.37 +- .669	2.10 +- .403	2.91 +- 1.039	2.53 +- .776	3.414 (125)	6.200 (125)	.036	.003

Day 1: F (125) = 3.414, p = .036; F ratio (F), the degrees of freedom (df) and the p-value are used.

Day 2: F (125) = 6.200, p = .003;

ANOVA Post-Hoc, Tukey's test

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		p – value NA with DP		p – value NA with CBD		p – value DP with CBD	
PROGRAMMING DIFFICULTIES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
How often you have come across programming difficulties	2.88 +- 1.053	2.71 +- .890	2.37 +- .669	2.10 +- .403	2.91 +- 1.039	2.53 +- .776	.064	.002	.981	.467	.045	.042

Types of Difficulties

Chi-square test Comparison between the all tree groups: NA, DP and CBD

Criteria	No Approach Count / Total (%)		DP Count / Total (%)		CBD Count / Total (%)		X^2		p – value	
TYPES OF DIFFICULTIES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Problems with Particular components	**44.8%** (22/49)	**48.9%** (24/49)	**33.3%** (10/30)	**43.3%** (13/30)	**21.3%** (10/47)	**23.4%** (11/47)	6.023	7.112	**.049**	**.029**
Logic Connections	**18.3%** (9/49)	**20.4%**	**30%** (9/30)	**23.3%** (7/30)	**25.5%** (12/47)	**23.4%** (11/47)	1.511	.153	**.470**	**.926**
Knowing what component to use	**30.6%** (15/49)	**24.5%** (12/49)	**26.7%** (8/30)	**20%** (6/30)	**38.3%** (18/47)	**34%** (16/47)	1.264	2.086	**.531**	**.352**
Valid Parameters	**12.2%** (6/49)	**12.2%** (6/49)	**13.3%** (4/30)	**16.7%** (5/30)	**17%** (8/47)	**10.6%** (5/47)	.476	.615	**.788**	**.735**
Idea to Algorithm translation	**44.9%** (22/49)	**42.8%** (21/49)	**53.3%** (16/30)	**60%** (18/30)	**48.9%** (23/47)	**53.2%** (25/47)	.538	2.360	**.764**	**.307**

Problems with Particular components:

Day 1: NA 22/49 (44.8%), DP 10/30 (33.3%), CBD 10/47 (21.3%), X2 = 6.023, p = .049, the count of responses, the percentage, the Chi-Square – value (X2) and the p-value are used.

Day 2: NA 22/49 (48.9%), DP 12/30 (43.3%), CBD 11/47 (23.4%), X2 = 7.112, p = .029,

Chi-square test Comparison between the DP and CBD groups

Criteria	DP (yes/30)		DP (%)		CBD (yes/47)		CBD (%)		X^2		p - value	
TYPES OF DIFFICULTIES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Problems with Particular components	10	13	33.3	43.3	10	11	21.3	23.4	1.384	3.390	.239	.066
Logic Connections	9	7	30	23.3	12	11	25.5	23.4	.184	.000	.668	.994
Knowing what component to use	8	6	26.7	20	18	16	38.3	34	1.108	1.769	.293	.183
Valid Parameters	4	5	13.3	16.7	8	5	17	10.6	.189	.589	.663	.443
Idea to Algorithm translation	16	18	53.3	60	23	25	48.9	53.2	.142	.344	.707	.557
Problems with the approach	0	4	0	13.3	0	1	0	2.1		3.787		.052

Chi-square test Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach (yes/ 49)		No Approach (%)		DP (%)		CBD (%)		p – value between N/A and DP group		p – value between N/A and CBD group	
TYPES OF DIFFICULTIES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Problems with Particular components	22	24	44.8	48.9	33.3	43.3	21.3	23.4	.218	.400	.012	.008
Logic Connections	9	10	18.3	20.4	30	23.3	25.5	23.4	.178	.485	.274	.457
Knowing what component to use	15	12	30.6	24.5	26.7	20	38.3	34	.456	.431	.282	.211
Valid Parameters	6	6	12.2	12.2	13.3	16.7	17	10.6	.573	.407	.354	.530
Idea to Algorithm translation	22	21	44.9	42.8	53.3	60	48.9	53.2	.310	.106	.424	.209

DEPENDENCY BETWEEN THE TYPES OF DIFFULTIES AND THE OVERALL AMOUNT OF PROBLEMS

All test groups

Programming difficulties		1. No Difficulties		2. 1- 3 Problems		3. 4 – 6 Problems		4. 7 – 9 Problems		5. 10 > Problems		p – value between YES / NO group	
TYPES OF DIFFICULTIES		DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Problems with Particular components	NO	4	2	30	44	26	20	15	10	9	2	.029	.711
	YES	1	1	25	31	13	12	3	4	1	0		
Logic Connections	NO	4	2	41	63	30	23	13	8	8	2	.896	.161
	YES	1	1	14	12	9	9	5	6	1	0		
Knowing what component to use	NO	3	3	42	60	25	22	9	6	6	1	.307	.036
	YES	2	0	13	15	14	10	9	8	3	1		
Valid Parameters	NO	4	3	47	67	36	28	16	10	5	2	.079	.381
	YES	1	0	8	8	3	4	2	4	4	0		
Idea to Algorithm translation	NO	3	2	32	34	17	15	10	10	3	1	.491	.455
	YES	2	1	23	41	22	17	8	4	6	1		

No Approach group

Programming difficulties		1. No Difficulties		2. 1- 3 Problems		3. 4 – 6 Problems		4. 7 – 9 Problems		5. 10 > Problems		p – value between YES / NO group	
TYPES OF DIFFICULTIES		DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Problems with Particular components	NO	1	1	7	8	7	10	8	5	4	1	.060	.512
	YES	1	1	13	13	6	6	2	4	0	0		
Logic Connections	NO	2	1	16	20	11	13	8	4	3	1	.952	.023
	YES	0	1	4	1	2	3	2	5	1	0		
Knowing what component to use	NO	2	2	16	19	7	12	6	4	3	0	.420	.027
	YES	0	0	4	2	6	4	4	5	1	1		
Valid Parameters	NO	2	2	18	19	12	14	9	7	2	1	.201	.845
	YES	0	0	2	2	1	2	1	2	2	0		
Idea to Algorithm translation	NO	1	2	14	10	7	7	5	8	0	1	.145	.095
	YES	1	0	6	11	6	9	5	1	4	0		

DP group

Programming difficulties		1. No Difficulties		2. 1- 3 Problems		3. 4 – 6 Problems		4. 7 – 9 Problems		5. 10 > Problems		p – value between YES / NO group	
TYPES OF DIFFICULTIES		DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Problems with Particular components	NO	2	1	9	13	8	3	1	-	-	-	.482	.464
	YES	0	0	7	12	3	1	0	-	-	-		
Logic Connections	NO	1	1	11	20	8	2	1	-	-	-	.835	.359
	YES	1	0	5	5	3	2	0	-	-	-		
Knowing what component to use	NO	1	1	13	21	7	2	1	-	-	-	.580	.253
	YES	1	0	3	4	4	2	0	-	-	-		
Valid Parameters	NO	1	1	15	21	10	3	0	-	-	-	.021	.815
	YES	1	0	1	4	1	1	1	-	-	-		
Idea to Algorithm translation	NO	2	0	8	10	4	2	0	-	-	-	.296	.659
	YES	0	1	8	15	7	2	1	-	-	-		

CBD group

Programming difficulties		1. No Difficulties		2. 1- 3 Problems		3. 4 – 6 Problems		4. 7 – 9 Problems		5. 10 > Problems		p – value between YES / NO group	
TYPES OF DIFFICULTIES		DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Problems with Particular components	NO	1	-	14	23	11	7	6	5	5	1	.667	.242
	YES	0	-	5	6	4	5	1	0	0	0		
Logic Connections	NO	1	-	14	23	11	8	4	4	5	1	.528	.773
	YES	0	-	5	6	4	4	3	1	0	0		
Knowing what component to use	NO	0	-	13	20	11	8	2	2	3	1	.192	.545
	YES	1	-	6	9	4	4	5	3	2	0		
Valid Parameters	NO	1	-	14	27	14	11	7	3	3	1	.214	.161
	YES	0	-	5	2	1	1	0	2	2	0		
Idea to Algorithm translation	NO	0	-	10	14	6	6	5	2	3	0	.538	.790
	YES	1	-	9	15	9	6	2	3	2	1		

Correlation between amount of programming difficulties and the other criteria. All groups:

PROGRAMMING DIFFICULTIES	Programming Difficulties How Often	Change In Design Intent Programming Difficulties	PROGRAMMING DIFFICULTIES	Programming Difficulties
DAY 1	DAY 2	DAY 1	DAY 2	Day 1
Pearson Correlation	.385**	.371**	Pearson Correlation	.385**
Sig. (2-tailed)	.000	.000	Sig. (2-tailed)	.000
N	126	126	N	126

No Approach group:

PROGRAMMING DIFFICULTIES	Change In Design Intent Programming Difficulties	Re-Used Algorithms How Often	Programming Difficulties	PROGRAMMING DIFFICULTIES	Programming Difficulties	Ability To Accomplish To Accomplish What Was Wanted	Ability To Model Original Idea	Change In Design Intent Programming Difficulties	Motivation Satisfaction With Output
DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.371**	.449**	.406**	Pearson Correlation	.406**	-.408**	-.426*	.456**	-.400**
Sig. (2-tailed)	.009	.001	.004	Sig. (2-tailed)	.004	.004	.034	.001	.004
N	49	49	49	N	49	49	25	49	49

DP group:

PROGRAMMING DIFFICULTIES	Usability Easy To Implement	Programming Difficulties	Change In Design Intent Programming Difficulties	Change In Design Intent Programming Difficulties	Motivation To Use Algorithmic In Future	PROGRAMMING DIFFICULTIES	Change In Design Intent Programming Difficulties	Programming Difficulties	Easy To Implement Approach Usability	Novelty	Change In Design Intent Programming Difficulties
DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2
Pearson Correlation	-.577**	.371*	.420*	.365*	-.490**	Pearson Correlation	.435*	.371*	-.358	-.374*	.446*
Sig. (2-tailed)	.001	.043	.021	.048	.006	Sig. (2-tailed)	.016	.043	.052	.041	.014
N	30	30	30	30	30	N	30	30	30	30	30

CBD group:

PROGRAMMING DIFFICULTIES	Ability To Accomplish What Was Wanted	Ability To Realise Original Idea	Change In Design Intent Programming Difficulties	Motivation Satisfaction With Outcome
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1
Pearson Correlation	-.350*	-.503**	.353*	-.357*
Sig. (2-tailed)	.016	.000	.015	.014
N	47	47	47	47

Dependent variable control (Experience / Gender):

Dependent Variable	Approach / p-value		Approach / F(df)		Design Experience / p		Design Experience / F	
How often: program. difficulties	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach / Design Experience	.011	.180	(1,67) 6.930	(1,67) 1.836	.601	.536	(4,67) .690	(4,67) .790
	Approach / p-value		Approach / F		Gender / p		Gender / F	
Approach / Gender	.012	.014	(1,73) 6.664	(1,73) 6.351	.880	.496	(1,73) .023	(1,73) .469

Descriptive Statistics

Dependent Variable: DAY 2. How often you have come across programming difficulties

Design Experience Groups	DP Mean	DP Std. Deviation	DP N	CBD Mean	CBD Std. Deviation	CBD N	Total Mean	Total Std. Deviation	Total N
0. - 1.9 years of experience	2.18	.405	11	2.67	.651	12	2.43	.590	23
2.0 - 3.9 years of experience	2.29	.488	7	2.33	.617	15	2.32	.568	22
4.0 - 5.9 years of experience	2.00	.000	8	2.67	.970	18	2.46	.859	26
6.0 - 7.9 years of experience	1.50	.707	2	2.00	.	1	1.67	.577	3
8 or more years of experience	2.00	.000	2	2.00	.	1	2.00	.000	3
Total	2.10	.403	30	2.53	.776	47	2.36	.687	77

Learning Curve

How Often Participants Have Implemented New Components

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
LEARNING CURVE	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
How often participants have implemented new programming components	2.16 +- 1.143	2.16 +- .986	2.43 +- 1.135	1.87 +- .819	2.21 +- 1.122	2.09 +- .830	.561 (99.4)	1.051 (99.4)	.572	.353

Correlation between how often participants implemented new components and the other criteria.

All groups:

LEARNING CURVE	Learning Curve	LEARNING CURVE	Learning Curve
DAY 1	DAY 2	DAY 2	DAY 1
Pearson Correlation	.366**	Pearson Correlation	.366**
Sig. (2-tailed)	.000	Sig. (2-tailed)	.000
N	126	N	126

No Approach group:

LEARNING CURVE	Ability to model Original Idea
DAY 1	DAY 1
Pearson Correlation	.400*
Sig. (2-tailed)	.047
N	25

DP group:

LEARNING CURVE	Learning Curve	LEARNING CURVE	Learning Curve
DAY 1	DAY 2	DAY 2	DAY 1
Pearson Correlation	.658**	Pearson Correlation	.658**
Sig. (2-tailed)	.000	Sig. (2-tailed)	.000
N	30	N	30

CBD group:

LEARNING CURVE	Implemented a DP/CBD solution that	LEARNING CURVE	Ability to accomplish what was wanted	Motivation To Use Algorithmic In Future
DAY 1	DAY 1	DAY 2	DAY 1	DAY 2
Pearson Correlation	.359*	Pearson Correlation	.348*	.457**
Sig. (2-tailed)	.013	Sig. (2-tailed)	.017	.001
N	47	N	47	47

Re-Use of Algorithms

Re-Use Of Knowledge

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
RE-USE OF KNOWLEDGE	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
How often participants have re-used algorithms from the external sources	1.98 +- .968	2.31 +- .962	2.37 +- .928	2.50 +- .682	2.32 +- .980	2.34 +- .867	2.09 1 (118)	.496 (93.2)	.128	.610

Correlation between how often participants have re-used algorithms from the external sources and the other criteria. All groups:

RE-USE OF KNOWLEDGE	Re-Use Of Knowledge	RE-USE OF KNOWLEDGE	Re-Use Of Knowledge
DAY 1	DAY 2	DAY 2	DAY 1
Pearson Correlation	.428**	Pearson Correlation	.428**
Sig. (2-tailed)	.000	Sig. (2-tailed)	.000
N	126	N	126

No Approach group:

RE-USE OF KNOWLEDGE	Programming Difficulties
DAY 2	DAY 1
Pearson Correlation	.449**
Sig. (2-tailed)	.001
N	49

DP group:

RE-USE OF KNOWLEDGE	Change in Design due to difficulties	Re-Use Of Knowledge	RE-USE OF KNOWLEDGE	Re-Use Of Knowledge	Model Complexity	Motivation Satisfaction With Output
DAY 1	DAY 2	DAY 2	DAY 2	DAY 1	DAY 1	DAY 1
Pearson Correlation	-.382*	.681**	Pearson Correlation	.681**	-.482**	-.380*
Sig. (2-tailed)	.037	.000	Sig. (2-tailed)	.000	.007	.038
N	30	24	N	30	30	30

CBD group:

RE-USE OF KNOWLEDGE	Re-Use Of Knowledge	RE-USE OF KNOWLEDGE	Re-Use Of Knowledge
DAY 1	DAY 2	DAY 2	DAY 1
Pearson Correlation	.535**	Pearson Correlation	.535**
Sig. (2-tailed)	.000	Sig. (2-tailed)	.000
N	47	N	47

Comparison of Approach Characteristics Criteria

Usability

Usability

T-test. Comparison between the DP and CBD groups

Criteria	DP (Mean)		CBD (Mean)		t		df		p - value	
USABILITY	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
It was easy to implement DP/CBD approach in my design.	2.90 +- .885	3.03 +- .809	3.66 +- .668	3.77 +- .666	-4.280	-4.326	75	75	.000	.000

Day 1: t(75) = -4.280, p = 0.000; the t-value (t), the degrees of freedom (df) and the p-value

Day 2: t(75) = -4.326, p = 0.000;

Correlation between approach usability and the other criteria. DP group:

USABILITY easy to implement	Ability To Accomplish What Wanted	Change In Design Intent Programming Difficulties	Programming Difficulties	Motivation Satisfaction With Output	Flexibility Found Solution Which Fits	Change In Design Intent Found New Solutions	Programming Difficulties	Motivation Satisfaction With Output	Flexibility Found Solution Which Fits	Usability Easy To Implement	USABILITY easy to implement	Usability Easy To Implement	Motivation Satisfaction With Output	Flexibility Found Solution Which Fits
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2
Pearson Correlation	.486	-.480	-.577	.577	.395	.350	-.358	.434	.430*	.583	Pearson Correlation	.583	.462	.459
Sig. (2-tailed)	.006	.007	.001	.001	.031	.058	.052	.017	.018	.001	Sig. (2-tailed)	.001	.010	.011
N	30	30	30	30	30	30	30	30	30	30	N	30	30	30

Correlation between approach usability and the other criteria. CBD group:

USABILITY easy to implement	Design Objectives Accomplish What Wanted	Change In Design Intent Found New Solutions	Design Objectives Accomplish What Wanted	Motivation Satisfaction With Output	Usability Easy To Implement	Motivation Satisfaction With Output	Utility Approach Is Helpful	USABILITY easy to implement	Usability Easy To Implement	Design Objectives Accomplish What Wanted	Motivation Satisfaction With Output	Flexibility Found Solution Which Fits
DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.627**	.397**	.408**	.495**	.354*	.409**	.434**	Pearson Correlation	.354*	.358*	.415**	.400**
Sig. (2-tailed)	.000	.006	.004	.000	.015	.004	.002	Sig. (2-tailed)	.015	.013	.004	.005
N	47	47	47	47	47	47	47	N	47	47	47	47

Dependent variable control (Experience / Gender):

Dependent Variable	Approach / p-value		Approach / F (df)		Design Experience / p		Design Experience / F	
It was easy to implement DP/CBD approach in my design	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach / Design Experience	.021	.022	(1,67) 5.610	(1,67) 5.465	.675	.793	(4,67) .585	(4,67) .421
	Approach / p-value		Approach / F		Gender / p		Gender / F	
Approach / Gender	.000	.000	(1,73) 17.272	(1,73) 17.646	.548	.845	(1,73) .364	(1,73) .039

Intuitiveness

Intuitiveness

Comparison between the DP and CBD groups

Criteria	DP (Mean)		CBD (Mean)		t		df		p - value	
INTUITIVENESS	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
I find DP/CBD approach - intuitive.		3.37 +- .718		3.81 +- .851		-2.357		75		.021

t(75) = -2.357, p = 0.021; the t-value (t), the degrees of freedom (df) and the p-value

Correlation between approach intuitiveness and the other criteria. DP group:

INTUITIVNESS	Utility Approach Is Helpful
	DAY 2
Pearson Correlation	.355
Sig. (2-tailed)	.054
N	30

CBD group:

INTUITIVNESS	Utility Approach Is Helpful
	DAY 2
Pearson Correlation	.438**
Sig. (2-tailed)	.002
N	47

Dependent variable control (Experience / Gender):

Dependent Variable	Approach / p-value		Approach / F (df)		Design Experience / p		Design Experience / F	
INTUITIVNESS	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach / Design Experience		.562		(1,67) .339		.552		(4,67) .765
	Approach / p-value		Approach / F		Gender / p		Gender / F	
Approach / Gender		.024		(1,73) 5.352		.563		(1,73) .337

Descriptive Statistics

Dependent Variable: DAY 2 INTUITIVNESS

Design Experience Groups	DP Mean	DP Std. Deviation	DP N	CBD Mean	CBD Std. Deviation	CBD N	Total Mean	Total Std. Deviation	Total N
0. - 1.9 years of experience	3.45	.820	11	3.92	.996	12	3.70	.926	23
2.0 - 3.9 years of experience	3.29	.756	7	3.60	.986	15	3.50	.913	22
4.0 - 5.9 years of experience	3.38	.744	8	4.00	.594	18	3.81	.694	26
6.0 - 7.9 years of experience	3.00	.000	2	3.00	.	1	3.00	.000	3
8 or more years of experience	3.50	.707	2	3.00	.	1	3.33	.577	3
Total	3.37	.718	30	3.81	.851	47	3.64	.826	77

Flexibility

Flexibility

Comparison between the DP and CBD groups

Criteria	DP (Mean)		CBD (Mean)		t		df		p - value	
FLEXIBILITY	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
I have successfully found / implemented a DP/Case-Base solution that fits my design idea.	3.47 +- .730	3.80 +- .761	3.66 +- .562	3.64 +- .735	-1.305	.929	75	75	.196	.356

Day 1: t(75) = -1.305, p = 0.196; the t-value (t), the degrees of freedom (df) and the p-value

Day 2: t(75) = 0.929, p = 0.356;

Used DP/CBD solutions [from the documented Design Patterns / Online Case-Base]. Comparison between the DP and CBD groups

Criteria	DP (yes/30l)		DP(%)		CBD (yes/47l)		CBD(%)		X^2		p - value	
FLEXIBILITY	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Used DP/CBD solution	17	18	56.7 %	60 %	35	38	74.4 %	80.8 %	2.647	4.014	.085	.042

Day 1: DP 17/30 (56.7%), CBD 35/47 (74.4%), X2 = 2.647, p = 0.085. Chi-Square – value (X2)

Day 2: DP 18/30 (60%), CBD 38/47 (80.8%), X2 = 4.014, p = 0.042.

