Untangling Parametric Schemata : Enhancing Collaboration through Modular Programming

DAVIS Daniel, BURRY Jane and BURRY Mark
Royal Melbourne Institute of Technology, Australia
daniel.davis@rmit.edu.au, jane.burry@rmit.edu.au, mark.burry@rmit.edu.au

Abstract. Presently collaboration is difficult on complex parametric models, in part due to the illegibility of unstructured parametric schemata. This lack of legibility makes it hard for an outside author to understand a parametric model, reducing their ability to edit and share the model. This paper investigates whether the legibility of a parametric model is improved by restructuring the schema with modular programming principles. During a series of thinking-aloud interviews, where designers were asked to describe the function of unfamiliar schemata, the schema structured with modular programming principles were consistently better understood. Modular programming is found to be a beneficial, albeit small, change to parametric modelling that derives clear benefits in terms of legibility, particularly when the model is complex and used in a collaborative environment.

1. Introduction : Why parametric modelling can be difficult

Collaboration on a parametric model occurs through the shared language of the parametric schema. In a parametric model, the schema is the collection of relationships between functions and parameters, with the model itself being a geometric model where the form is a function of these finite parameters. The legibility of this schema is a central factor in determining how easily other team members can modify and reuse the model. An unstructured schema, represented as a graph, resembles a tangle of spaghetti, which makes it hard for team members to follow geometric relationships through the schema.

Historically parametric models have been small, making the few relationships they did contain easily understood regardless of the structure. Better software, more computational power, and the widespread adoption of parametric modelling, has seen the complexity and size of parametric models increase – the authors own recent projects have involved schemata with over 10,000 relationships. At this scale, the structure of the schema is critically important in making sense of the interwoven relationships the schema contains. This is especially true in a collaborative environment where the person trying to understand the schema might not be the person who created the schema.

Ideally the geometry of a parametric model can be modified by changing the model parameters. Understanding the schema becomes vital when the schema does not contain the parameters to modify the geometry in the desired way. When this occurs, the schema needs to be modified to include the desired parameter in a process Woodbury describes as "erase, edit, relate and repair"[14]. If the schema is tangled, knowing where to "erase, edit, relate and repair" can be difficult because the function of each node is not obvious and once the appropriate node is found, the implications for erasing this node can be very hard to trace through the many layers of interwoven relationships. On some projects making these changes can become so difficult that building a new schema is easier [3]. A number of authors have cited this as the cause of project delays, the cause of design options not being explored, and the cause of parametric modelling being relegated largely to design rationalisation [2, 3, 10, 15].

A similar problem existed in computer science during the 1960’s when unstructured programs, relying on the GOTO statement, reached a point where the program flow became so hard to understand it was feared complex programs were unmaintainable – starting over was easier than trying to maintain the code [4, 11]. One solution was organise the code with modules, which reduced reliance on the GOTO statement and increased code legibility. This paper discusses a method of applying the principles of modular programming to the organisation of parametric schemata, with the aim of investigating whether modular programming increases the legibility of parametric models. To achieve this aim, the paper reports on a series of "thinking aloud" interviews designed to assess the legibility of schemata structured with modular programming in relation to the legibility of unstructured schemata. Also discussed is the success of a website setup to exchange parametric modules. The paper begins by reporting on previous attempts to translate the benefits of computer science into parametric modelling, and why modular programming may help increase the legibility of parametric models.

2. Drawing on computer science in architecture

Computer science and architecture have an intertwined history of sharing and borrowing concepts from each other. From the outset programming was seen in relation to engineering and architecture with senior programmers adopting the title of "software architect". A well cited example of software and architecture informing each other is Ivan Sutherland’s 1963 thesis on *Sketchpad*, where he invented both Object Oriented programming and CAD/Parametric modelling – two concepts that were interdependent to Sutherland. As Object Oriented programming became today’s dominant programming paradigm, Gamma, Helm, Johnson and Vlissides (The Gang of Four) drew upon the work of architect Christopher Alexander in developing their ‘design patterns’ for structuring object oriented code [6].

