Understanding visual scripts: Improving collaboration through modular programming

Daniel Davis, Jane Burry and Mark Burry

Understanding visual scripts: Improving collaboration through modular programming

Daniel Davis, Jane Burry and Mark Burry

Abstract

Modularisation is a well-known method of reducing code complexity, yet architects are unlikely to modularise their visual scripts. In this paper the impact that modules used in visual scripts have on the architectural design process is investigated with regard to legibility, collaboration, reuse and design modification. Through a series of thinking-aloud interviews, and through the collaborative design and construction of the parametric Dermoid pavilion, modules are found to impact the culture of collaborative design in architecture through relatively minor alterations to how architects organise visual scripts.

1. INTRODUCTION: WHY VISUAL SCRIPTING CAN BE DIFFICULT

A rapidly expanding group of architects and architectural students have embraced visual scripting as a means of constructing parametric models. The origins of visual programming can be traced back to the GRAIL system in 1969 [1]. The use of visual programming in the modelling and design of architecture was rare prior to the advent of Generative Components™ around 2003 and subsequently Grasshopper™ in 2007. Much like text-based scripts, visual scripts allow the designer to specify a sequence of relationships and operations to automate the construction of geometry. Ideally the visual script facilitates the exploration of design options by allowing the designer to change the inputs to the script and thereby sculpt the geometry into the desired shape. On occasion the geometry will not flex into the desired shape because there is no appropriate input for the modification. When this occurs, the designer needs to reorganise the structure of the script to include the required parameter in a process Woodbury calls "erase, edit, relate and repair" [2]. In a visual script it can be difficult to know what to erase, where to make the edit, and what needs relating and repairing, because the visual tangle of relationships within the script can obfuscate – rather than reveal – the function and relations of operations. This is compounded in a collaborative environment where the designer modifying the script may not be the original author, making it difficult for them to uncover the design intentions within the interwoven relationships of the script. On some parametrically modelled projects, making these types of changes can become so difficult that building a new script is easier [3]. A number of authors have cited this as the cause of project delays and the cause of design options not being explored [3-5].

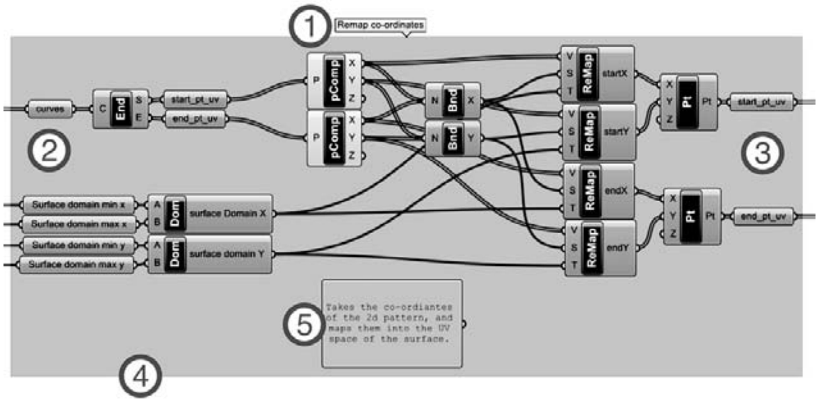
A similar problem existed in computer science during the late 1960's when unstructured programs, relying on the GOTO statement, reached a point where the relationships within the program became so difficult to understand it was feared complex programs were unmaintainable – starting over was easier than trying to maintain the code [6-7]. One solution was to organise the code with modules. It is a solution that has persisted with modules becoming a fundamental method of abstracting complicated text-based code to make it more comprehensible. However, a survey of approximately 2000 visual scripts created by designers and reported here, indicates that designers are currently much less likely to use the time honoured method of creating modules to organize and manage their visual scripts than programmers to organize their programs. This is a peculiar situation, particularly when modules seem to address the specific difficulties architects are experiencing with collaborative authorship and modifications of tangled scripts.