Used DP/CBD solution [from the documented Design Patterns and patterns for which participants used different names / On-line case-Base and cases from tutorials]. Comparison between the DP and CBD groups

Criteria	DP (yes/30l)		DP(%)		CBD (yes/47l)		CBD(%)		X^2		p - value	
FLEXIBILITY	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Used DP/CBD solutions (from the Case-base and from tutorials)	**21**	**20**	**70 %**	**66.7 %**	**36**	**41**	**76.6 %**	**87.2 %**	.414	4.706	**.350**	**.031**

Day 1: DP 21/30 (70%), CBD 36/47 (76.6%), X2 = 0.414, p = 0.350.

Day 2: DP 20/30 (66.7%), CBD 41/47 (87.2%), X2 = 4.706, p = 0.031.

Correlation between approach flexibility and the other criteria. DP group:

FLEXIBILITY found solution which fits	Design Objectives Accomplish What	Motivation Satisfaction With Output	Change In Design Intent Difficulties	Usability Easy To Implement	FLEXIBILITY found solution which fits	Usability Easy To Implement	Motivation Satisfaction With Output	Motivation Satisfaction With Output	Usability Easy To Implement	Utility Approach Helpful	Model Complexity
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 1
Pearson Correlation	.432*	.382*	-.403*	.395*	Pearson Correlation	.430*	.485**	.600**	.459*	.397*	.629**
Sig. (2-tailed)	**.017**	**.037**	**.027**	**.031**	Sig. (2-tailed)	**.018**	**.007**	**.000**	**.011**	**.030**	**.000**
N	30	30	30	30	N	30	30	30	30	30	30

Correlation between approach flexibility and the other criteria. CBD group:

FLEXIBILITY found solution which fits	Motivation Satisfaction With Output	Learning Curve Implemented A New	Change In Design Intent Difficulties	FLEXIBILITY found solution which fits	Design Objectives Accomplish What	Motivation Satisfaction With Output	Usability Easy To Implement
DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.495**	.359*	-.387**	Pearson Correlation	.604**	.372*	.400**
Sig. (2-tailed)	**.000**	**.013**	**.007**	Sig. (2-tailed)	**.000**	**.010**	**.005**
N	47	47	47	N	47	47	47

Dependent variable control (Experience / Gender):

Dependent Variable	Approach / p-value		Approach / F (df)		Design Experience / p		Design Experience / F	
Used DP/CBD	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach / Design Experience	.026	.007	(1,58) 5.200	(1,62) 7.651	.534	.774	(4,58) .793	(4,62) .447
	Approach / p-value		Approach / F		Gender / p		Gender / F	
Approach / Gender	.001	.000	(1,64) 13.077	(1,68) 14.206	.472	.662	(1,64) .523	(1,68) .193

Utility

Approach Helpful

Comparison between the DP and CBD groups

Criteria	DP (Mean)		CBD (Mean)		t		df		p - value	
UTILITY	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
I find DP/CBD approach - helpful.		3.93 +- .640		4.30 +- .507		-2.775		75		.007

Day 1: t(75) = -2.775, p = 0.007; the t-value (t), the degrees of freedom (df) and the p-value

Correlation between how helpful is each approach and the other criteria.

DP group:

UTILITY approach is helpful	Algorithm Complexity	Variety	Model Complexity	Motivation Satisfaction With Output	Motivation Satisfaction With Output	Implemented A Dp/Cbd Solution That Fits	Approach Intuitive	Model Complexity	Algorithm Complexity	Motivation To Use Algorithmic In Future
DAY 2	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.434	.357	.355	.452*	.454*	.397*	.355	.385*	.361*	.406*
Sig. (2-tailed)	.017	.053	.054	.012	.012	.030	.054	.036	.050	.026
N	30	30	30	30	30	30	30	47	30	30

CBD group:

UTILITY approach is helpful	Usability Easy To Implement	Motivation Satisfaction With Output	Approach Intuitive
DAY 2	DAY 1	DAY 1	DAY 2
Pearson Correlation	.434**	.357*	.438**
Sig. (2-tailed)	.002	.014	.002
N	47	47	47

Dependent variable control (Experience / Gender):

Dependent Variable	Approach / p-value		Approach / F (df)		Design Experience / p		Design Experience / F	
UTILITY / Approach helpful	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Approach / Design Experience		.008		(1,67) 7.591		.238		(4,67) 1.415
	Approach / p-value		Approach / F		Gender / p		Gender / F	
Approach / Gender		.014		(1,73) 6.394		.230		(1,73) 1.462

Comparison of Design Ideation Criteria

Change in the Design Intent

Change in the Design Intent

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
CHANGE IN THE INTENT	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Ability to model original idea	3.00 +- .957	3.20 +- .957	2.80 +- .925	3.30 +- .750	3.15 +- .932	3.53 +- .997	1.274 (101)	1.229 (101)	.284	.297
Change in the design strategy due to programming difficulties	2.96 +- .841	2.67 +- .689	2.93 +- .828	2.70 +- .750	3.19 +- .876	2.68 +- .810	1.201 (125)	.012 (125)	.304	.988
Change in the design strategy because participants found some interesting solutions, which they decided to use	3.29 +- .866	3.27 +- .811	3.23 +- 1.040	3.27 +- .868	3.45 +- .996	3.47 +- .747	.553 (125)	.937 (125)	.577	.395
. I was able to accomplish all what I wanted	3.41 +- .888	3.39 +- .837	3.33 +- .802	3.50 +- .630	3.36 +- .870	3.70 +- .976	.077 (125)	1.666 (125)	.926	.193

Design Objectives

Chi-square test Comparison between the all tree groups: NA, DP and CBD

Criteria	No Approach Count / Total (%)		DP Count / Total (%)		CBD Count / Total (%)		X^2		p – value	
DESIGN OBJECTIVES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
To achieve the form I originally Sketched	**40%** (10/25)	**48%** (12/25)	**56.7%** (17/30)	**60%** (18/30)	**51%** (24/47)	**80.8%** (38/47)	1.555	8.775	**.460**	**.012**
To explore/learn algorithmic form-making process	**24%** (6/25)	**28%** (9/25)	**63.3%** (19/30)	**40%** (12/30)	**46.8%** (22/47)	**23.4%** (11/47)	8.510	2.672	**.014**	**.263**
To experiment with parameters / iterations / variables	**8%** (2/25)	**12%** (3/25)	**20%** (6/30)	**46.7%** (14/30)	**19.1%** (9/47)	**8.5%** (4/47)	1.801	17.800	**.406**	**.000**
To understand / apply the logics and components that I have learned (test my skills)	**28%** (7/25)	**20%** (5/25)	**26.7%** (8/30)	**30%** (9/30)	**23.4%** (11/47)	**21.2%** (7/47)	.212	1.004	**.899**	**.605**
to combine / explore a few Design Patterns / DRR or other definitions to create a complex form	**4%** (1/25)	**8%** (2/25)	**6.7%** (2/30)	**13.3%** (4/30)	**2.1%** (1/47)	**6.4%** (3/47)	1.002	1.127	**.606**	**.569**

To achieve the form I originally sketched

Day 2: NA 12/25 (48%), DP 18/30 (60%), CBD 38/47 (80.8%), X2 = 8.775, p = .012, the count of responses, the percentage, the Chi-Square – value (X2) and the p-value are used.

To explore/learn algorithmic form-making process

Day 1: NA 6/25 (24%), DP 19/30 (63.3%), CBD 22/47 (46.8%), X2 = 8.510, p = .014,

To experiment with parameters / iterations / variables:

Day 2: NA 3/25 (12%), DP 14/30 (46.7%), CBD 4/47 (8.5%), X2 = 8.510, p = .014,

Chi-Square Comparison between the DP and CBD groups

Criteria	DP (yes/30)		DP (%)		CBD (yes/47)		CBD (%)		X^2		p - value	
DESIGN OBJECTIVES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
To achieve the form I originally Sketched	17	18	56.7	60	24	38	51	80.8	.231	4.014	.631	.045
To explore/learn algorithmic form-making process	19	12	63.3	40	22	11	46.8	23.4	2.009	2.408	.156	.121
To experiment with parameters / iterations / variables	6	14	20	46.7	9	4	19.1	8.5	.008	14.884	.927	.000
To understand / apply the logics and components that I have learned (test my skills)	8	9	26.7	30	11	10	23.4	21.2	.105	.750	.746	.387
to combine / explore a few Design Patterns / DRR or other definitions to create a complex form	2	4	6.7	13.3	1	3	2.1	6.4	1.008	1.070	.315	.301

Chi-Square Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach (yes/ 25)		No Approach (%)		DP (%)		CBD (%)		p – value between N/A and DP group		p – value between N/A and CBD group	
DESIGN OBJECTIVES	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
To achieve the form I originally Sketched	10	12	40	48	56.7	60	51	80.8	.169	.268	.259	.005
To explore/learn algorithmic form-making process	6	7	24	28	63.3	40	46.8	23.4	.004	.260	.049	.438
To experiment with parameters / iterations / variables	2	3	8	12	20	46.7	19.1	8.5	.193	.006	.184	.463
To understand / apply the logics and components that I have learned	7	5	28	20	26.7	30	23.4	21.2	.575	.297	.438	.577
to combine / explore a few Design Patterns / DRR or other definitions to create a complex form	1	2	4	8	6.7	13.3	2.1	6.4	.569	.427	.577	.572

Correlation between the ability to realise original idea and the other criteria. All groups:

Ability to realise ORIGINAL IDEA	Design Objectives Accomplish What Wanted	Ability to realise ORIGINAL IDEA	Design Objectives Accomplish What Wanted	Changed Design Idea Because Of Difficulties
DAY 1	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.457**	Pearson Correlation	.519**	-.386**
Sig. (2-tailed)	.000	Sig. (2-tailed)	.000	.000
N	102	N	102	102

No Approach group:

Ability to realise ORIGINAL IDEA	Design Objectives Accomplish What Wanted	Changed Design Idea Because Of Difficulties	Learning Curve Implemented New	Motivation Satisfaction With Output	Variety	Ability to realise ORIGINAL IDEA	Design Objectives Accomplish What Wanted	Changed Design Idea Because Of Difficulties	Model Complexity	Programming Difficulties
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 1	DAY 2
Pearson Correlation	.734**	-.418*	.400*	.533**	.480*	Pearson Correlation	.667**	-.634**	-.386	-.426**
Sig. (2-tailed)	.000	.038	.047	.006	.015	Sig. (2-tailed)	.000	.001	.057	.034
N	25	25	25	25	25	N	25	25	25	25

DP group:

Ability to realise ORIGINAL IDEA	Changed Design Idea Because Of Programming Difficulties	Design Objectives Accomplish What Wanted	Novelty	Ability to realise ORIGINAL IDEA	Design Objectives Accomplish What Wanted
DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	-.378*	.418*	.379*	Pearson Correlation	.402*
Sig. (2-tailed)	.039	.021	.039	Sig. (2-tailed)	.028
N	30	30	30	N	30

CBD group:

Ability to realise ORIGINAL IDEA	Design Objectives Accomplish What Wanted	Programming Difficulties	Ability to realise ORIGINAL IDEA	Design Objectives Accomplish What Wanted	Design Objectives Accomplish What Wanted	Change In Design Intent Programming Difficulties
DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2
Pearson Correlation	.361*	-.503**	Pearson Correlation	.350*	.479**	-.512**
Sig. (2-tailed)	.013	.000	Sig. (2-tailed)	.016	.001	.000
N	47	47	N	47	47	47

Correlation between change in design intent due to programming difficulties and other criteria.

All groups:

CHANGE IN DESIGN INTENT programming difficulties	Programming Difficulties	CHANGE IN DESIGN INTENT programming difficulties	Ability to realise ORIGINAL IDEA
DAY 1	DAY 1	DAY 2	DAY 2
Pearson Correlation	.371**	Pearson Correlation	-.386**
Sig. (2-tailed)	.000	Sig. (2-tailed)	.000
N	126	N	102

No Approach group:

CHANGE IN DESIGN INTENT programming difficulties	Ability To Realise Original Idea	Programming Difficulties	CHANGE IN DESIGN INTENT programming difficulties	Ability To Realise Original Idea	Programming Difficulties
DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	-.418*	.371**	Pearson Correlation	-.634**	.456*
Sig. (2-tailed)	.038	.009	Sig. (2-tailed)	.001	.001
N	25	49	N	25	49

DP group:

CHANGE IN DESIGN INTENT programming difficulties	Ability To Accomplish What Was Wanted	Ability To Realise Original Idea	Programming Difficulties	Motivation Satisfaction With Output	Implemented A Dp/Cbd Solution That Fits	Approach Easy To Implement	Change In Design Intent Programming Difficulties	Programming Difficulties	CHANGE IN DESIGN INTENT programming difficulties	Change In Design Intent Programming Difficulties	Programming Difficulties	Re-Used Algorithms: How Often	Programming Difficulties
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 1	DAY 1	DAY 1	DAY 2
Pearson Correlation	-.433*	-.378*	.420*	-.439*	-.403*	-.480**	.356	.435*	Pearson Correlation	.356	.365*	-.382*	.446*
Sig. (2-tailed)	.017	.039	.021	.015	.027	.007	.054	.016	Sig. (2-tailed)	.054	.048	.037	.014
N	30	30	30	30	30	30	30	30	N	30	30	30	30

CBD group:

CHANGE IN DESIGN INTENT programming difficulties	Change In Design Intent Interesting Solution	Programming Difficulties	CHANGE IN DESIGN INTENT programming difficulties	Ability To Accomplish What Was Wanted	Implemented A Dp/Cbd Solution That Fits	Ability To Realise Original Idea
DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 1	DAY 2
Pearson Correlation	.423**	.353*	Pearson Correlation	-.357*	-.387**	-.512**
Sig. (2-tailed)	.003	.015	Sig. (2-tailed)	.014	.007	.000
N	47	47	N	47	47	47

Correlation between change in design intent, because 'discovered solutions' and the other criteria.

No Approach group:

CHANGE IN DESIGN INTENT interesting solution	Ability to accomplish what was wanted	CHANGE IN DESIGN INTENT interesting solution
DAY 1	DAY 1	DAY 2
Pearson Correlation	.414**	Pearson Correlation
Sig. (2-tailed)	.003	Sig. (2-tailed)
N	49	N

DP group:

CHANGE IN DESIGN INTENT interesting solution	Approach Easy To Implement	Model Complexity
DAY 2	DAY 1	DAY 2
Pearson Correlation	.350	-.371*
Sig. (2-tailed)	.058	.044
N	30	30

CBD group:

CHANGE IN DESIGN INTENT interesting solution	Change In Design Intent Programming Difficulties	Approach Easy To Implement	Novelty	CHANGE IN DESIGN INTENT interesting solution	Ability To Accomplish What Was Wanted	Motivation To Use Algorithmic In Future
DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.423**	.397**	-.380**	Pearson Correlation	.374**	.386**
Sig. (2-tailed)	.003	.006	.009	Sig. (2-tailed)	.010	.007
N	47	47	47	N	47	47

Correlation between ability to accomplish what was wanted and the other criteria. All groups:

Ability to accomplish what wanted	Ability To Realise Original Idea	Motivation Satisfaction With Output	Ability to accomplish what wanted	Motivation Satisfaction With Output	Ability To Realise Original Idea
DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.457**	.578**	Pearson Correlation	.628**	.519**
Sig. (2-tailed)	.000	.000	Sig. (2-tailed)	.000	.000
N	102	126	N	126	30

No Approach group:

Ability to accomplish what wanted	Ability To Realise Original Idea	Change In Design Intent Found New Solutions	Motivation Satisfaction With Output	Motivation To Use Gh In Future	Ability to accomplish what wanted	Ability To Realise Original Idea	Programming Difficulties	Motivation Satisfaction With Output
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Correlation	.734**	.414**	.565**	.455**	Pearson Correlation	.667**	-.408**	.663**
Sig. (2-tailed)	.000	.003	.000	.001	Sig. (2-tailed)	.000	.004	.000
N	25	49	49	49	N	25	49	49

DP group:

Ability to accomplish what wanted	Usability Easy To Implement	Ability To Realise Original Idea	Change In Design Intent Programming Difficulties	Motivation Satisfaction With Output	Motivation To Use Gh In Future	Flexibility Found Solution Which Fits	Ability to accomplish what wanted	Ability To Realise Original Idea
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2
Pearson Correlation	.486	.418*	-.433	.591	.352	.432	Pearson Correlation	.402
Sig. (2-tailed)	.006	.021	.017	.001	.056	.017	Sig. (2-tailed)	.028
N	30	30	30	30	30	30	N	30

CBD group:

Ability to accomplish what wanted	Ability To Realise Original Idea	Model Complexity	Ability To Realise Original Idea	Change In Design Intent Programming Difficulties	Algorithm Complexity	Programming Difficulties	Usability Easy To Implement	Motivation Satisfaction With Output	Ability to accomplish what wanted	Usability Easy To Implement	Ability To Realise Original Idea	Change In Design Intent Found New Solutions	Motivation Satisfaction With Output	Flexibility Found Solution Which Fits	Usability Easy To Implement	Algorithm Complexity
DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2	DAY 2
Pearson Corr-n	.361	-.359	.350	-.357	-.362	-.350	.627	.626	Pearson Corr-n	.408	.479	.374	.702	.604	.358	-.378
Sig.	.013	.013	.016	.014	.013	.016	.000	.000	Sig.	.004	.001	.010	.000	.000	.013	.009
N	47	47	47	47	47	47	47	47	N	47	47	47	47	47	47	47

Comparison of Motivation Criteria

Satisfaction with Outcome / Motivation

ANOVA Comparison between No Approach group and the DP/ CBD groups

Criteria	No Approach Group (Mean)		DP (Mean)		CBD (Mean)		F (df)		p – value	
MOTIVATION	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
I am satisfied with what I was able to accomplish	3.47 +- .981	3.69 +- .895	3.72 +- .826	3.80 +- .664	3.70 +- .805	3.98 +- .847	1.359 (125)	1.440 (125)	.261	.241
In the near future, I plan to use Grasshopper for Rhino very often		3.96 +- .815		3.80 +- .610		4.04 +- .908		.825 (125)		.441

Satisfaction with output

Day 1: F (125) = 1.359, p = 0.261; F ratio (F), the degrees of freedom (df) and p-value are used.

Day 2: F (125) = 1.440, p = 0.241;

Motivation to use algorithmic design tools in future

Day 2: F (125) = 0.825, p = 0.441;

Correlation between satisfaction with output model and the other criteria. All groups:

MOTIVATION satisfaction with output	Ability to accomplish what wanted	MOTIVATION satisfaction with output	Ability to accomplish what wanted
DAY 1	DAY 1	DAY 2	DAY 2
Pearson Correlation	.578**	Pearson Correlation	.628**
Sig. (2-tailed)	.001	Sig. (2-tailed)	.000
N	126	N	126

Correlation between satisfaction with output model and the other criteria.