The recent trend in architecture towards scripting and parametric modelling owes much to computer science. Notably The Gang of Four’s design patterns were reappropriated by Woodbury et al. into *Some Patterns for Parametric Modelling* [15]. One of these patterns, *Clear Names*, addresses the structure of the schema and will be investigated further in this paper. The other patterns solve design problems, such as how to implement recursion. Woodbury et al. work shows that the methods for structuring code, which computer scientists have spent over 50 years developing, can be applied to parametric schemata whilst retaining the benefits experienced in computer science. The primary difference between Woodbury et al. work and our work is that their patterns focus on how to structure code to solve a particular design problem whereas this paper investigates how to structure code to increase schema legibility. The following section will discuss how modular programming improves code legibility before applying this to parametric modelling.

3. Modular programming

In the late 1960’s, programming was at a crisis [5]. In some ways this crisis parallels the problems currently present in parametric modelling : the essential mechanics of programming had been developed, and the speed of computers was increasing exponentially along with the size of programs, but the unstructured nature of these programs was leading to tangles of GOTO statements, which were difficult to produce in a team environment and hard to maintain [9]. During this period it was feared that computer programs were becoming too complex for humans to write, and that this would ultimately limit the application of programming [4]. There was no "silver bullet" to the software crisis but one of the earliest and still prevalent strategies was to structure the code into modules [1].

A module in a dataflow programming language (the programming paradigm of graph based parametric schemata), defined by Wong and Sharp, is "a sequence of program instructions bounded by an entry and exit point", which performs "one problem-related task" [16]. Leaving aside the precise details of a module’s implementation, of which there are many, all modules have the general characteristics Wong and Sharp discuss :

- Modules perform one task, which is often conveyed through the name of the module.
- Modules contain defined input parameters – the only place data enters the module.
- Modules contain defined output parameters – the only place data leaves the module.
- Modules have instructions between these parameters, which can only be evoked by passing data through the module's inputs.

These changes help organise code by preventing GOTO commands threading wildly through the code and instead ensures that self-contained chunks of code only connect through designated entry and exit points, defined by the modules. The advantage of these changes was not initially apparent, and many programmers were opposed to structuring code believing it destroyed the art of programming. This opposition diminished once the benefits of modular programming became apparent [9] :

- Modules could be shared and reused because the code was self-contained.
- People could work collaboratively by developing modules separately and connecting them together later.
- Debugging could occur at the module level rather than the program level.
- The code became self-documenting – the name of the module and the inputs and outputs, gives some indication of what the module does without looking at external documentation.

Modular programming is one foundation of the structured programming movement. The concept of assembling a program from smaller, task oriented, pieces of code, persists in all modern programming languages (although depending on precise details of the implementation, a module may be called a function or an object or a method). The widespread success and adoption of modular programming in computer science, as well as the parallels between the tangles of GOTO statements and the tangles of relationships in parametric schemata, is reason to investigate whether the benefits of modular programming translate to parametric modelling.

4. Applying modular programming to parametric architecture

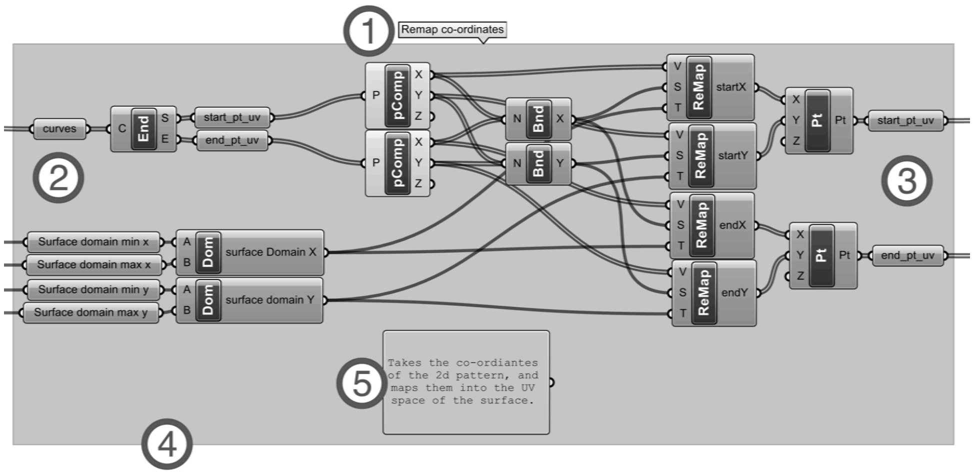

Fig. 1. A typical module in a graph based parametric model.