The aim of this article is to understand how the modularisation of visual scripts relates to the culture of architectural practice, with regard to collaborative design and the modification of geometry. The principle benefactors of this research are the increasing numbers of architects and architecture students using visual scripts to address complicated architectural problems in collaborative design environments. The two research methods employed to address this aim are: a series of 'thinking aloud' interviews to compare the comprehension of unstructured visual scripts with modularised visual scripts; and a reflective practice account of using modularised visual scripts during the collaborative design of the Dermoid pavilion. McNeel's Grasshopper™ with its wide and rapidly increasing uptake by architects, and accessible usage data, is adopted as the exclusive basis of the study. This imposes particular characteristics and limitations on the implications of the research and types of models interrogated, which we also acknowledge and explore in the paper.

2. STRUCTURED PROGRAMMING

By the late 1960's programming had come of age: the essential mechanics of programming had been developed, the speed of computers was increasing exponentially, and more code was being written. Yet despite these advancements, programmers were struggling to produce and maintain collaboratively written code, in part because the GOTO statements would often tangle and become 'spaghetti code'[8]. During this period it was feared that computer programs were becoming too complex for humans to write and human cognition would be the limiting factor in the application of computation [6].

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 [9]. A module, defined by Wong and Sharp, is "a sequence of program instructions bounded by an entry and exit point," which performs "one problem-related task" [10]. Translated into a visual scripting tool architects use, Grasshopper™, a module might resemble the graph in Figure 1.



```
Takes the co-ordiantes
of the 2d pattern, and
maps them into the UV
space of the surface.
```

◀ Figure 1: A typical module in Grasshopper™. The grey boxes are operations that have been linked together to form a larger module, with a set of inputs (2) and a set of outputs (3). Note that while each grey box is itself a module, the code encapsulated within is unable to be modified by a user of Grasshopper™.

Leaving aside the precise details of a module's implementation – of which there are many – all modules, including the one shown in Figure 1, have the general characteristics Wong and Sharp discuss:

- Modules perform one task, which is often conveyed through the name of the module (1, 5).
- Modules contain defined input parameters – the only place data enters the module (2).
- Modules contain defined output parameters – the only place data leaves the module (3).
- Modules have commands between these parameters that can only be evoked by passing data through the module's inputs (4).

These changes help organise code by creating self-contained chunks of code, at a comprehensible size, linked together through designated entry and exit points rather than with unordered threads of GOTO commands. The advantage of these changes was not initially apparent and many programmers were opposed to structuring code believing that it destroyed the art of programming. This opposition diminished once the benefits of modular programming became apparent [8]:

- 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.

The principles of text-based modularisation translate to visual programming. The abstraction attained by modularisation is considered a "standard remedy" for improving clarity in visual programming languages [11].

3. THE STRUCTURE OF EXISTING VISUAL SCRIPTS

The most comprehensive text on how to structure visual scripts in architecture is a paper by Woodbury, Aish and Kilian, entitled *Some Patterns for Parametric modeling* [5]. The paper riffs on the seminal book *Design Patterns* by Gamma, Helm, Johnson and Vlissides [12] (itself is based upon the work of architect Christopher Alexander). *Design Patterns* focuses on methods of structuring code to address problems with the code itself – such as reuse, readability and extendibility. In contrast, *Some Patterns for Parametric modeling* (later published in *Elements of Parametric Design* [2]) presents patterns that solve problems specific to architecture – such as ordering points, projecting geometry and selecting objects. The 'Clear

Names' pattern is the only one to addresses a problem with the visual script itself.

Clear Names advocates naming objects with "clear, meaningful, short and memorable names" [2]. There is an obvious benefit to knowing what a parameter does and it generally takes little effort to name a parameter. However in a survey of 1982 visual scripts publicly shared by 575 designers on the McNeel's Grasshopper™ online forum over a 29-month period [13], 56% of scripts have no named parameters. In part this is an artefact of visual programming where, unlike with text-based programming, often variables can be instantiated without naming them. It may also be an issue of education and experience. It could even be that Clear Names are not particularly useful, although since the designers on the forum are giving the scripts to their peers, they have an incentive to make the script more legible than they might otherwise.