No Approach group:

MOTIVATION satisfaction with output	Ability To Accomplish What Wanted	Ability To Model Original Idea	Algorithm Complexity	Variety	MOTIVATION satisfaction with output	ability to accomplish what wanted	programming difficulties
DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.565**	.533**	.363*	.406**	Pearson Correlation	.663**	.400*
Sig. (2-tailed)	.000	.006	.010	.004	Sig. (2-tailed)	.000	.004
N	49	25	49	49	N	49	49

DP group:

MOTIVATION satisfaction with output	Usability Easy To Implement	Design Objectives Accomplish What Wanted	Change In Design Intent Programming Difficulties	Motivation Satisfaction With Output	Flexibility Found Solution Which Fits	Flexibility Found Solution Which Fits	Utility Approach Is Helpful	Model Complexity	Variety
DAY 1	DAY 1	DAY 1	DAY 1	DAY 2	DAY 1	DAY 2	DAY 2	DAY 1	DAY 1
Pearson Correlation	.577**	.591**	-.439*	.439*	.382*	.485**	.452*	.463*	.350
Sig. (2-tailed)	.001	.001	.015	.015	.037	.007	.012	.010	.058
N	30	30	30	30	30	30	30	30	30

MOTIVATION satisfaction with output	Usability Easy To Implement	Motivation Satisfaction With Output	Re-Used Algorithms	Flexibility Found Solution Which Fits	Usability Easy To Implement	Utility Approach Is Helpful	Model Complexity
DAY 2	DAY 1	DAY 1	DAY 2	DAY 2	DAY 2	DAY 2	DAY 1
Pearson Correlation	.434*	.439*	-.380	.600**	.462*	.454*	.441*
Sig. (2-tailed)	.017	.015	.038	.000	.010	.012	.015
N	30	30	30	30	30	30	30

Correlation between satisfaction with output model and the other criteria. CBD group:

MOTIVATION satisfaction with output	Usability Easy To Implement	Design Objectives Accomplish What Wanted	Programming Difficulties	Flexibility Found Solution Which Fits	Approach Helpful	MOTIVATION satisfaction with output	Usability Easy To Implement	Design Objectives Accomplish What Wanted	Flexibility Found Solution Which Fits	Usability Easy To Implement
DAY 1	DAY 1	DAY 1	DAY 1	DAY 1	DAY2	DAY 2	DAY 1	DAY 2	DAY 2	DAY 2
Pearson Correlation	.495**	.626**	-.357*	.495**	.357*	Pearson Correlation	.409**	.702**	.372*	.415**
Sig. (2-tailed)	.000	.000	.014	.000	.014	Sig. (2-tailed)	.004	.000	.010	.004
N	47	47	47	47	47	N	47	47	47	47

Correlation between motivation to use Grasshopper in future and the other criteria. No Approach group:

MOTIVATION to use algorithmic in future	Design Objectives Accomplish What Wanted
DAY 2	DAY 1
Pearson Correlation	.455**
Sig. (2-tailed)	.001
N	49

Correlation between motivation to use Grasshopper in future and the other criteria. DP group:

MOTIVATION to use algorithmic in future	Design Objectives Accomplish What Wanted	Programming Difficulties	Utility Approach Is Helpful
DAY 2	DAY 1	DAY 1	DAY 2
Pearson Correlation	.352	-.490**	.406*
Sig. (2-tailed)	.056	.006	.026
N	30	30	30

Correlation between motivation to use Grasshopper in future and the other criteria. CBD group:

MOTIVATION to use algorithmic in future	Changed Design Idea : Found New Solutions	Learning Curve: Implemented New
DAY 2	DAY 2	DAY 2
Pearson Correlation	.386**	.457**
Sig. (2-tailed)	.007	.001
N	47	47

Gender as Influence Factor

**Only the cases when the p-value is below .05 are shown*

Comparison between All Male and Female Participants (All Groups) T-Test / Chi-Square:

Criteria	MALE (Mean / %)		FEMALE (Mean/ %)		t		df		p - value	
	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
I was able to model the original design idea.	3.20 +- .923	3.48 +- .894	2.83 +- .926	3.24 +- .923	2.013	1.346	100	100	.047	.181
How often have you re-used the algorithms from any external sources	2.14 +- .921	2.17 +- .701	2.27 +- 1.036	2.61 +- .985	-.716	-2.792	124	96.071	.475	.006
DESIGN OBJECTIVES: To combine a few Design Patterns / Case-Base solutions	3.5 %	3.5 %	4.3%	15.2%					.614	.043
Algorithm Novelty	31.40 +- 17.088	50.71 +- 22.734	24.11 +- 16.483	46.39 +- 27.755	2.418	.968	124	124	.017	.335

Design Patterns Approach. Comparison between Male And Female Participants:

Criteria	MALE (Mean)		FEMALE (Mean)		t		df		p - value	
	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Algorithm Novelty	36.60 +- 22.443	55.40 +- 19.813	22.00 +- 12.048	51.193 +- 22.413	2.220	.449	28	28	.037	.657
Algorithm Variety	16.87 +- 4.984	16.87 +- 5.680	13.00 +- 4.984	18.33 +- 4.608	2.132	-.777	28	28	.042	.444
Algorithm Complexity	64.40 +- 40.57	52.94 +- 29.85	36.80 +- 14.87	60.20 +- 27.04	2.240	-0.699	28	28	.020	.490
How often have you re-used the algorithms from any external sources	2.20 +- .775	2.13 +- .352	2.53 +- 1.060	2.87 +- .743	-.983	-3.454	28	28	.334	.003

Case-Based Design Approach. Comparison between Male And Female Participants:

Criteria	MALE (Mean)		FEMALE (Mean)		t		df		p - value	
	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
Algorithm Novelty	31.17 +- 15.676	44.77 +- 16.768	20.82 +- 16.452	41.47 +- 19.900	2.135	.605	45	45	.038	.548
Algorithm Variety	13.57 +- 3.510	15.90 +- 2.551	11.35 +- 3.390	15.53 +- 4.230	2.103	.376	45	45	.041	.709
Algorithm Complexity	53.06 +- 32.67	54.00 +- 29.27	45.70 +- 25.20	52.88 +- 24.87	0.802	.132	45	45	.427	.895
How often have you re-used the algorithms from any external sources	2.37 +- .928	2.20 +- .761	2.24 +- 1.091	2.59 +- 1.004	.438	- 1.495	45	45	.664	.142

Male Participants. Comparison between The Approaches:

Criteria	DP(Mean)		CBD (Mean)		t		df		p - value	
	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
How often you have come across programming difficulties	2.40 +- .632	2.07 +- .458	2.87 +- 1.008	2.63 +- .850	- 1.634	- 2.904	43	42.668	.110	.006
Algorithm Variety	16.87 +- 4.984	16.87 +- 5.680	13.57 +- 3.510	15.90 +- 2.551	2.577	.792	43	43	.013	.433
Algorithm Complexity	64.40 +- 40.57	52.93 +- 29.85	53.07 +- 32.67	54.00 +- 29.27	1.011	-.114	43	43	.318	.909
Model Complexity	12.67 +- 1.952	13.93 +- 2.764	12.13 +- 2.270	13.03 +- 2.606	.777	1.071	43	43	.442	.290
It was easy to implement DP/CBD approach in my design.	3.07 +- .704		3.77 +- .626			- 3.393		43		.001
I find DP/CBD approach - intuitive	3.27 +- .704		3.80 +- .847			- 2.100		43		.042
I find DP/CBD approach - helpful.	4.00 +- .655		4.37 +- .556			- 1.965		43		.056

Female Participants. Comparison between the Approaches:

Criteria	DP(Mean)		CBD (Mean)		t		df		p - value	
	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2	DAY 1	DAY 2
How often you have come across programming difficulties	2.33 +- .724	2.13 +- .352	3.00 +- 1.118	2.35 +- .606	-2.024	-1.270	27.673	26.185	.053	.215
Algorithm Variety	13.40 +- 3.851	18.33 +- 4.608	11.35 +- 3.390	15.53 +- 4.230	1.600	1.795	30	30	.120	.083
Algorithm Complexity	36.80 +- 14.87	60.20 +- 27.04	45.70 +- 25.20	52.88 +- 24.87	-1.196	.797	30	30	.241	.432
Model Complexity	11.80 +- 2.111	14.27 +- 2.404	12.18 +- 2.270	12.24 +- 2.437	-.484	2.368	30	30	.632	.025
It was easy to implement DP/CBD approach in my design.		3.00 +- .926		3.76 +- .752		-2.577		30		.015
I find DP/CBD approach - intuitive		3.47 +- .743		3.82 +- .883		-1.228		30		.229
I find DP/CBD approach - helpful.		3.87 +- .640		4.18 +- .393		-1.672		30		.105

Results of the Comparative Study

Criteria	No Approach (Mean / %)		DP Approach (Mean / %)		CBD Approach (Mean / %)		p value	
MODELLING SPEED / MODEL COMPLEXITY	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
Model complexity score	11.73	13.94	12.23	14.10	12.15	12.74	.560	.031
ALGORITHM COMPLEXITY	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
Algorithm complexity score	40.69	54.61	50.60	56.56	50.40	53.59		
EXPLORED SOLUTION SPACE	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
Algorithm Variety score	12.43	16.65	15.13	17.60	12.77	15.77	.008	.268
Algorithm Novelty score	28.16	50.82	29.30	53.67	27.43	43.57	.898	.171
PROGRAMMING DIFFICULTIES	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
How often participants came across programming difficulties	2.88	2.71	2.37	2.10	2.91	2.53	.036	.003
Type of difficulties (5)	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
Problems with particular Components	44.8%	48.9%	33.3%	43.3%	21.3%	23.4%	.049	.029
LEARNING CURVE	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
How often participants have implemented new components	2.16	2.16	2.43	1.87	2.21	2.09	.572	.353
RE-USE OF KNOWLEDGE	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
How often participants have re-used algorithms	1.98	2.31	2.37	2.50	2.32	2.34	.128	.610
USABILITY	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
How easy to learn/ implement each approach			2.90	3.03	3.66	3.77	.000	.000
INTUITIVENESS	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
How intuitive participants find each approach				3.37		3.81		.021
FLEXIBILITY	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
Ability to find and adapt/ implement Design Pattern / CBD Solution which fits			3.37	3.80	3.66	3.64	.196	.356
How often participants have implemented DP / CBD solutions			70%	66.7%	76.6%	87.2%	.350	.031
UTILITY / usefulness	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
How helpful did participants find each approach				3.93		4.30		.007
CHANGE IN THE DESIGN INTENT	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
Ability to model original idea	3.00	3.20	2.87	3.30	3.15	3.51	.284	.297
Change in the design strategy due to programming difficulties	2.96	2.67	2.93	2.70	3.19	2.68	.304	.988
Change in the design strategy because participants found some interesting solutions	3.29	3.27	3.23	3.27	3.45	3.47	.577	.395
Ability to accomplish what was intended / wanted	3.41	3.39	3.33	3.50	3.36	3.70	.926	.193
Design objectives	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
To achieve the form I originally sketched	56%	68%	56.7%	60%	51%	80.8%	.460	.012
To explore/learn algorithmic form-making	24%	28%	63.3%	40%	46.8%	23.4%	.014	.263
To experiment with parameters / iterations	8%	12%	20%	46.7%	19.1%	8.5%	.406	.000
IDEATION / KEY WORDS	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
Geometry/ Shape key words								
Non-standard Geometry			70%		48.9%		.069	
Commands / Programming Components				30%		12.7%		.063
Abstract attributes / Metaphors key words								
Descriptive Attributes			60%		80.0%		.045	
Algorithmic Modelling key words								
Non-Standard Geometry			0%	0%	19.1%	19.1%	.011	0.11
Descriptive Attributes				0%		12.7%		0.42
Commands / Programming Components			96.7%		80.8%		.044	
DEGREE OF SATISFACTION	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2	Day 1	Day 2
Level of satisfaction with the design outcome	3.47	3.69	3.77	3.80	3.70	3.98	.261	.241
Motivation to use algorithmic design in future		3.96		3.80		4.04		.441

Diagrams and Illustrations

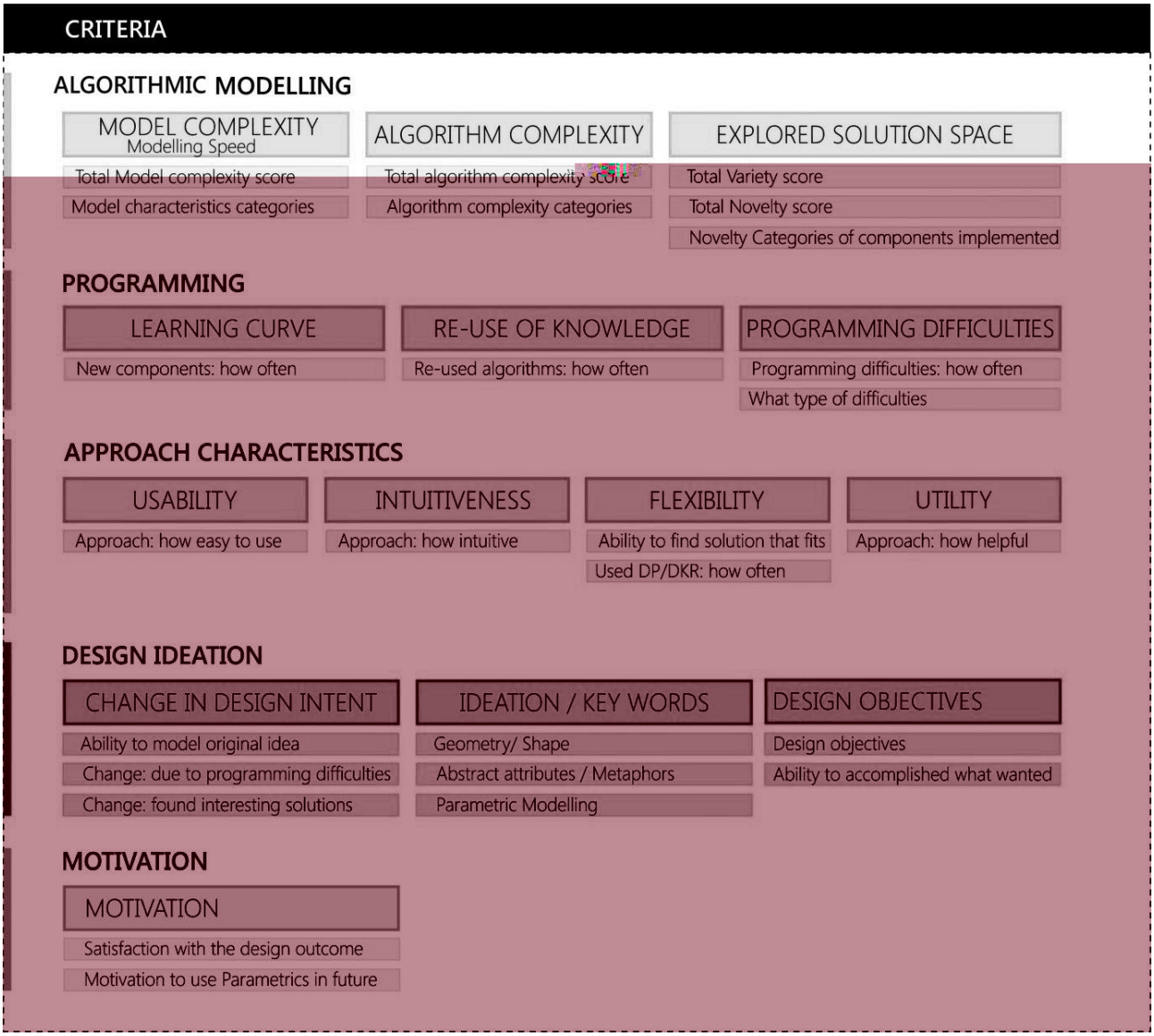


Exhibit B1. Evaluation Criteria Groups.

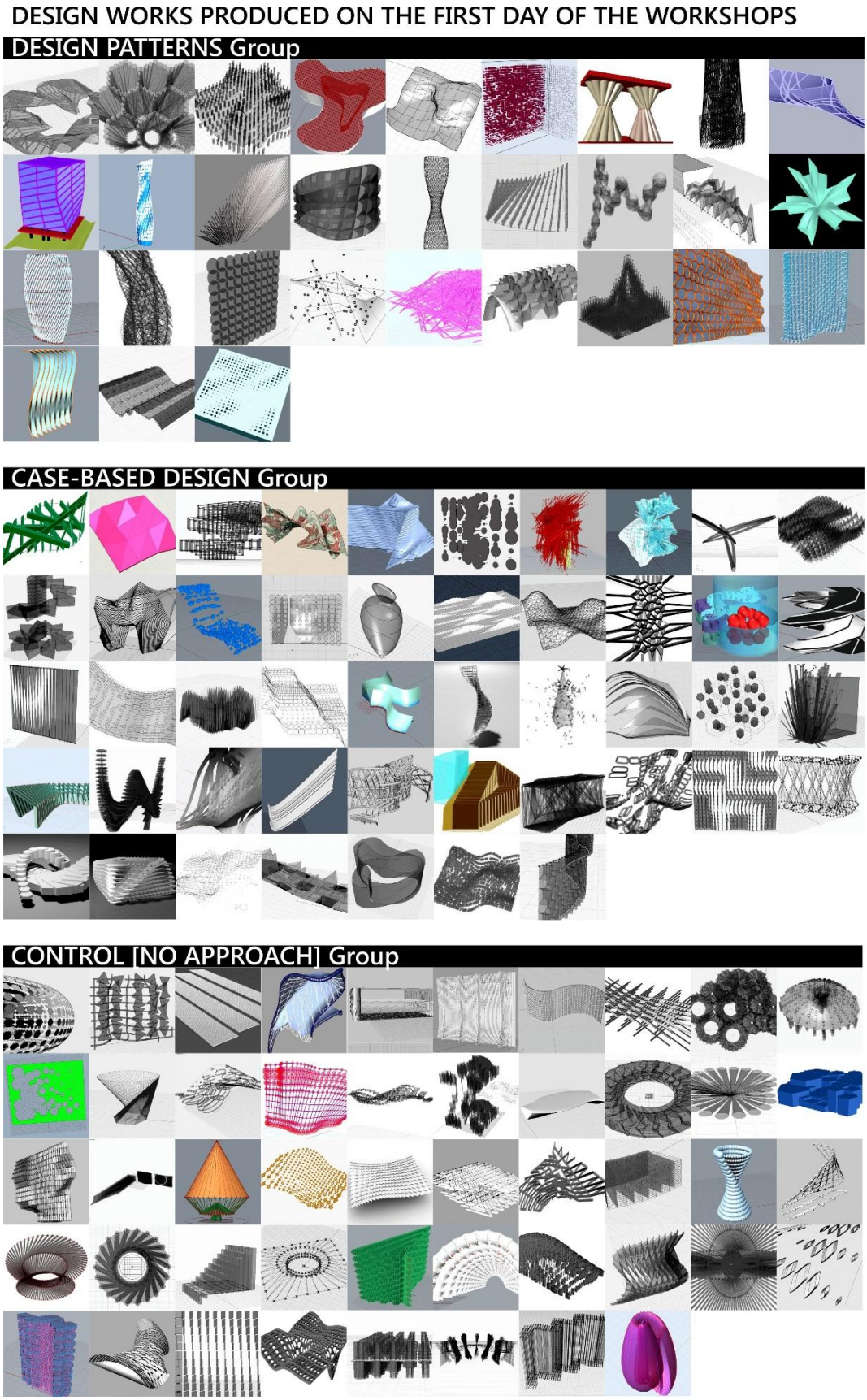


Exhibit B2. Design works produced by the participants of the DP, CBD and NA groups on the first day of the workshops

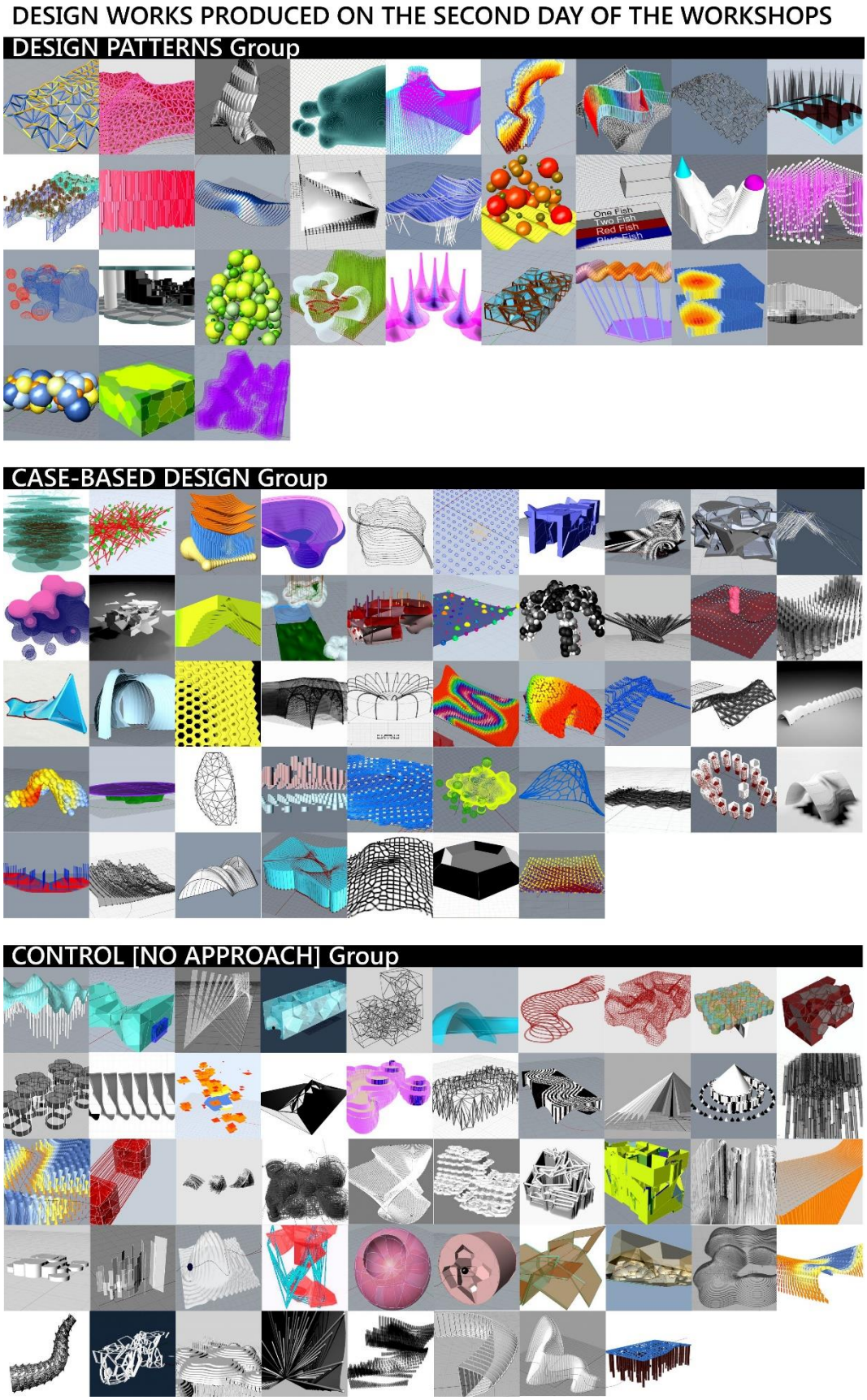


Exhibit B3. Design works produced by the participants of the DP, CBD and NA groups on the second day of the workshops