The principles of modular programming translate to graph-based parametric schemata. Figure 1 shows an example module, in a graph based schema. Like with Wong and Sharp’s modules, the key characteristics of this module are :

1. **Title.** The module performs one task, which is identified with a concise and descriptive title.
2. **Inputs.** Any relationship into the module are defined and grouped on the right. This single point of entry clearly displays what data the module requires, and enables the easy redefinition of relationships between the inputs and other nodes.
3. **Outputs.** Any data returned by the module is grouped to the right, providing a single point to gather the data the module produces, and like with the inputs, the easy redefinition of relationships.
4. **Group.** The whole module is grouped together, making it easy to reuse by copy and pasting. The colour of the group makes it easy to identify within the entire schema.
5. **Description.** The module contains a short description of how it works to help anyone modifying its function.

In parametric modelling, like with computer science, modules appear so simple they almost seem self-evident. However, an informal survey of parametric models indicates that most architects do not apply any of the techniques for modularising a parametric model shown above and instead prefer to leave their model unstructured like in Figure 2. This is not because parametric software inhibits structure; some, such as McNeel's Grasshopper 0.8, encourage it through features like clustering and grouping. Therefore, while modular programming is a basic modification to existing parametric models, the underuse of this technique, and the evidence it is beneficial in other domains, warrants an interest in whether these benefits translate to parametric modelling.

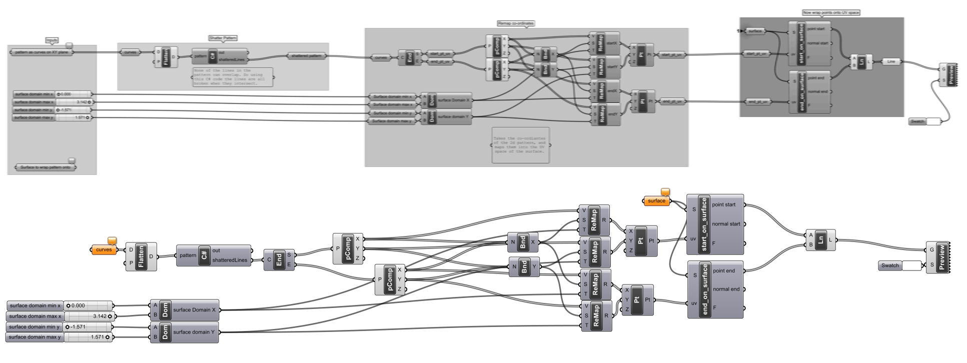

Fig. 2. The module from Fig. 1 linked with other modules (above). The equivalent Schema without modules (below). These schemata are Schema-A1 and Schema-C1 respectively, referred to in the next section

5. Evaluating parametric models constructed with modular programming

5.1. Method

To ascertain the legibility of schemata structured with modular programming principles, relative to the legibility of unstructured schemata, we conducted a series of "thinking-aloud" interviews. The participants were given a schema and asked to verbally describe how the schema parameters manipulate the geometry, which gives an understanding of how legible they found the schema.

Thinking-aloud interviews are an interview method commonly used in computer usability studies [13]. Typically the user is asked to perform a task in a software package and describe "things they find confusing, decisions they are making" and what they are reading [8]. The method provides an insight into how users carry out a task and what the users find difficult and easy about the task, although users are unreliable sources for understanding why this is [8]. In this case, the task is to describe the functionality of a schema and the data gathered gives an insight into how a designer comes to understand what a schema does.

The participants were selected from a group of 25 students studying architecture at the Royal Danish Academy of Art, who were attending a weeklong workshop on parametric modelling. On the final day of the workshop, four students were randomly selected to take part in the interviews. Using four participants provides a statistically significant sample according to usability expert Jacob Neilson (who uses thinking-aloud interviews in his own research) [12]. The selected students each had between one and seven years experience using computers to design architecture, although they all had only one years experience using parametric software – making them competent modellers, but by no means experts. As will be discussed shortly, none of the participants were familiar with the workings of the schemata they were shown, so they take on a position similar to a colleague who is given another’s parametric model for the first time and needs to understand it before they can collaborate.