The same can be seen in the modularisation of parametric models on the Grasshopper™ online forum. Most visual scripting languages for architects already have tools for creating modules; referred to as 'features' in Bentley's Generative Components™, called 'digital assets' in Sidefx's Houdini™ and 'clusters' in McNeel's Grasshopper™. For the same group of models exchanged on the Grasshopper™ online forum, 97.5% contained no clusters. Like with Clear Names the likely cause of this low use of modules might be education, or nuances in the Grasshopper™ software, or perhaps even because architects do not find modules useful in their visual scripts.

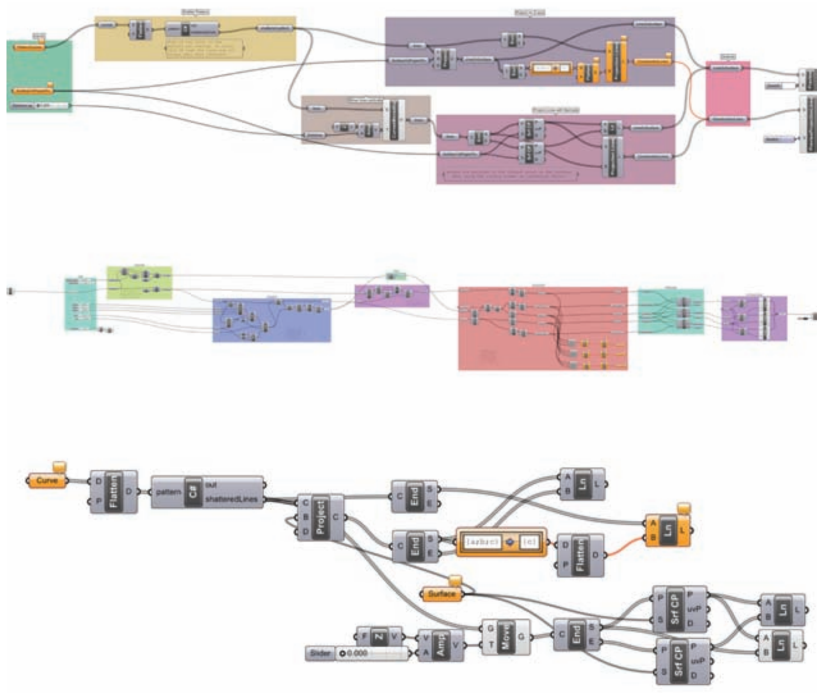
These results support Woodbury's assertion that designers leave "abstraction, generality and reuse mostly for 'real programmers'" [2]. Woodbury suggests this is due to motivation, with architects wanting to design with visual scripts rather than learn to structure them like 'real programmers' [2]. However with very little published about how architects can structure visual scripts, and with no discussion of how script structure affects the culture of design or the architectural outcome, designers may not readily have enough information to make an informed decision.

4. COMPREHENSION OF MODULAR VISUAL SCRIPTS

4.1. Method

To understand how architects comprehend modular visual scripts, we conducted a series of 'thinking-aloud' interviews. These interviews were used to compare the legibility of scripts structured with modular programming principles relative to the legibility of unstructured scripts. Thinking-aloud interviews are an interview method commonly used in computer usability studies to understand how users carry out a task [14-15]. Typically, participants were asked to perform a task in a software package and describe "things they find confusing, decisions they are making" and what they are reading [14]. In this case each participant was asked to explain the functionality of an unfamiliar visual script by describing how the script inputs control the geometry. Participants could not see the geometry the script produced but were free to explore the graph by dragging, zooming and clicking on screen. The participants' responses and actions give some insight into how designers attempt to understand an unfamiliar script and how legible designers find the various visual scripts.

The participants in the study were randomly selected from a group of 25 students studying architecture at the Royal Danish Academy of Fine Art and attending a weeklong workshop on parametric modelling. Four students were selected, based on usability expert Jacob Neilson's recommendation to use between 3-5 participants in thinking-aloud interviews [16]. The selected students each had between one and seven years' experience designing architecture with a computer, although they all had only one year's experience using visual scripts – making them competent enthusiasts but by no means experts. Being familiar with Grasshopper™ but unfamiliar with the specific scripts they were shown, the participants were in a role similar to a designer trying to understand a script a colleague had shared with them.