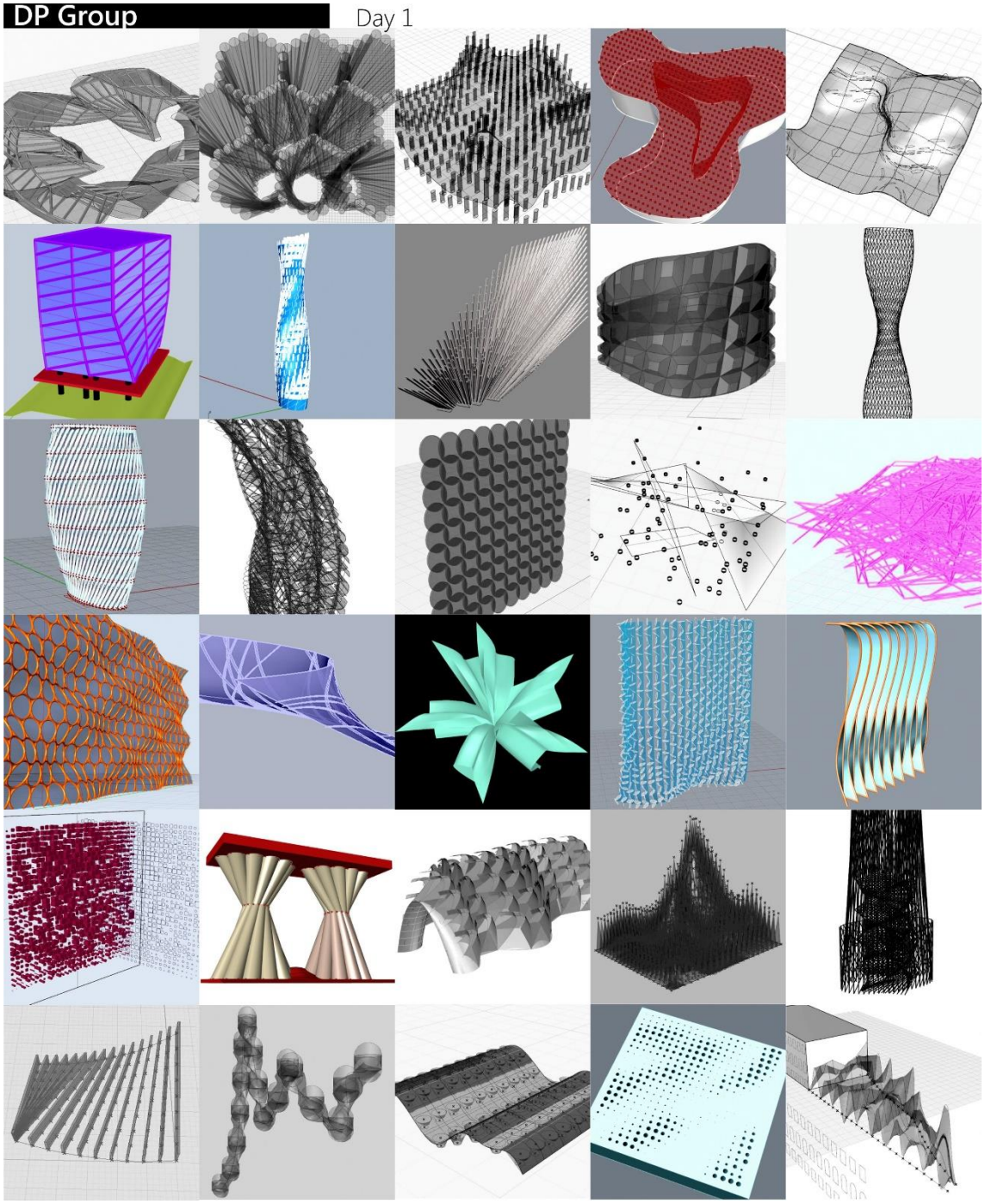


Exhibit B4. Design works produced by the participants of the DP group on the first day of the workshops

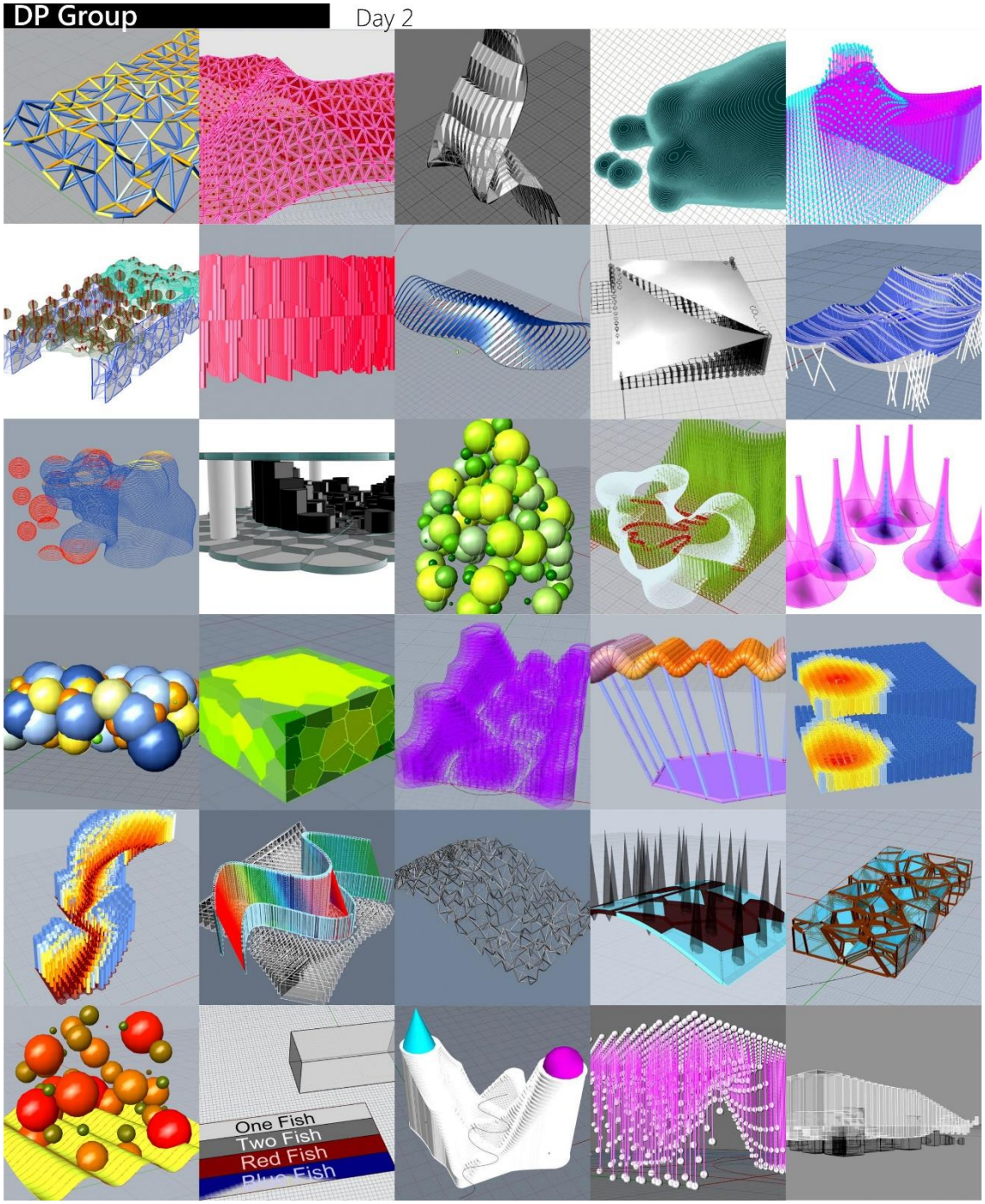


Exhibit B5. Design works produced by the participants of the DP group on the second day of the workshops

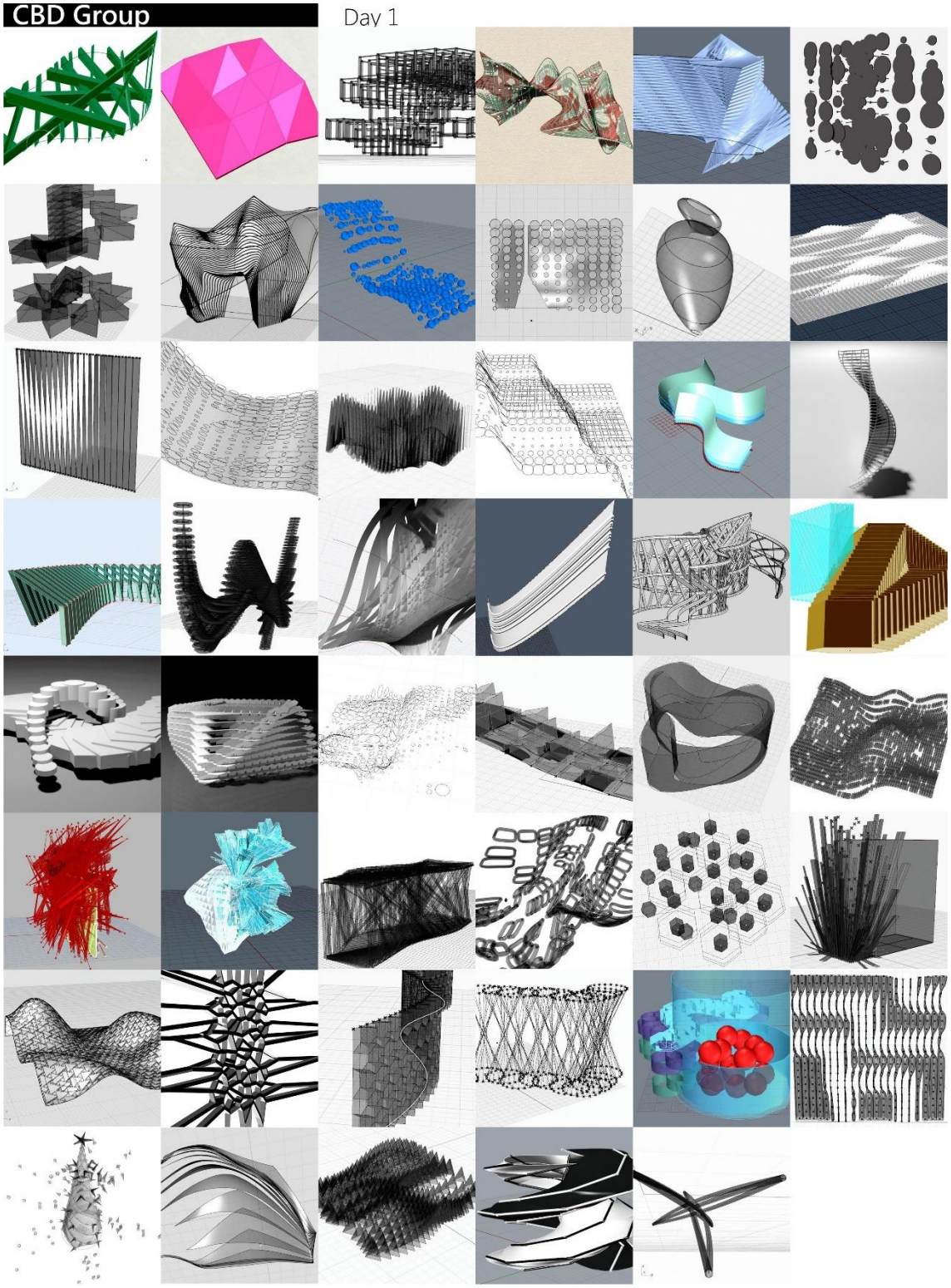


Exhibit B6. Design works produced by the participants of the CBD group on the first day of the workshops

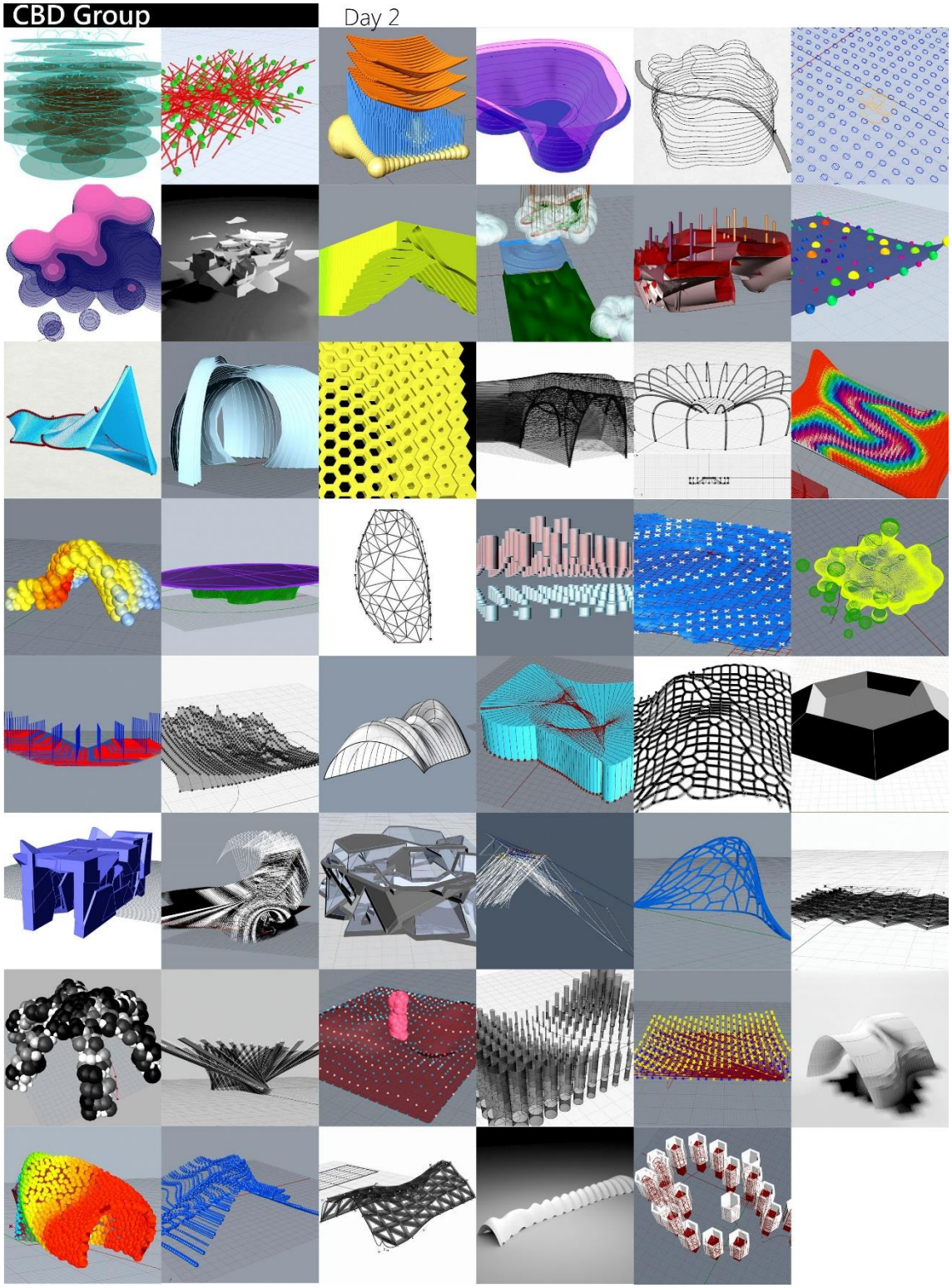


Exhibit B7. Design works produced by the participants of the CBD group on the second day of the workshops

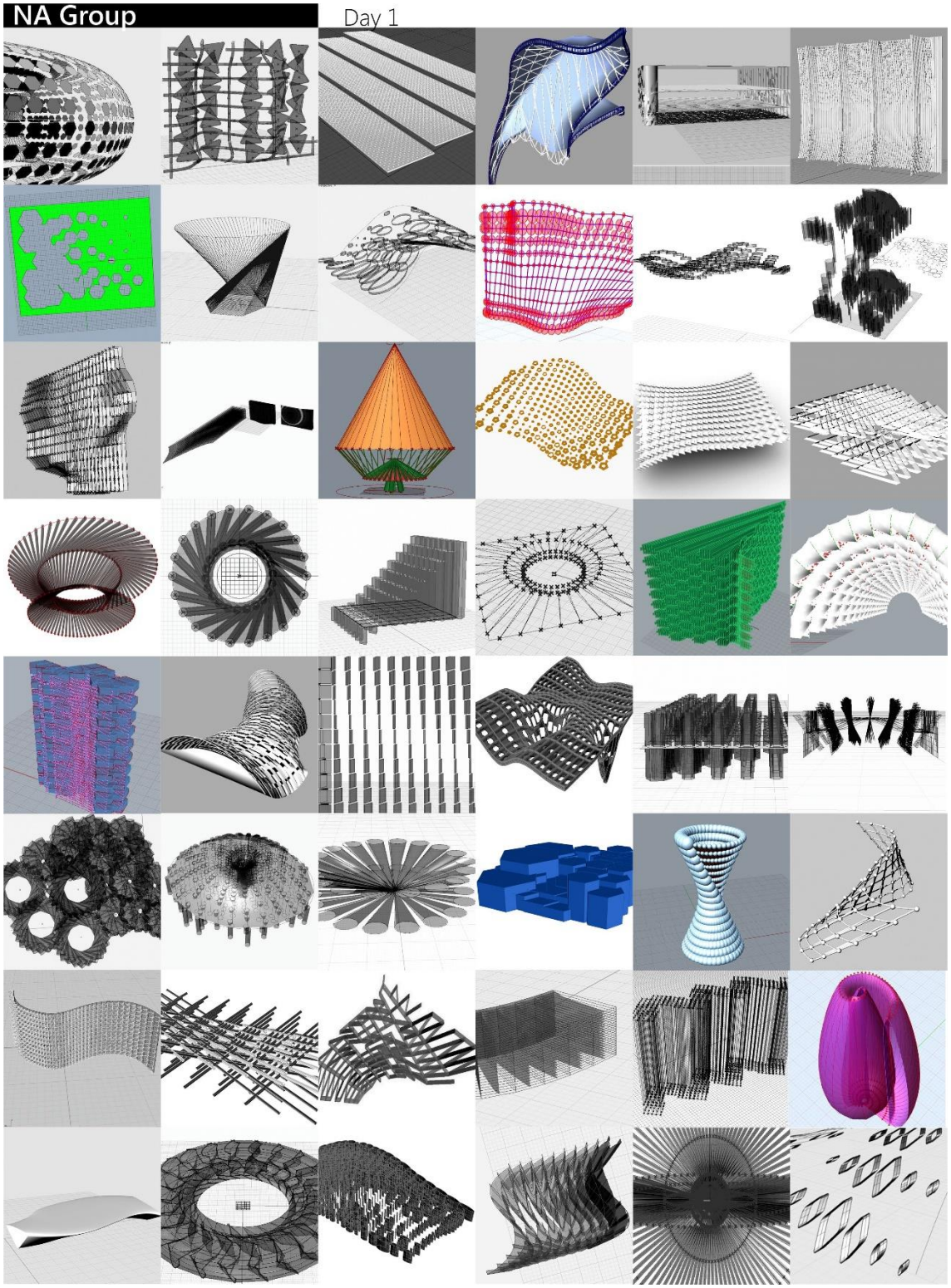


Exhibit B8. Design works produced by the participants of the control group on the first day of the workshops

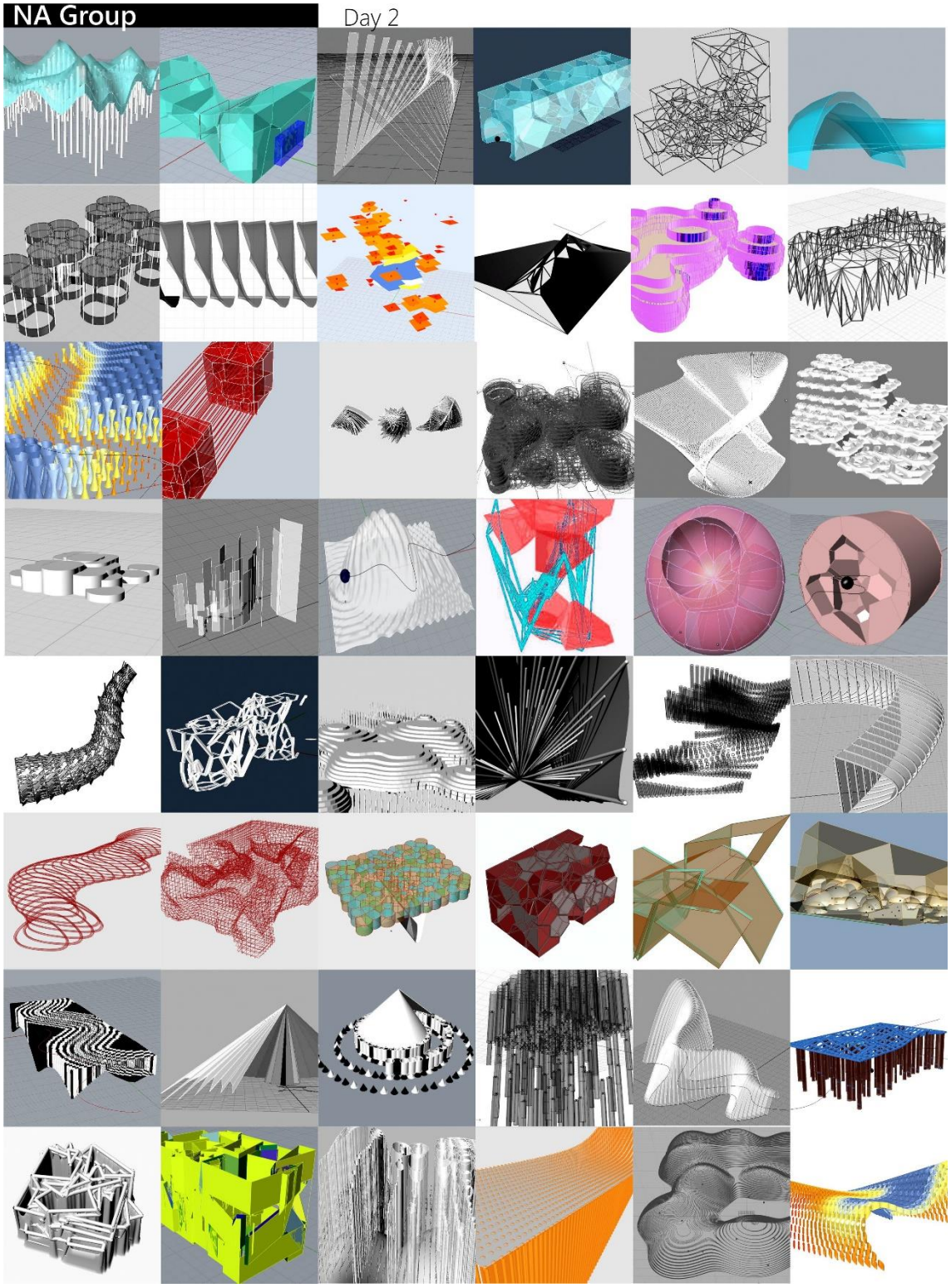


Exhibit B9. Design works produced by the participants of the control group on the second day of the workshops