The interviews were conducted in private, at a computer. The computer contained the graph-based parametric software the participants used during the workshop. With this software, the interviewer opened one schema at a time for the participant. The participant could not see the geometry the schema produced, but was free to explore the graph by dragging and zooming on screen. The participants were directed to think-aloud by describing how the schema parameters manipulate the geometry. This required the participant identifying the inputs and outputs on the schema, and then describing the main geometric steps to generate the geometry. This question was chosen to expose how legible the schema is to someone preparing to "erase, edit, relate and repair", the schema and needing to understand the main stages of the schema before they do so.

Table 1. The five schemata shown.

NAME	TYPE	SIZE	NODES	RELATIONS / EDGES	FAMILIAR	EQUIVA-LENT TO
A1	Modular	Small	41	52	Yes	C1
A2	Modular	Small	33	39	Yes	C2
B	Modular	Large	121	142	No	
C1	Unstruct	Small	26	37	Yes	A1
C2	Unstruct	Small	20	26	Yes	A2

The participants saw three schemata from a pool of five possible schemata They either saw schemas A1, B and C1 or they saw schemas A2, B and C2, in that order. The combination they saw was randomly selected to help reduce the likelihood that the results would be biased by any one schema being uncharacteristically legible or illegible.

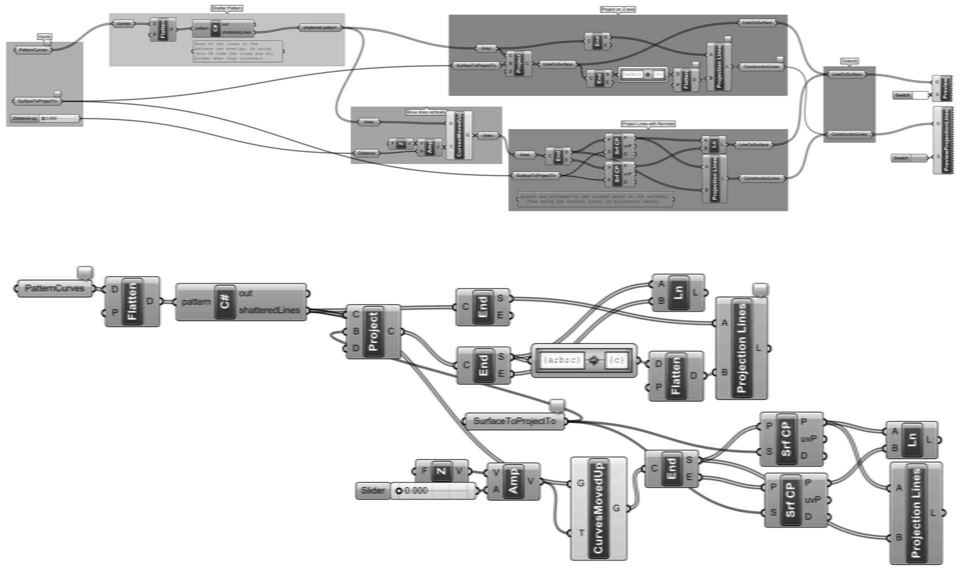

Fig. 3. The modular Schema-A2 (above) and the same schema in an unstructured form, Schema-C2 (below).

Schema-A1 and Schema-A2, were small, modular parametric models that did tasks the students had learnt about in the workshop – projecting lines onto surfaces. Schema-A1 and Schema-A2, are a modular version of the unstructured Schema-C1 and Schema-C2, meaning the structure of the schema is the only difference between A1 and C1, or A2 and C2. The participants were not aware that Schema-A and Schema-C were functionally the same, and none realised during the interviews. Schema-B was a much larger modular parametric model, which did a task the students were unfamiliar with – drawing triangles on a hemisphere from an inscribed polyhedron. It was expected that Schema-B would prove far more challenging for the participants to understand due to its size.