► Figure 2: Three Grasshopper™ scripts in the order they were shown to the students. Top: A modularised script (2M). Middle: A large modularised script (3M). Bottom: An unstructured script (2Us), functionally equivalent to the top script (2M).

Name	Function	Structure	Size	Nodes	Edges	Familiar task	Equivalent
1M	1	Modular	Small	41	52	Yes	1Us
1Us	1	Unstructured	Small	26	37	Yes	1M
2M	2	Modular	Small	33	39	Yes	2Us
2Us	2	Unstructured	Small	20	26	Yes	2M
3M	3	Modular	Large	121	142	No	

◀ Table 1: Five scripts shown to the participants – each participant either saw 1M, 3M and 1Us or 2M, 3M and 2Us.

The participants were shown three Grasshopper™ scripts in a predefined order. The first and last were a small script that did a task the participants had learnt about in the workshop – projecting lines onto surfaces. The scripts were functionally equivalent, containing the same operations in the same order, with the only difference being the first script was a modularised version of the unstructured final script. These two scripts allowed a comparison to be made between the comprehension of a modularised script and the comprehension of an unstructured script. To mask the fact that the first script and the final script were functionally identical, the participants were shown a script in-between that was much larger and did a task the participants were unfamiliar with – drawing triangles on a hemisphere from an inscribed polyhedron. To reduce the bias from one script being uncharacteristically legible or illegible, two versions of the first and last script were created and randomly shown. Therefore of the scripts shown in Table 1, the participants were shown in order either script-1M, 3M and 1Us or script-2M, 3M and 2Us.

4.2. Results of comparison

The modular visual scripts (1M or 2M) were far easier for the participants to describe than the unstructured equivalents (1Us or 2Us). This is not a surprising result given what is already known about modularisation. What is unexpected is how poorly the participants understood the unstructured scripts. When shown the first modular script (1M or 2M), all participants could describe the inputs, outputs and function of the script. Half could describe all the script's major stages. When asked about individual nodes, they generally understood what the nodes did but sometimes could not understand what the nodes were doing within the context of the model. In contrast when shown the equivalent script in an unstructured form (1Us or 2Us) all participants guessed incorrectly the function of the script. A typical comment from Participant-2 was: "It relaxes the lines? That's a guess though, because I am not sure what any of these elements [nodes], I am not sure what any of them do [in this context]." They all struggled to find the inputs and outputs of the script and none could assemble their knowledge of the individual nodes into an understanding of the larger stages of the script. What is interesting here is not so much that modularity improves the legibility of the scripts (which is already well known) but that unstructured scripts – even small ones – are fairly incomprehensible to designers unfamiliar with them. Even the much larger modular script (3M) was better understood by the participants than the small unstructured visual script. When shown 3M all the participants could methodically work through the nodes in each module, however none could put the modules together to understand the overall aim of the script (admittedly they had not learnt about the geometry 3M produced). Significantly this shows modularisation does not improve comprehension of a script so much as it determines whether a visual script will be legible to a designer unfamiliar with it. Large scripts, if structured, can be more legible than unstructured small scripts.