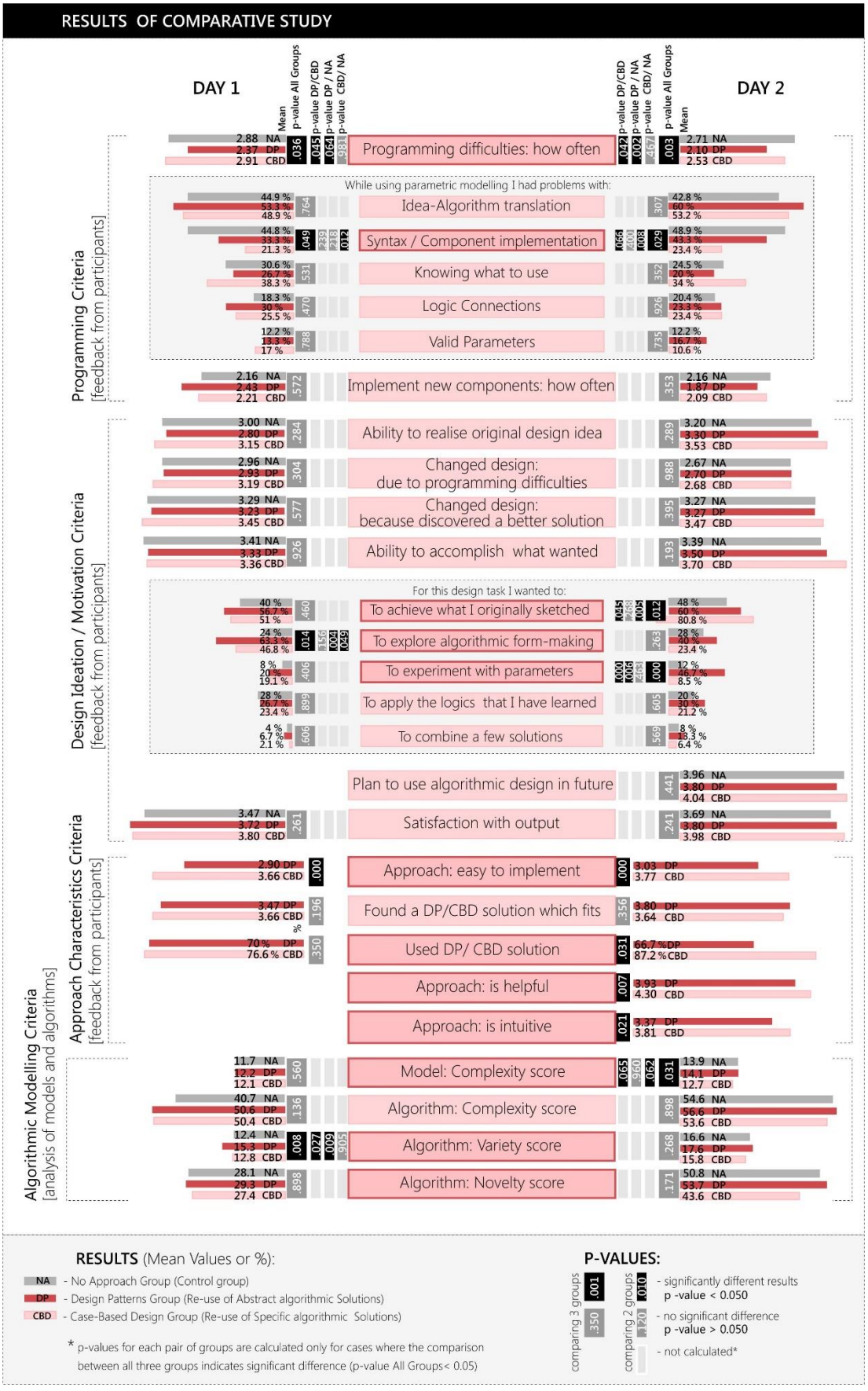


Exhibit B10. All criteria groups: Results of the comparative study

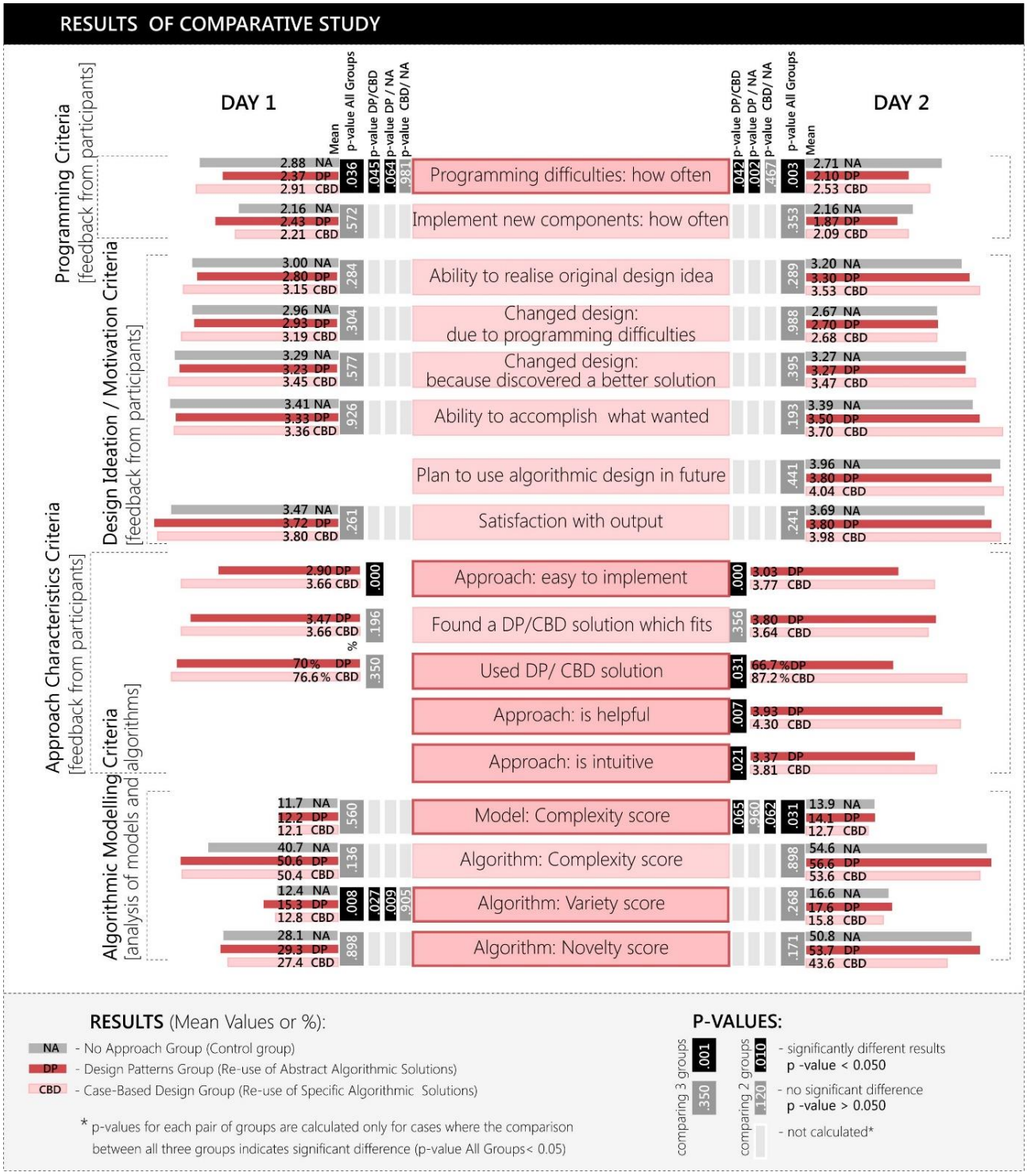


Exhibit B11. Main criteria groups: Results of the comparative study

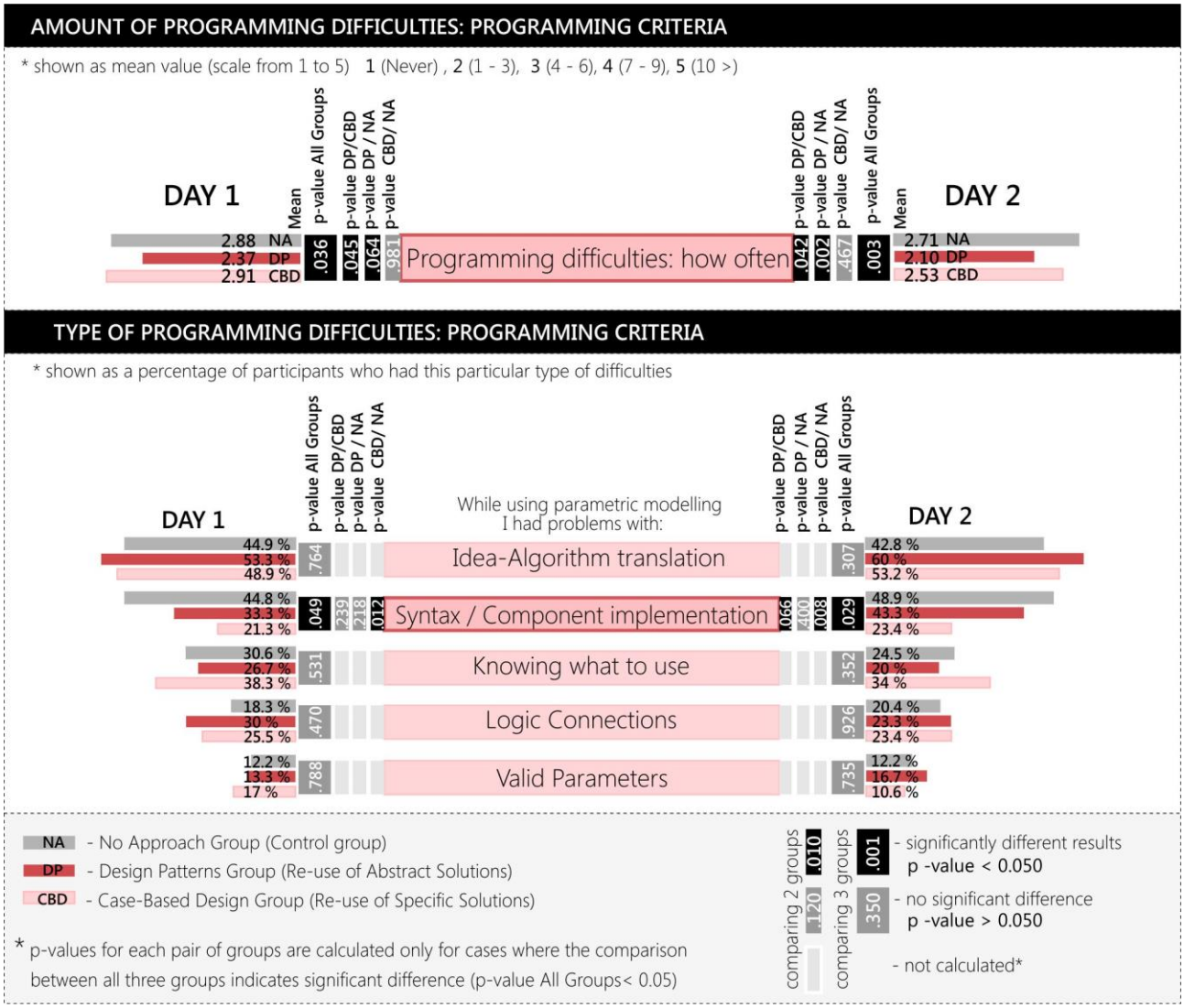


Exhibit B12. Amount of programming barriers chart / Typology of programming barriers comparison

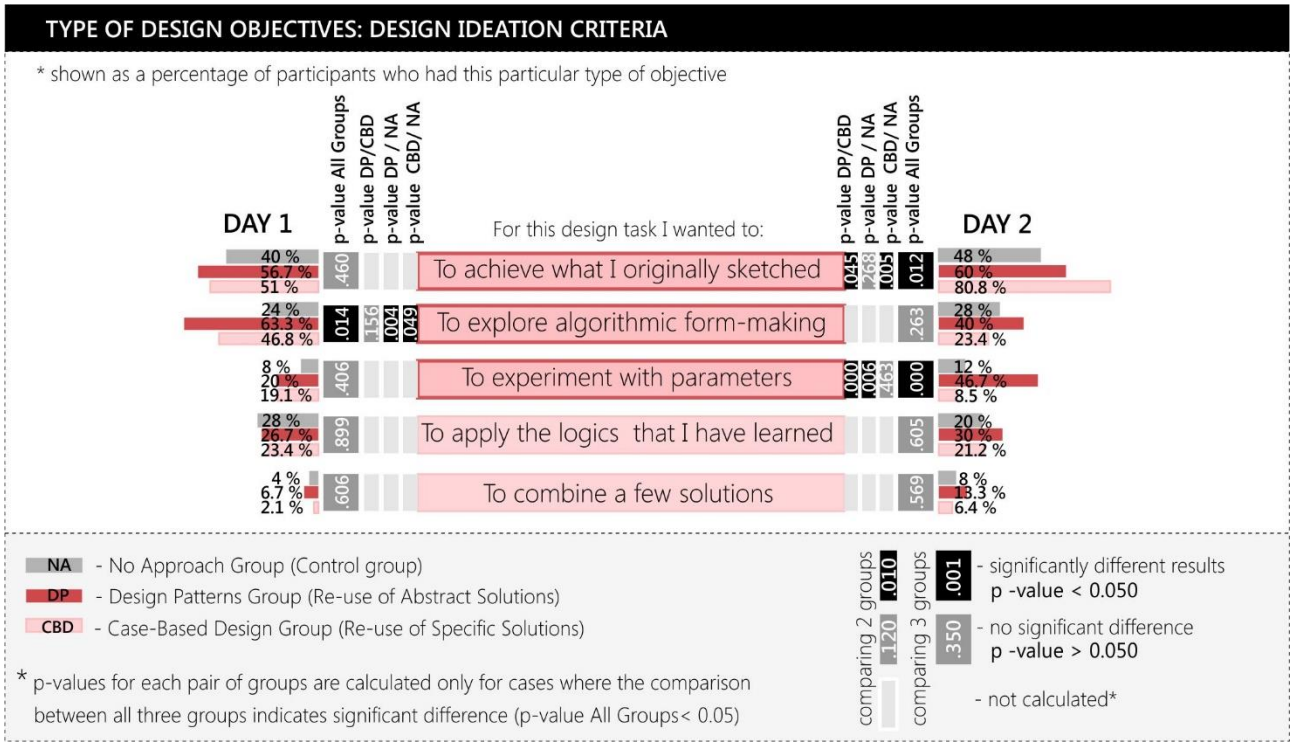


Exhibit B13. Ideation Criteria chart: Types of design objectives

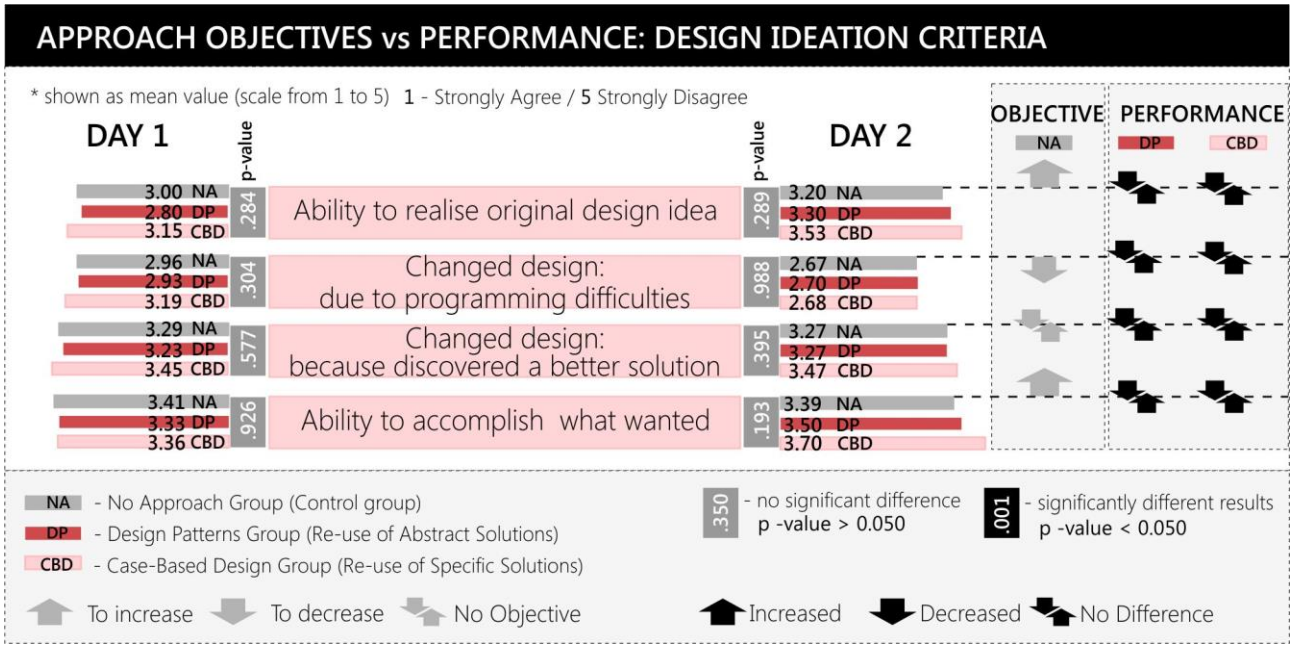


Exhibit B14. Design Ideation. Comparison chart: Approach objectives vs Performance

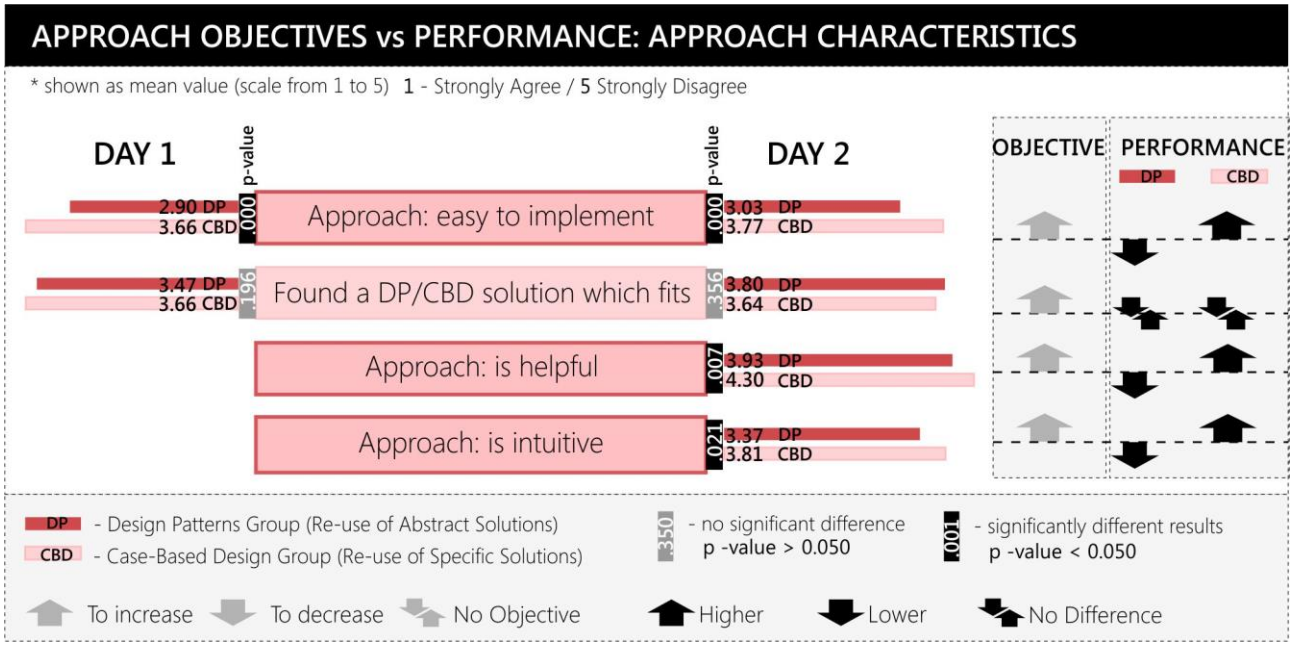


Exhibit B15. Approach Characteristics. Comparison chart: Approach objectives vs Performance