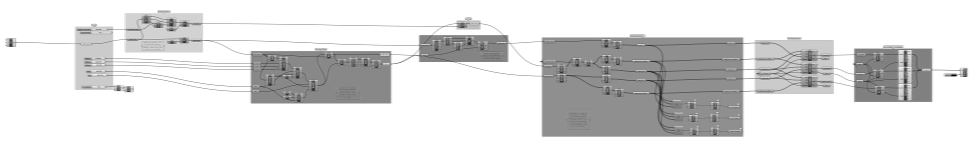

Fig.4. The much larger Schema B, with 121 nodes.

5.2. Results

5.2.1. Comparisons between schemata

When shown Schema-A, all of the participants could describe the inputs, outputs and function of the schema. Half identified all of the major stages of the schema, and the other half could identify most of them, although they often would not understand the function of specific nodes within a particular stage. Yet when shown Schema-C, no one realised it was functionally identical to Schema-A, and all guessed incorrectly what Schema-C did. On Schema-C they struggled to find the inputs and outputs, and when they did, they could say what type of geometry was represented, but no one could describe what this geometry would look like. A typical comment from Participant-2, when describing the function was : "It relaxes the lines ? That’s a guess though, because I am not sure what any of these elements [talking about the nodes] I am not sure what any of them do". We were surprised participants found it so difficult to identify the function of the nodes in Schema-C, and particularly that none realised Schema-A and Schema-C were equivalent. This demonstrates that Schema-A were drastically more legible than Schema-C.

Interestingly, the participants much better understood Schema-B, which is far larger than Schema-C and functioned on a problem they were unfamiliar with. The size of Schema-B meant it took the participants longer to trace the flow of data through the model compared with Schema A or C. However, unlike Schema-C – where the participants started guessing the functions – all participants could methodically work through the stages of the schema from inputs to outputs, although because the function of Schema-B had not been taught to the students, some struggled to understand why the schema performed these operations.

Instinctively it seems a larger schema would produce a less legible schema. Indeed a catalyst for this research was the assumption that parametric models were becoming larger as architects embraced parametric modelling, and the size of some models had reached a point where sharing a model was difficult because the number of connections made it illegible. Significantly, and against our initial assumptions, this study has shown that the size of a schema is not necessarily a measure of its legibility; it is possible to have a large and legible schema, provided the schema is well structured. Scale still plays a role, and the smaller Schema-A was more legible to the participants than the larger Schema-B. In some ways this finding is similar to findings in linguistics, where the difficulty of text can be measured as a factor of its length and its structure (as well as its vocabulary) [7].

5.2.2. Measuring legibility

While size is not the exclusive measure of legibility, it is interesting to note the ratio between nodes and edges in the more legible Schema-A and Schema-B is far lower than with Schema-C (1.23 on average vs 1.42, where 1 is the minimum possible ratio). However the Cyclomatic Complexity does not reflect this change, meaning that the number of unique paths through Schema-A and Schema-C are the same, while the number of nodes on these paths have changed. Cyclomatic Complexity is one measure of software quality used by computer scientists, primarily to plan and monitor projects. A similar metric would be desirable in parametric modelling, however measures of software quality like Cyclomatic Complexity, lines of code and Fan-in and Fan-out, do not seem to successfully distinguish between Schemata-A and Schemata-C. Our research is not conclusive enough to define an alternative quantitative metric, but it is possible to draw some qualitative guidelines for structuring legible schemata. When observing and listening to the participants try to understand a schema, the key aspects of the schema's structure that guided them were:

- **Names** : Participants regularly referred to the names of nodes and the names of node's parameters as they explained the schema. This reinforces the "Clear Names" design pattern suggested by Woodbury et al. [15].
- **Positioning** : in Schema-C where the input and output nodes are positioned amongst the other nodes, participants struggled to identify them. This lead to participants missing some of the generated geometry, or missing important inputs to the schema. In Schema-A and Schema-B, where all the inputs are to the left and all the outputs are to the right, the participants would quickly find the inputs and outputs.
- **Geometry Type** : The participants would frequently click on nodes to discover what type of geometry it produced (point, line or surface). This seemed to help them understand some operations inside the schema. While this is not strictly relevant to the structure of the schema, it is important for parametric software developers to give the users context specific hints for the parts they cannot understand.
- **Headings** : Like the names of the nodes, participants would often refer to the headings of modules as they thought aloud. Keeping these short and descriptive should aid comprehension.
- **Explanations** : Schema-A and Schema-B had explanations inside the schema of what each module did. Participants seldom read these, indicating the self documenting aspects of schema (achieved through clear names and a clear structure) is a more important than external explanations. Nonetheless, for tricky or unusual problems, explanations are still going to be necessary.
- **Color:** Interestingly two participants cited color as a major aid. The colors were not chosen to signifying anything, so it is more likely that visually separating parts of the schema – by color or by other means – visually aids a user's understanding of the relationships between modules within schemata.