4.3. Factors in Comprehension

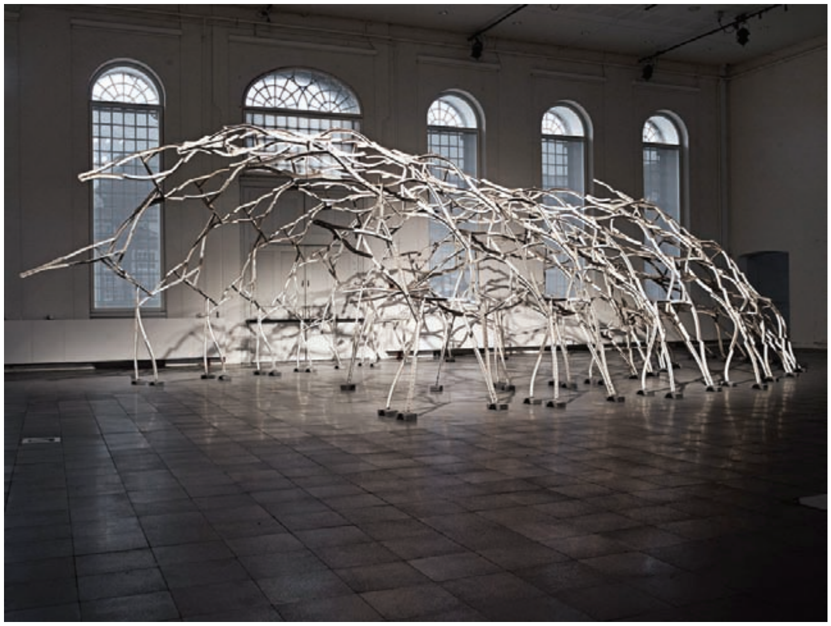
Modularisation seems to work because the overview it provides helps guide the user's understanding of successively smaller parts of the script. When observing the participants, they were rarely able to reverse this and deduce the overview through understanding the interaction of the smaller parts. This is possibly why the unstructured models were so difficult for them to understand. The paradox is that designers create visual scripts by assembling small parts of the script into a whole and yet they find it much easier to read a script starting with the overview and delving into the smaller parts. Designers are probably able to pull off this paradox because when they are assembling the smaller parts of the script it is likely they have in mind some overall understanding of what they are creating. It seems without modularisation this overall mental model of how the script works is lost, and with it the ability for an a designer not familiar with it to comprehend the script.

In addition to this preference to understand the script in a top-down rather than bottom-up way, a few other generalisations about key factors in comprehension can be made:

- **Names:** Participants regularly referred to node names and module names as they explained the script. This reinforces the 'Clear Names' design pattern suggested by Woodbury [2].
- **Positioning:** Participants struggled to identify parameter nodes and output nodes when they were positioned amongst the other nodes in the unstructured script (1Us or 2Us). When the inputs and outputs were positioned on the left and right of the script respectively the participants could readily identify them. Similarly separating the parts of the script both physically and with colour were cited by the participants as being helpful.
- **Explanations:** Some of the modules inside the modular scripts (1M, 2M or 3M) contained short explanations of what they did. Participants seldom took the time to read these, indicating a self-documented script (clear names and a clear structure) is preferable to one explained through external documentation.

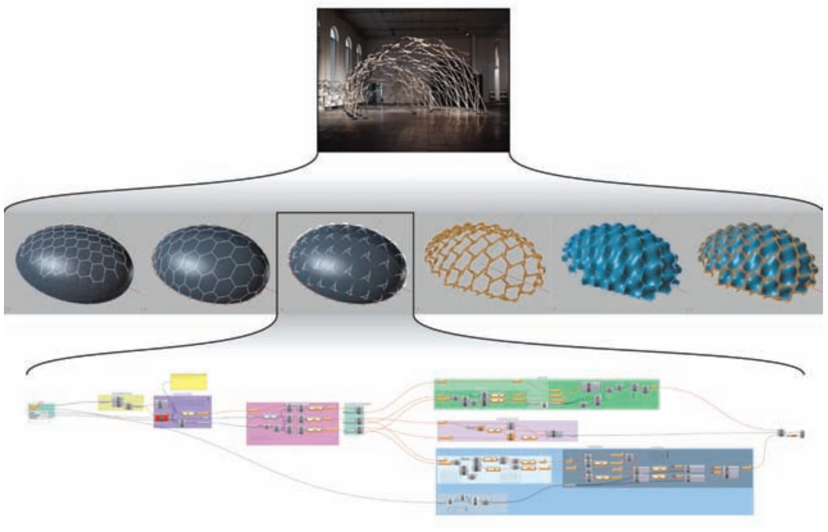
The key finding of these interviews is that the way designers come to understand a visual script is different to the way they construct them. If a designer's intention is not communicated through the names of the nodes and through the structure of the nodes, then a designer unfamiliar with the script will find it difficult – if not impossible – to assemble an understanding of the script from its component parts. Thankfully including clear names and grouping nodes together by function are relatively minor modifications to the visual script, which can substantially improve its legibility.