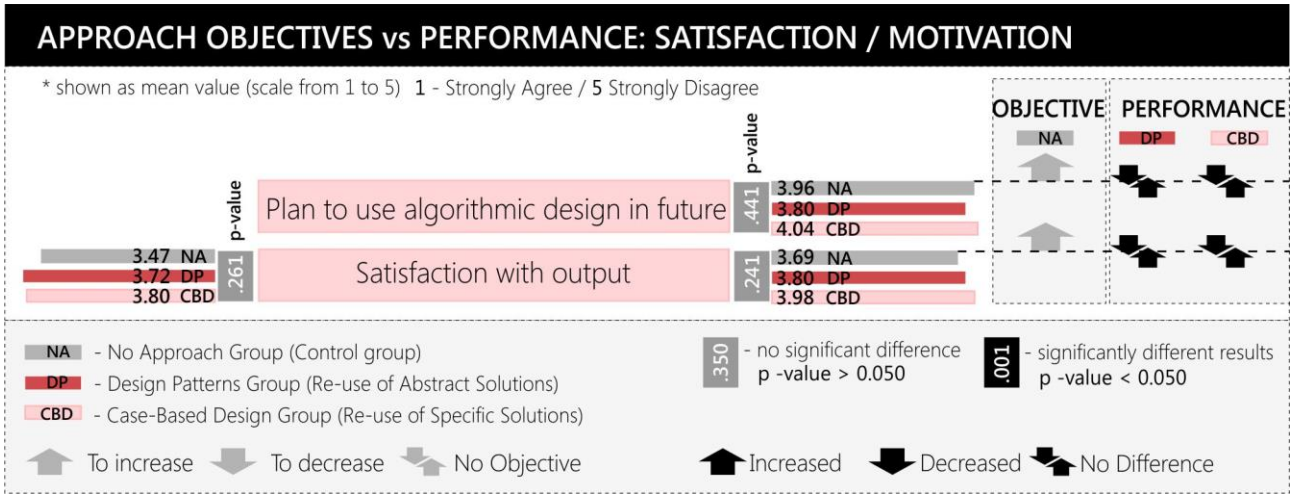


Exhibit B16. Satisfaction / Motivation criteria. Comparison chart: Approach objectives vs Performance

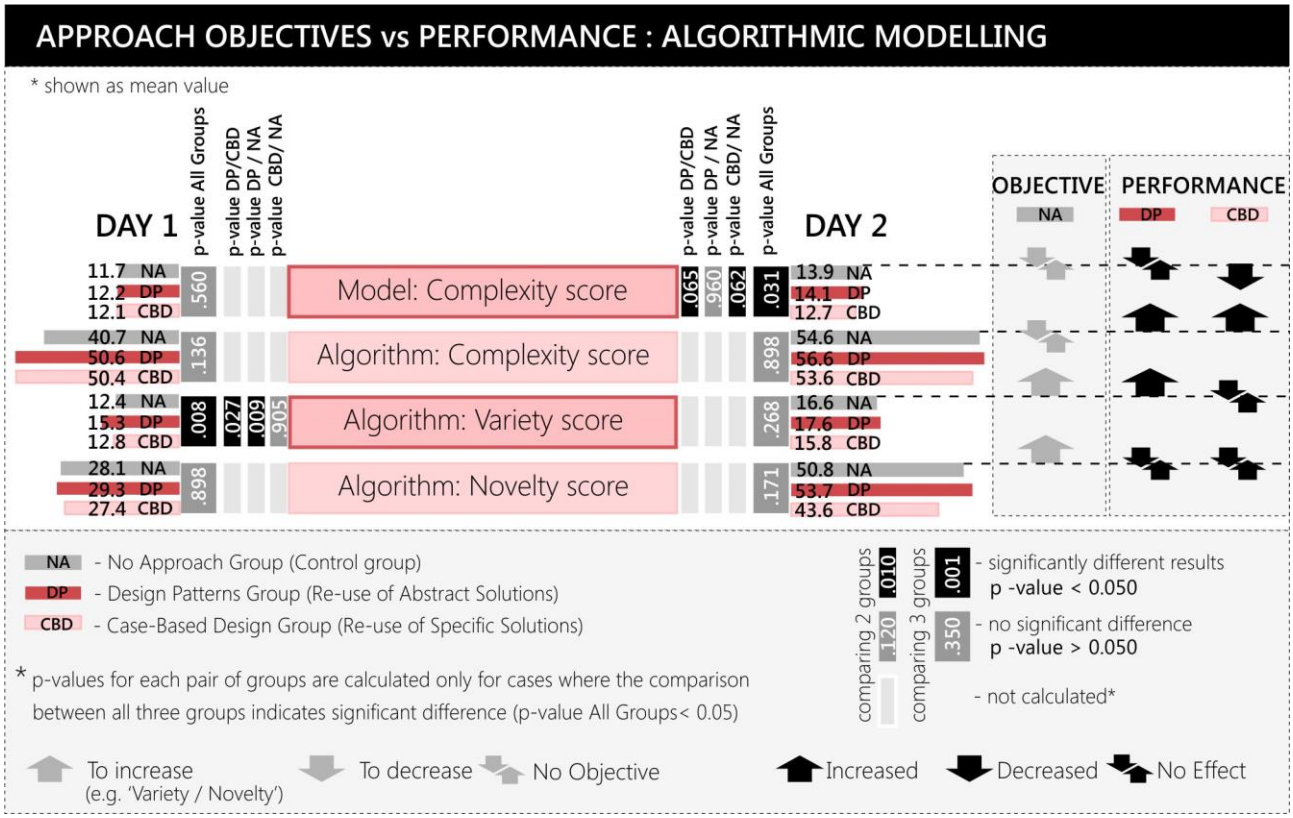


Exhibit B17. Algorithmic modelling criteria. Comparison chart: Approach objectives vs Performance

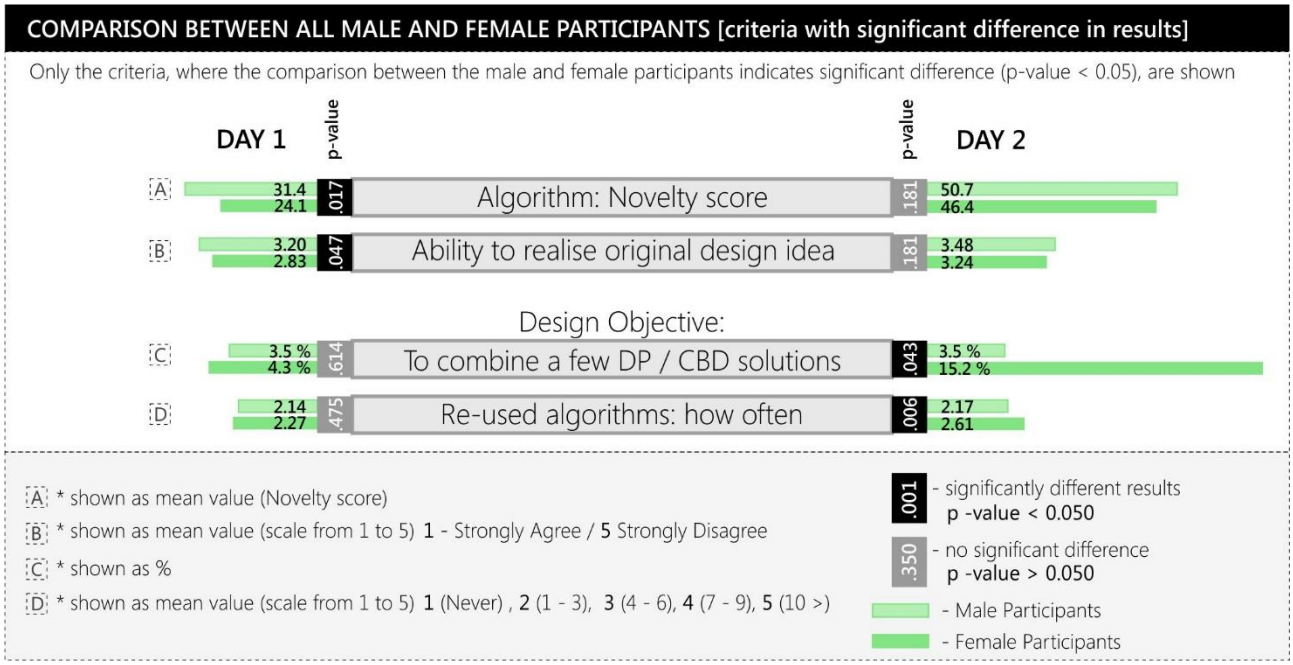


Exhibit B18. Comparison between all male and female participants. Only criteria with the significant difference in results are shown

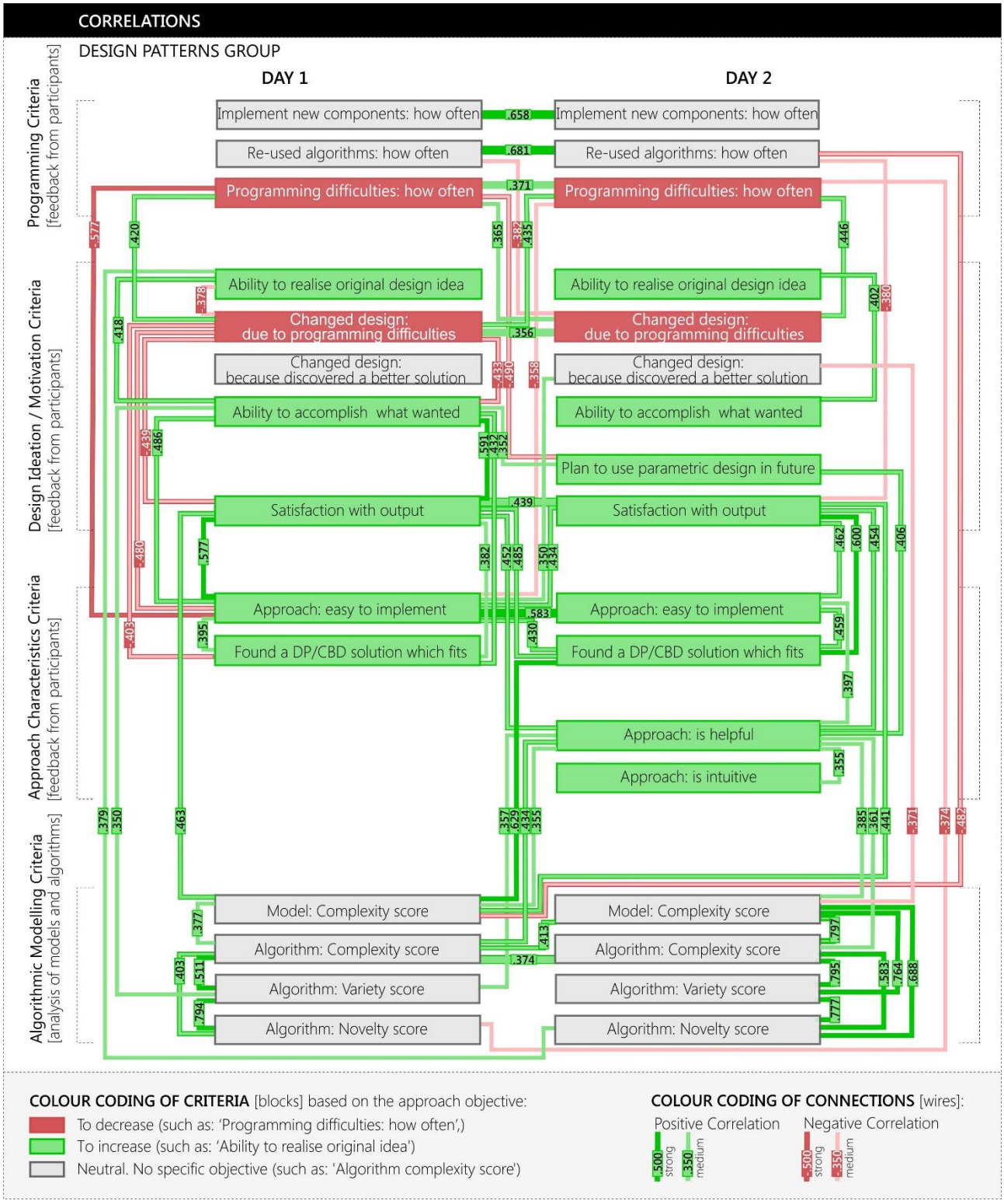


Exhibit B19. DP group. Correlations between all criteria.

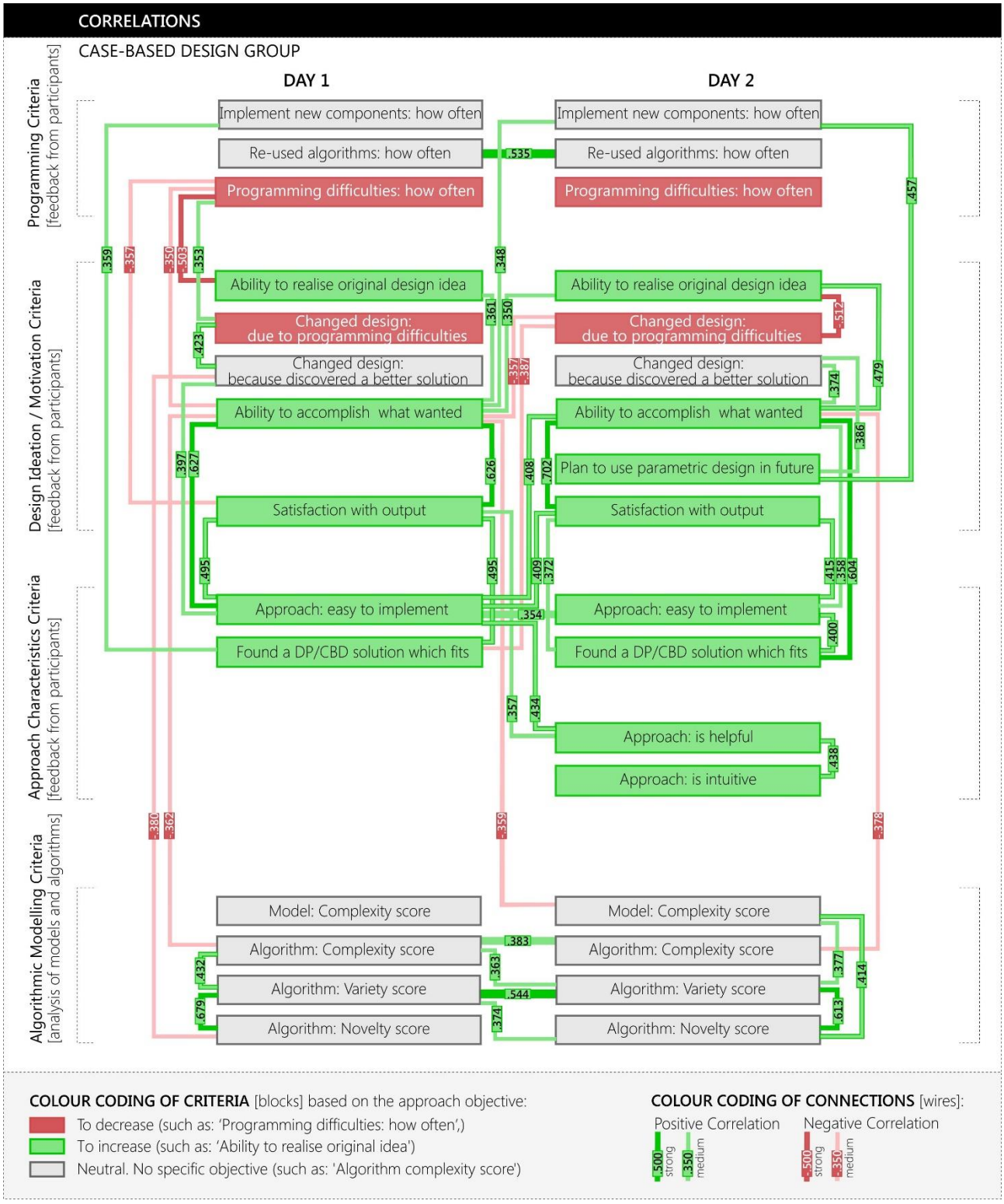


Exhibit B20. CBD group. Correlations between all criteria.

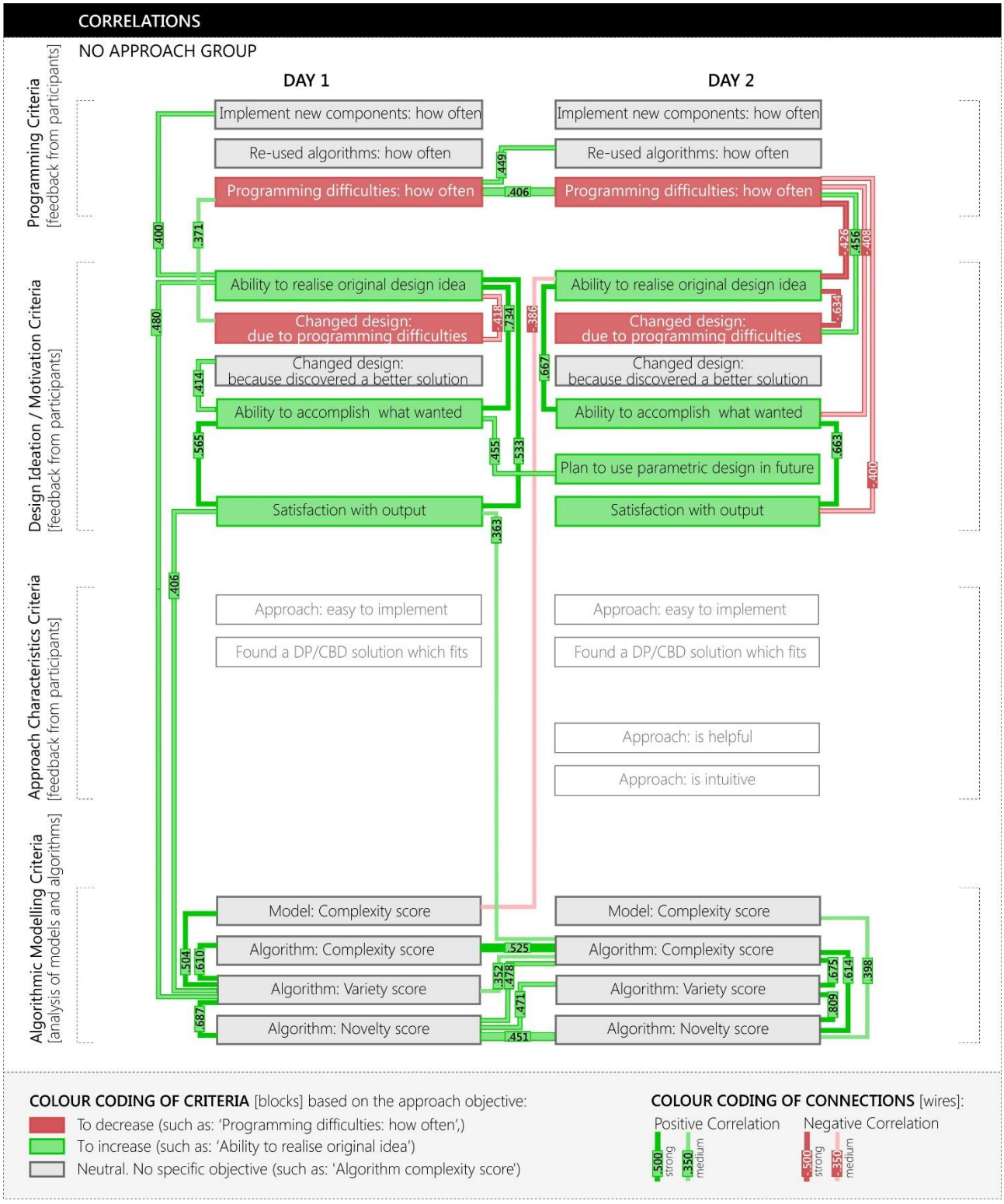


Exhibit B21. Control group. Correlations between all criteria.