The key findings of this study are that legibility is not entirely related to schema size, and those small changes to the structure of schema greatly improves legibility. The most effective changes are including clear names and grouping nodes together by function with one entry and exit point. The implications for these changes are investigated further in the discussion.

6. Sharing modular parametric models

One anticipated advantage of modular parametric modelling was increased model reuse and sharing brought about by easy interchange with existing schema, since the modules are self-contained solutions to specific problems. To enable people to share modular parametric models, and to track what was exchanged, we set up the website parametricmodel.com, shortly before the Copenhagen workshop, in November of 2010. This website was subsequently opened to the public, allowing anyone to share parametric modules.

The upload function of the website prompts users to use modular programming principles by asking them to define the inputs, and outputs, the problem the module solves and how it works, along with uploading the module itself and an image of it. These prompts were chosen to strike a balance between being prescriptive in the structure and minimizing the barriers to uploading a module. In the downloading section of the site, the pages deliberately resemble the documentation that typically comes with programming modules : it starts with a short blurb, notes the inputs and outputs and then enters into a detailed description of how the module works. If a user improves a module, they can upload their improvement – much like Wikipedia – although this functionality is outside the scope of the present discussion.

At the time of writing (March 2011), 15 authors have contributed 35 modules, which have been downloaded by 7000 unique people over the four months the site has been running. The rate of downloading indicates other designers find the modules useful, and this is confirmed to some extent by the number of times the site has been shared on social media sites.

The number of uploads, while relatively small, is larger than the 27 uploaded to the Grasshopper forum during the same time period[1] (the site where graph based parametric models are normally shared). Parametricmodel.com is by no measure a huge success, but its significance is in demonstrating how easily modules can be shared. If structuring parametric models into modules becomes normal, then reuse (which is currently rare in parametric modelling) could be enabled by the ease

[1] This is the number of modules uploaded to the "Sample and Examples" section of the site, after removing all the posts that were actually questions.

with which modules can be extracted and shared. The success of sharing modules depends in part upon the success of modular parametric modelling, and largely on resolving intellectual property rights and peoples motivations to share modules. Parametricmodel.com is one example of how module reuse and sharing may occur.

7. Discussion : The implications for designers

On the whole, the results of this research should be seen as good news for designers: relatively minor changes to the way they structure their graph based parametric schema can greatly increase the legibility of a model, making it easier to share a model collaboratively. The most valuable modifications appear to be :

1. Grouping together nodes that perform a particular task and in doing so designate data entry and exists for the group – forming a module.
2. Clearly naming the module, the nodes and the data parameters.

The major question is whether designers have the time, or inclination, to fuss with the structure of their schemata in this way. Woodbury asserts in *The Elements of Parametric Design*, that most people using parametric models are amateur programmers, and that "amateurs satisfice - they leave abstraction, generality and reuse mostly for 'real programmers" [14]. This claim appears to be based on the assumption that structuring the schema gets in the way of actually designing. It is worthwhile pointing out that the introduction of structured programming in computer science was initially met with resistance from programmers who thought "engineering" the structure of code would destroy the flow and the art of programming [9]. For computer science, the debate of whether structure interferes with design became slowly resolved in practice as structured programs took on ever more difficult and complex tasks [5]. Today, structuring code in programming enables designs that would be too complex otherwise. For architects, like programmers, the question of whether architects will structure their schemata is likely to be resolved in practice. There are many arguments for why architects might not be inclined to structure parametric schemata, but if structure provides a way of collaborating on a design that would be too complex otherwise, architects may have no option but to become "real programmers" and learn about abstraction, generality and reuse.