5. MODULES IN PRACTICE


◀ Figure 3: Dermoid installed at the 1:1 Exhibition, Copenhagen, 2011. Photo by Anders Ingvartsen.

In the design of the Dermoid pavilion, we tried to apply modular visual scripts within the architectural design process to better evaluate the impact of modularisation on the design process. The pavilion was a collaboration between researchers at the Center for Information Technology and Architecture (CITA) at the Royal Danish Academy of Fine Arts, led by Prof. Mette Thomsen, and researchers at the Spatial Information Architecture Laboratory (SIAL) at the Royal Melbourne Institute of Technology, led by Prof. Mark Burry as part of his VELUX Visiting Professorship to CITA. The collaboration occurred both remotely through shared digital files and through a series of workshops held between 2010 and 2011 at CITA in Copenhagen. Burry has discussed the rational for the Dermoid as well as the research outcomes in *Scripting Cultures* [17]. The following section will instead discuss the technical aspects of creating and using modular visual scripts to assist in the design of Dermoid.

Dermoid is constructed from a wooden reciprocal frame where the interlocking members weave under and over a doubly curved surface of continuously changing curvature, forming hexagons [18]. For structural reasons the wooden members could not twist over the doubly curved surface and could not be too short nor too long. It was necessary to model the project parametrically since many of the design constraints, such as material performance and the overall form, were not known until shortly before construction commenced (having been calculated progressively through a series of physical modelling experiments in the previous workshops). Yet the reciprocal frame in Dermoid did not lend itself to parametric modelling due to the circular relationships formed in the structure. The geometry of Dermoid presents a significant challenge to model, made more difficult by many parameters being unknown and by the collaborators being remote from each other.


► Figure 4: The modules that make up Dermoid. Top: Dermoid. Middle: The outputs of the 6 major stages. Bottom: The modular visual scripts from stage 3.

The design and modelling of Dermoid were intermixed in terms of subjects, methods and application of software. One of the early investigations looked at how patterns could be distributed evenly across a doubly curved surface (during this time the surface shape and pattern were not known). This took the form of a visual script and over time a chain of visual scripts had been developed, the first creating the underlying surface, and the rest adding details to the design until the last produced the construction documentation. Each script was a self-contained parametric model that could be thought of as a module, with a prescribed set of inputs from the previous stage and a distinct set of outputs. This structure could not have been anticipated at the start of the project because the design intentions were unclear. As the design crystallised, so too the structure emerged and evolved with the project. Breaking the project into these modules allowed different team members to work independently on different stages and swap the new versions into the overall structure without breaking the previous (and concurrent) work. Provided the stage module produced the expected outputs, designers were free to create the stage using whatever software they thought fit, which proved useful for 'wicked' stages – like beam distribution – where over the course of the project alternative strategies were developed on at least five different software packages.

The visually scripted stages were themselves broken down into modules. As with the overall structure, it was difficult to anticipate these modules prior to the creation of the scripts. Normally the scripts were constructed as an unstructured assembly of parts and then refactored once they began to work. These refactorings normally involved pruning the branches of code not contributing to the outcome and adding in new nodes to name the paths of data. The refactorings also involved grouping the nodes into modules, which could often be found by looking for places where the data was naturally channelled into one or two streams. On a number of occasions the modules were reused by designers who had not created them. Perhaps the most salient example was a designer involved only in the final months of the project who was tasked with changing the topology of the beam. He changed the topology primarily by linking existing modules together, without disrupting the overall flow of the major stages in the project. This type of reuse suggests that designers were able to interpret their colleague's modules and it demonstrates they were able to cleanly extract a module from one context and reconnect it in another context. The modules seemed to help overcome some of the difficulties of understanding, reusing and modifying unstructured visual scripts, although they are cumbersome to implement during the design and construction of the stages.