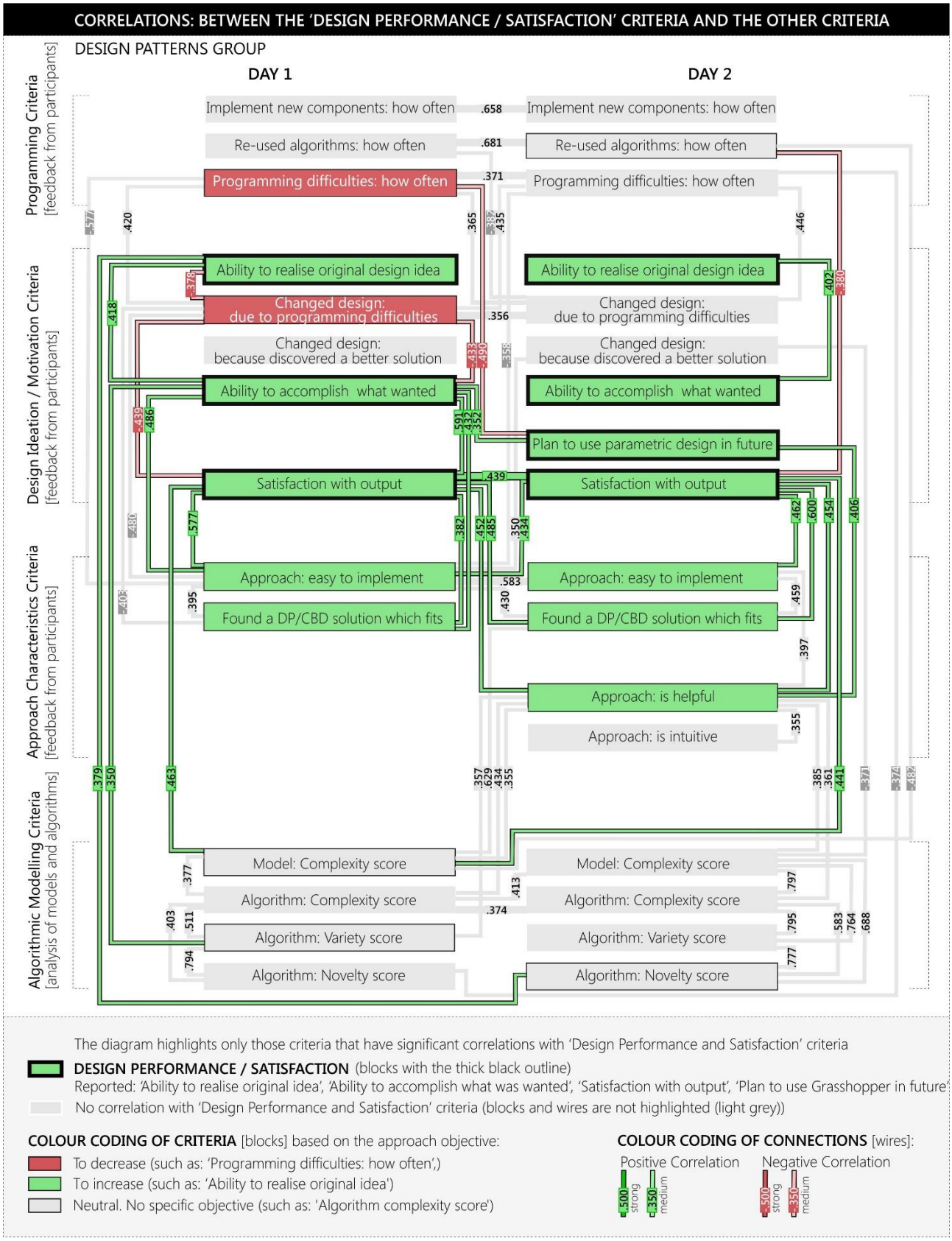


Exhibit B22. DP group. Correlations between Design Performance/ Satisfaction criteria and the other criteria.

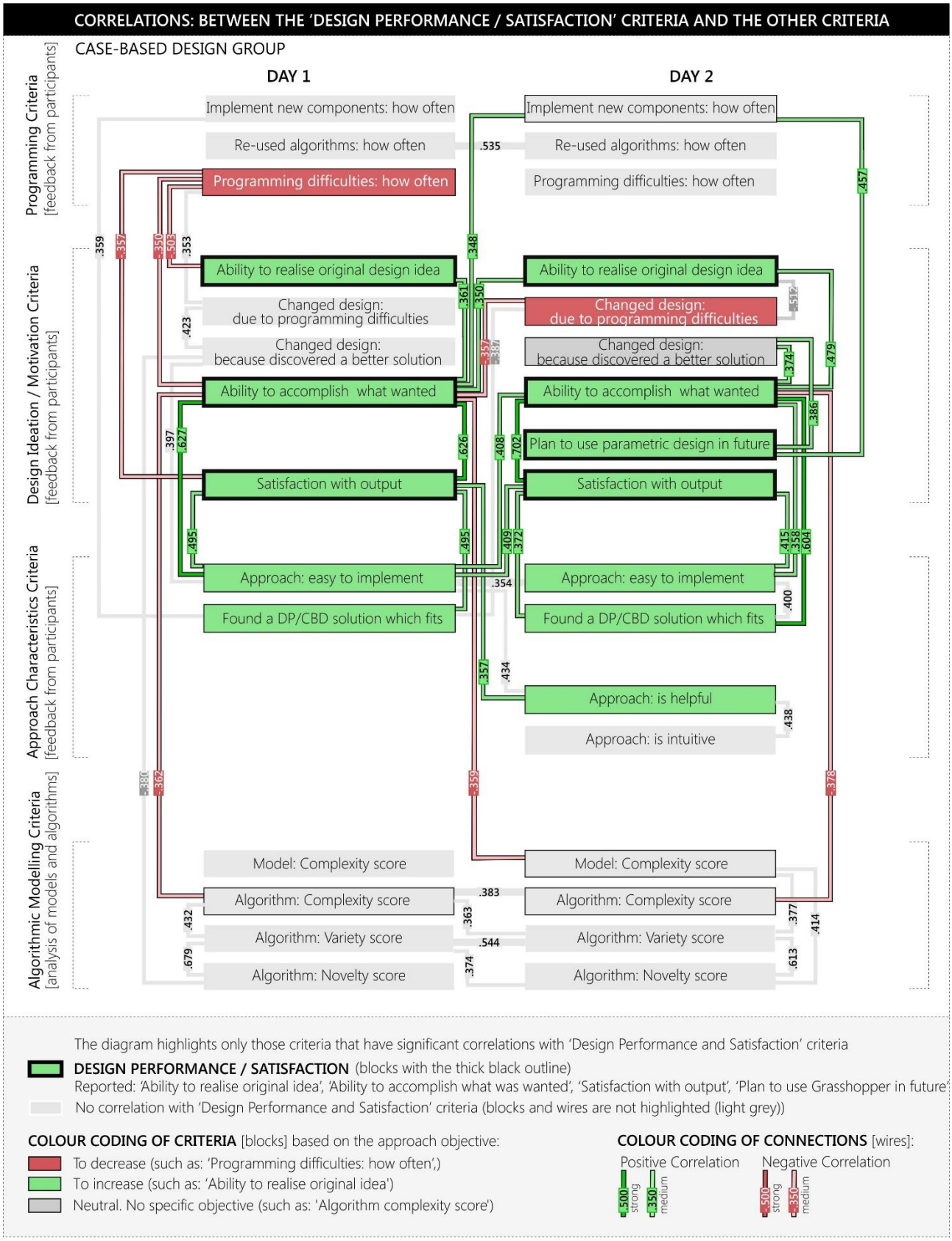


Exhibit B23. CBD group. Correlations between Design Performance/ Satisfaction criteria and the other criteria.

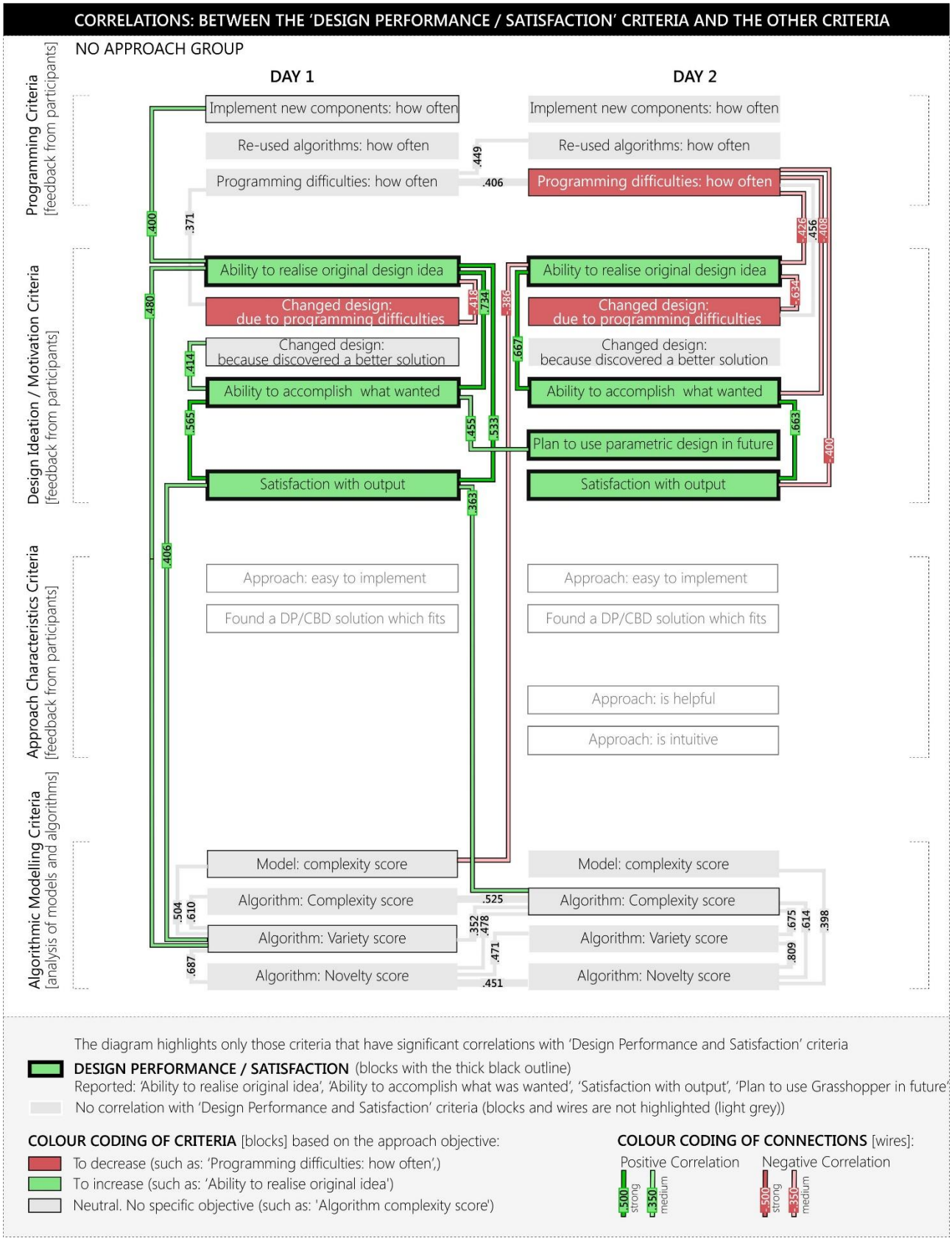


Exhibit B24. Control group. Correlations between Design Performance/ Satisfaction criteria and the other criteria.

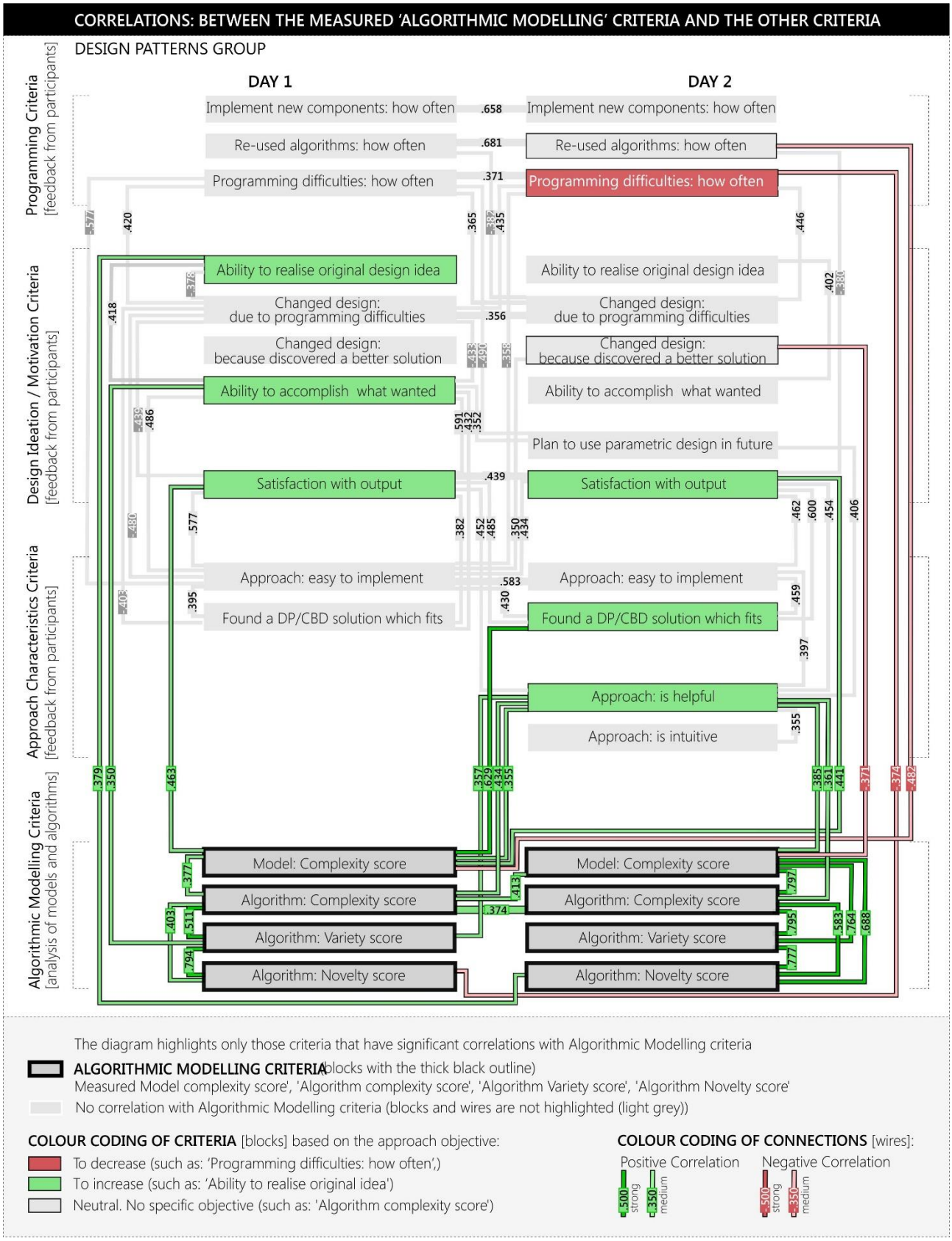


Exhibit B25. DP group. Correlations between Algorithmic Modelling criteria and the other criteria.

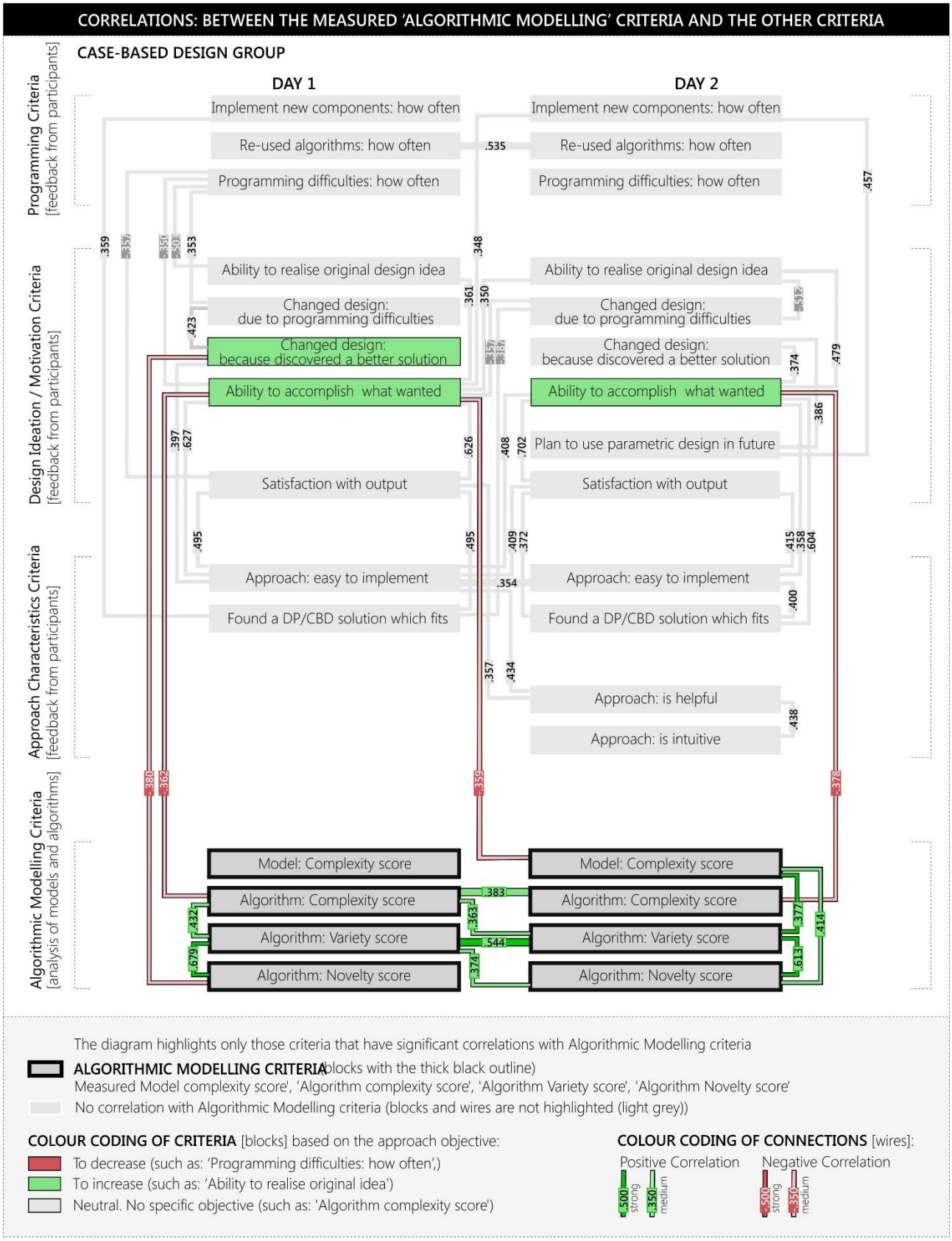


Exhibit B26. CBD group. Correlations between Algorithmic Modelling criteria and the other criteria.

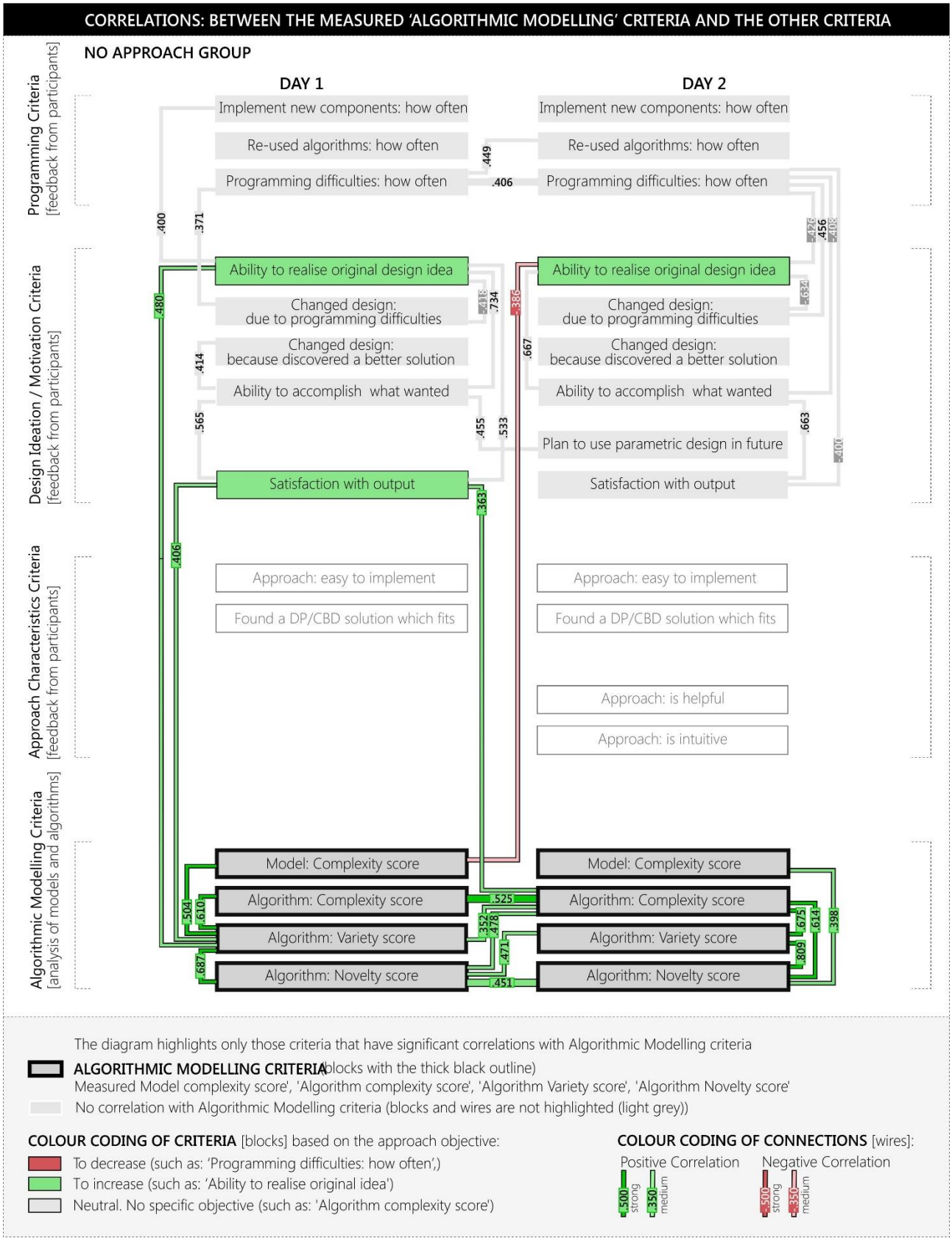


Exhibit B27. Control group. Correlations between Algorithmic Modelling criteria and the other criteria.