The two drivers for this change will be how much value is placed on complexity in the future, and how much more benefit can be extracted from structuring parametric schemata. Currently complexity is fetishised in architecture with projects often expressing it visually. If parametric modelling is to become central to the design process, then it will be necessary to deal with complexity – not, as it is now, for complexities sake – but because modelling a building is necessarily complex, particularly in a collaborative environment. Structuring the schema is one way to make this increased complexity more legible. How much more legible the schema can become depends on the further development of these techniques. In computer science, structuring code has met ever diminishing returns, and the same is likely in parametric modelling. In this sense, modular programming is probably an outlier, and the translation of other, more difficult, techniques are less likely to produce such a clear improvement. The other obvious candidate for translation would be to enable instancing of groups rather than just copying them. Whether architects end up structuring their programs depends; it depends on whether the benefits of structure, namely increased complexity and better collaboration, are seen as beneficial by architects, and it depends on how much further development of structuring techniques can improve parametric modelling.

No matter what the outcome is for structured programming in parametric schemata, our research shows that structuring parametric schemata with modular programming principles brings about immediate benefits in terms of legibility. Some architects may be reluctant to do this, but for those making large models in collaborative environments, we recommend implementing modular programming.

8. Conclusion

Based on this research, modular programming appears to increases the legibility of parametric schemata. Creating a module in a graph-based parametric model essentially involves grouping nodes together based on the task they perform, providing a single set of inputs to invoke these nodes and providing a single set of outputs to retrieve the data, as well as giving clear names to nodes and parameters. All of these changes are relatively minor, with the benefit in legibility being particularly pertinent in collaborative environments where the model is being shared amongst many people.

9. Acknowledgements

This research was funded by the Australian Research Council discovery grant "Challenging the Inflexiblity of the Flexible Digital Model" lead by Mark Burry, John Frazer and Jane Burry. The parametric workshop was part of Mark Burry's VELUX visiting professorship to the Royal Danish Academy of Fine Arts in association with the Center for IT and Architecture.

References

1. Brooks, F. (1975). The mythical man-month : essays on software engineering. Addison Wesley Longman Inc.
2. Burry, J., & Burry, M. (2006). Sharing hidden power - Communicating latency in digital models. In eCAADe Conference Proceedings. Volos, Greece.
3. Burry, M. (1996). Parametric Design and the Sagrada Família. Architectural Research Quarterly 1, N° Summer : 70-80.
4. Dijkstra, E.W. (1968). Go To Statement Considered Harmful. Communications of the Association for Computing Machinery 11, N° 3 : 147-148.
5. Dorfman, M. & Thayer R., (eds.). The Software Crisis. In Software Engineering, 1-22. Wiley-IEEE Computer Society Press, 1996.
6. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). Design Patterns. Edited by Addison-Wesley Pub Co. Elements. Addison-Wesley.
7. Graves, M. (2003). Scaffolding Reading Experiences : Designs for Student Success. Reading. 2nd ed. Christopher-Gordon.
8. Lewis, C. & Rieman, J. (1993). Task-Centered User Interface Design : A Practical Introduction. Shareware book available at http://hcibib.org/tcuid/tcuid.pdf.
9. Mall, R. (2004). Fundamentals of Software Engineering. 2nd ed. Prentice-Hall.
10. Monedero, J. (1997). Parametric design. A review and some experiences. In Challenges of the Future (15th eCAADe Conference Proceedings). Vienna (Austria).
11. NATO Science Committee. (1968). Software Engineering. Garmisch : NATO Science Committee.
12. Nielsen, J. (1994). Guerrilla HCI : Using Discount Usability Engineering to Penetrate the Intimidation Barrier. Edited by R.G., Bias & D.J., Mayhew. Costjustifying usability : 245-272.
13. Nielsen, J. (1993). Usability Engineering. San Diego : Morgan Kaufmann.
14. Woodbury, R.F. (2010). Elements of Parametric Design. Oxon : Routledge.
15. Woodbury, R., Aish, R. & Kilian, A. (2007). Some Patterns for Parametric Modeling. In 27th Annual Conference of the Association for Computer Aided Design in Architecture, 222-229. Halifax, Nova Scotia.
16. Wong, Y.K. & Sharp, J. (1992). A Specification and Design Methodology Based on Data Flow Principles. In Dataflow computing : Theory and Practice, edited by John Sharp, 37-79. Norwood : Ablex Pub.