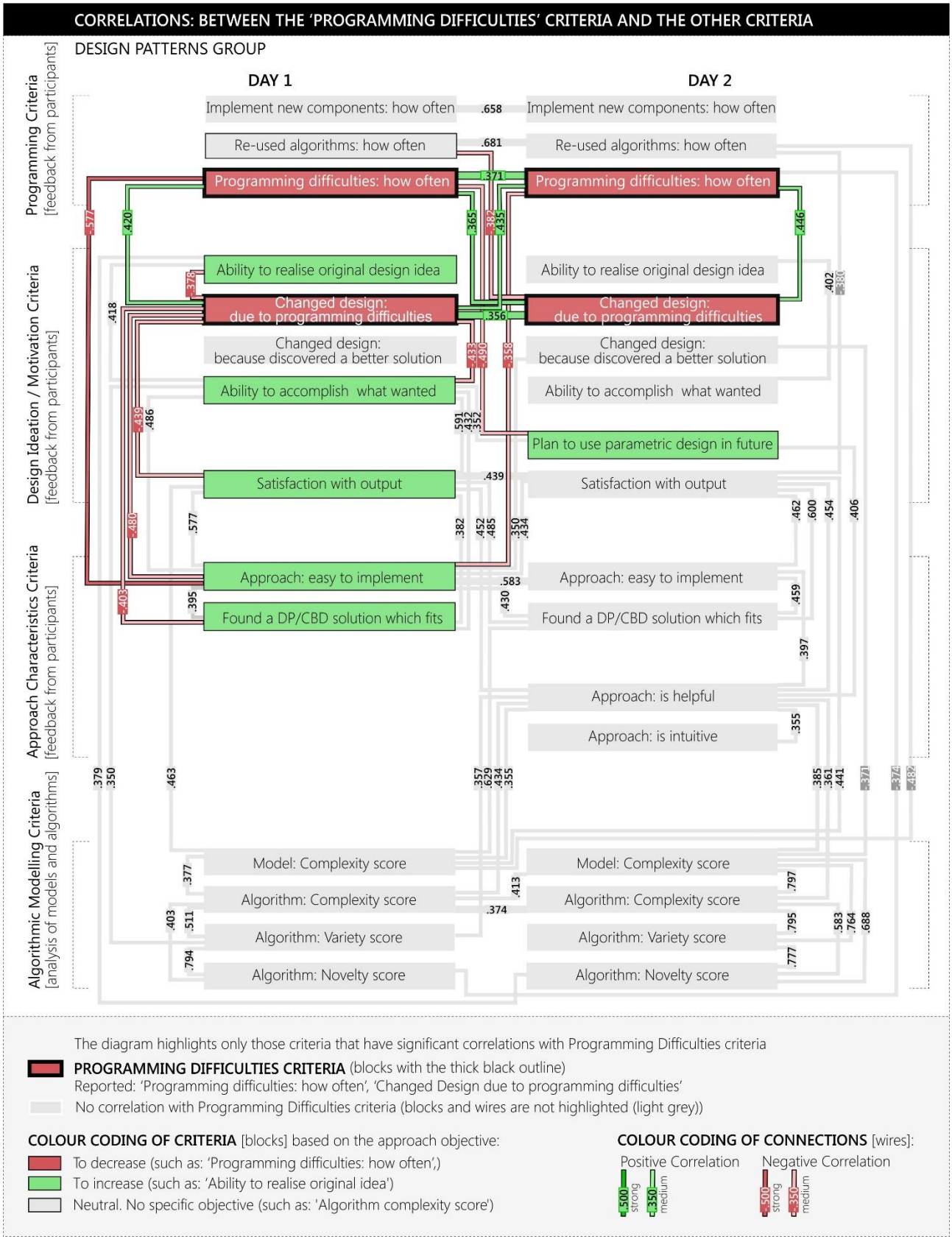


Exhibit B28. DP group. Correlations between the Programming criteria and the other criteria.

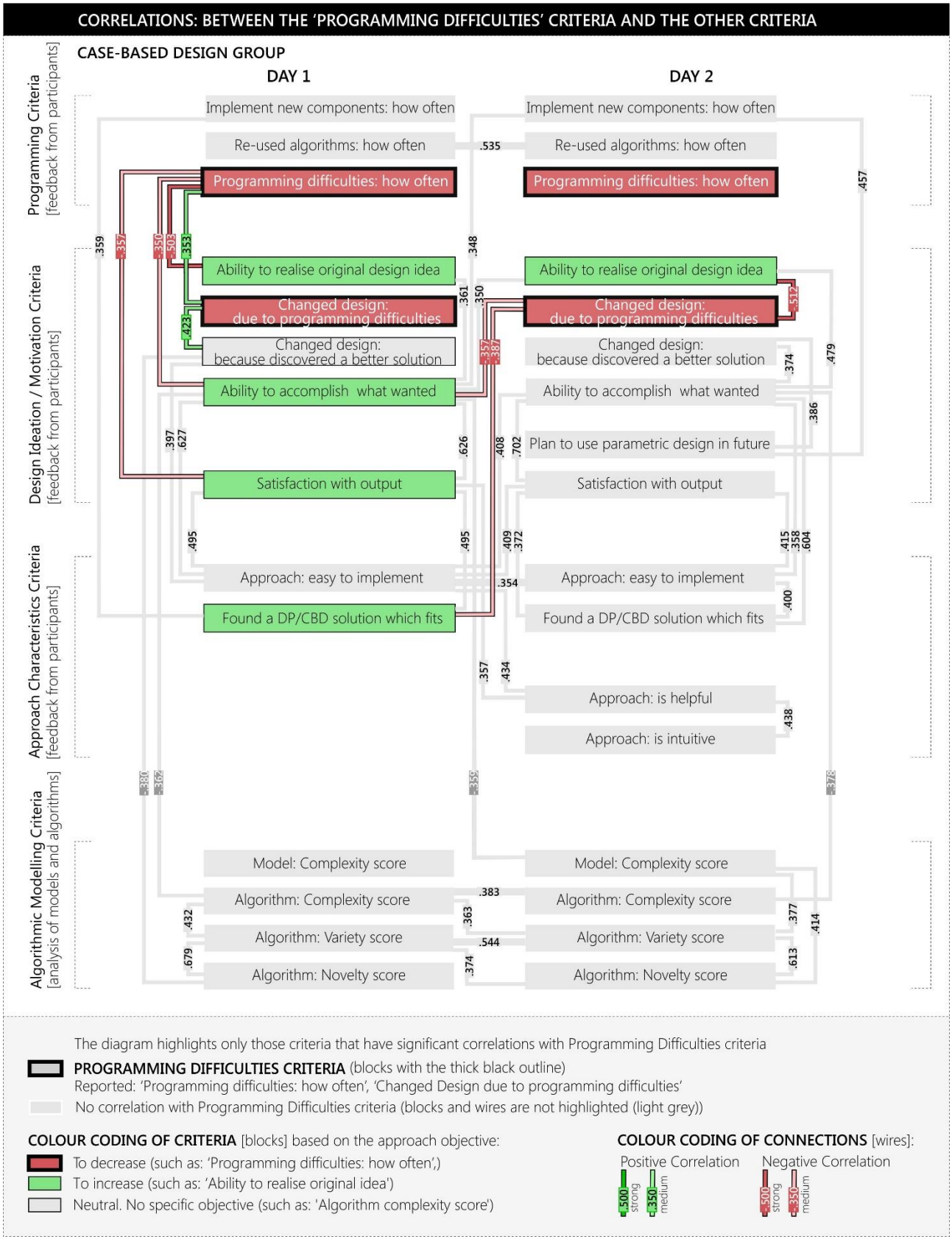


Exhibit B29. CBD group. Correlations between the Programming criteria and the other criteria.

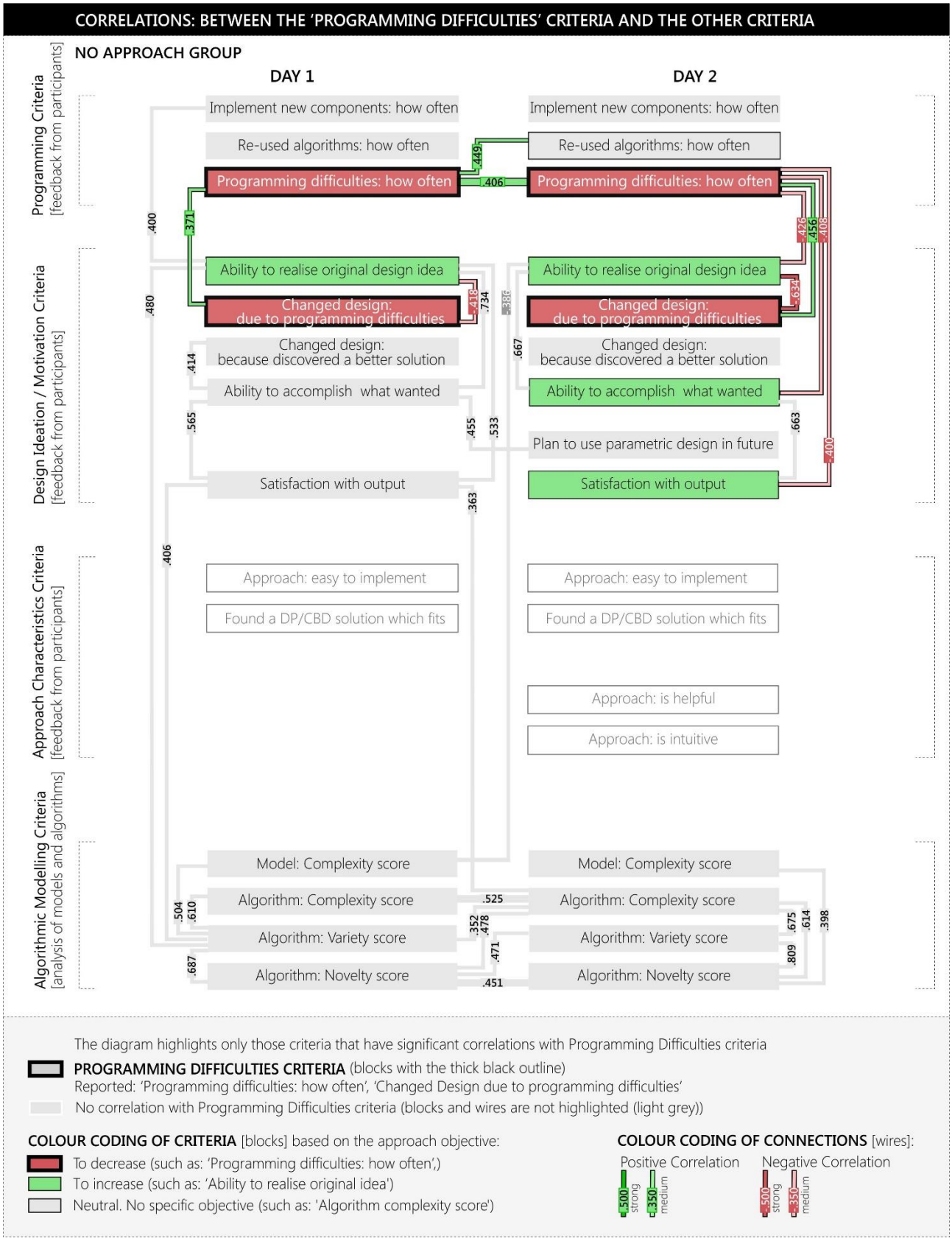


Exhibit B30. Control group. Correlations between the Programming criteria and the other criteria.

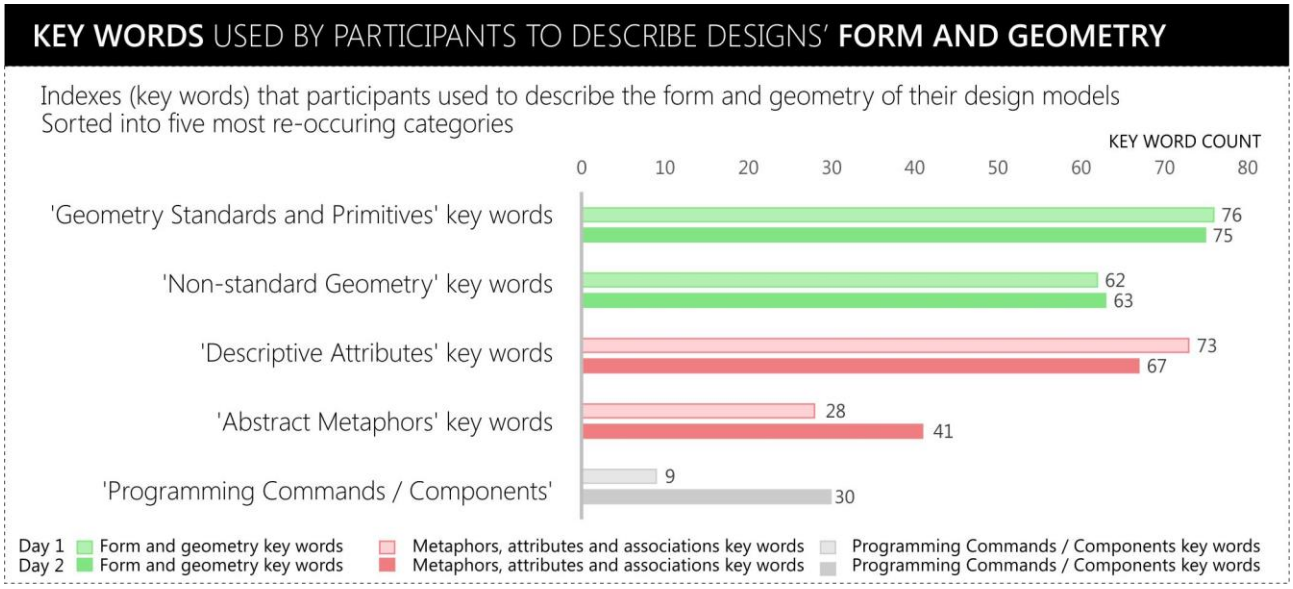


Exhibit B31. Indexing form and geometry of designs, (all groups) day 1/day 2 key words count.

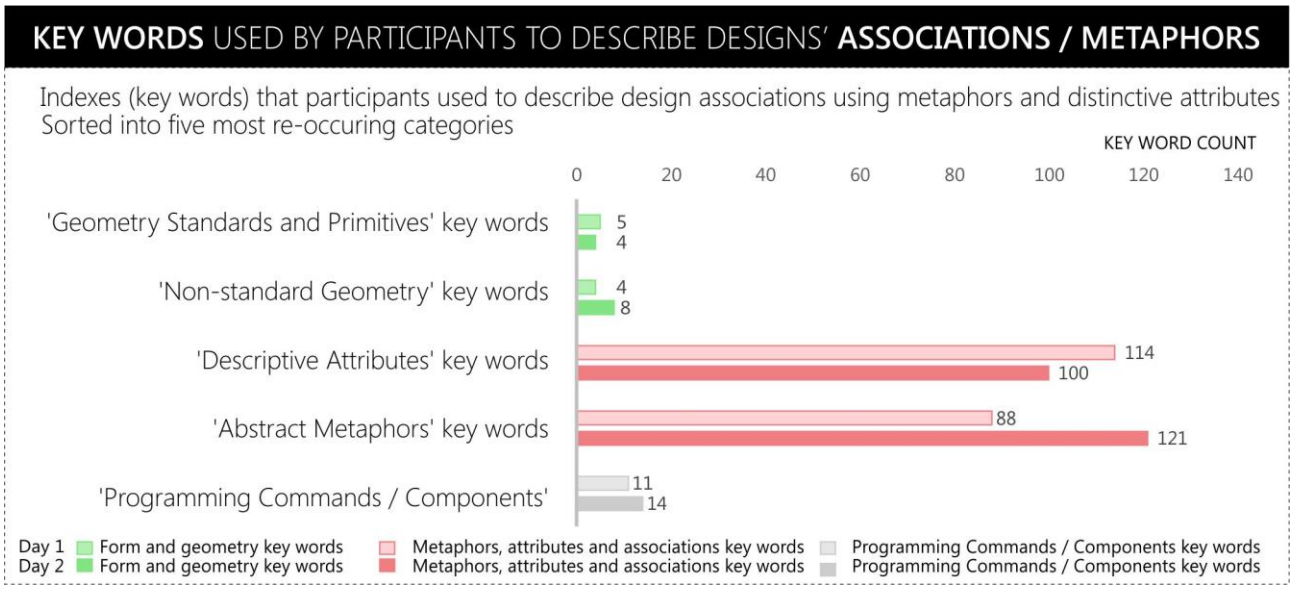


Exhibit B32. Indexing design associations using metaphors and distinctive attributes, (all groups) day 1/day 2 key words count.

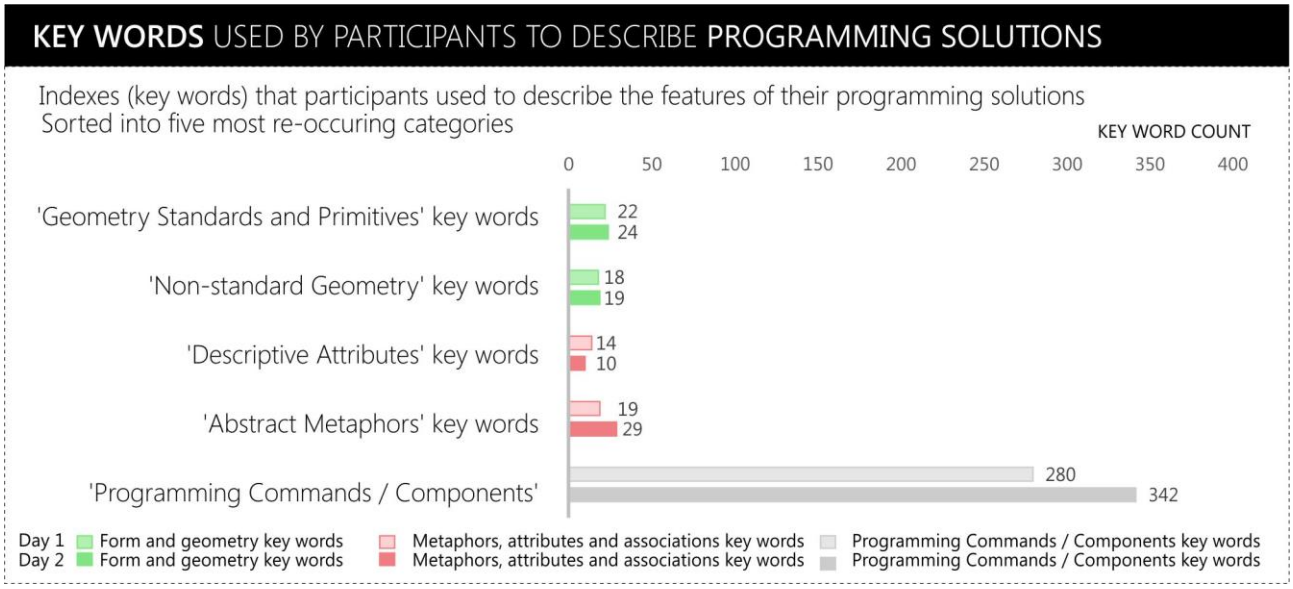


Exhibit B33. Indexing programming solutions/algorithmic modelling (all groups) day 1/day 2 key words count.

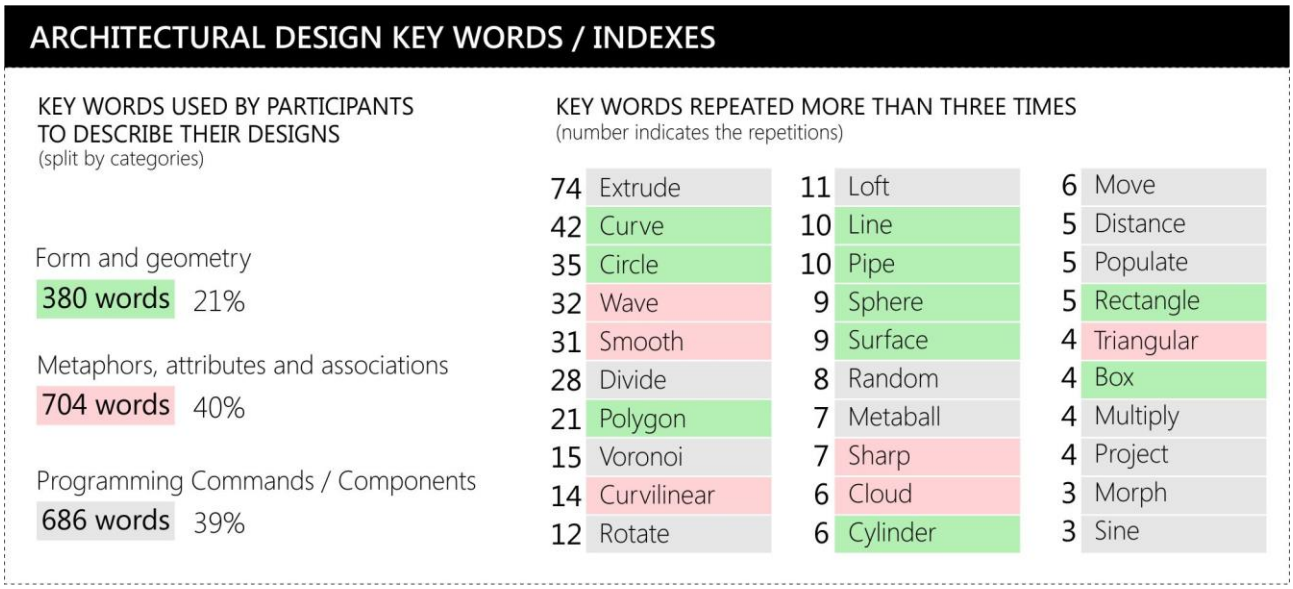


Exhibit B33. Key words used to describe parametric designs