The complexities of Dermoid, both in terms of geometry and in terms of collaboration, place it on the limit of what is currently possible in parametric architectural modelling, and perhaps beyond what is practical with an unstructured visual script. Breaking the project into a hierarchy of modules made it possible for designers to collaborate using disparate software, while the smaller modules seemed to promote script reuse. At both scales, structure was found in an unstructured beginning and became legible through a few, relatively minor, changes.

6. DISCUSSION: THE IMPLICATIONS FOR ARCHITECTS

The results of this research should be seen as good news for architects: relatively minor changes to the way they structure their visual scripts can greatly increase script legibility, making it easier to share and modify the script. The most valuable additions appear to be:

- Grouping together nodes that perform a particular task and in doing so designate data entry and exit points for the group – forming a module.
- Clearly naming the module and the nodes.

The major question is whether designers have the time, or inclination, to fuss with the structure of their scripts. Woodbury's assertion that designers, being amateur programmers, leave "abstraction, generality and reuse mostly for 'real programmers'" implies that structuring a script gets in the way of actually designing [2]. To a certain extent this is true, it is difficult to define a rigid structure of modules when the overall design is still fuzzy. However visual scripts left the way they were created are extremely difficult for other designers to understand and therefore use or modify. Whether architects take the time to organise their scripts is likely to be resolved in practice, with projects like the Dermoid being on the brink of being too complex to model in an unstructured way. The implications for teachers is that architecture students need to be taught to communicate with visual scripts in addition to learning how to assemble and design with visual scripts. This involves a shift from considering visual scripts as drawing tools – the digital equivalent of a pencil and French curve – to understanding them as a form of representation in their own right. There are many arguments for why architects might not be inclined to structure visual scripts 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 resolving complexity in the future, and how much more benefit can be extracted from structuring visual scripts. The challenge of applying parametric modelling to a relatively small and logically well-defined problem, like Dermoid, foreshadows the complexity that could be expected in future visual scripts. Modularising the script is one way to make this increased complexity more legible. How much more legible the script can become depends on the further development of these techniques. The other obvious candidates for translation are instancing of modules and polymorphism, however such developments first need to be supported within architectural visual scripting tools. In computer science, structured programming has accompanied a culture of sharing common libraries of components. This research has shown that the visual modularisation of visual scripts does help to make sharing them easier. With complicated projects being designed collaboratively using visual scripts, and with better structuring techniques likely in the future, structuring visual scripts may become an essential part of successfully designing architecture with scripts.

7. CONCLUSION

The way architects create visual scripts (by assembling small parts into a whole) is at odds with how they understand them (from the whole to the smaller parts). Modularising an unstructured script is one method for communicating the overall intention of the script, thereby making the script more legible for colleagues. Creating a module is a relatively minor exercise and 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 the nodes. While these changes seem trivial, this study shows they are critical factors in the legibility of the script, and therefore the ease with which the script can be shared, reused and modified.

Acknowledgements

This research was funded by the Australian Research Council discovery grant "Challenging the Inflexibility of the Flexible Digital Model" led by Mark Burry, John Frazer and Jane Burry. The Dermoid workshop was part of Prof. Mark Burry's VELUX Visiting Professorship at Center for Information Technology and Architecture at Royal Danish Academy of Fine Arts – 2009-2011. This article draws on a paper presented at CAAD Futures 2011 [18].

References

1. Ellis, T. O., Heafner, J. F. and Sibley, W. L., *The Grail Project: An experiment in Man-machine communications*, The RAND Corporation, 1969.
2. Woodbury, R. F., *Elements of Parametric Design*, Routledge, Abingdon, 2010.
3. Burry, M., Parametric Design and the Sagrada Família, *Architectural Research Quarterly*, 1996, (Summer), 70-80.
4. Monedero, J., Parametric design. A review and some experiences, in: Martens, B., Linzer, H., Voigt, A. eds. *Challenges of the Future: 15th eCAADe Conference Proceedings*, Österreichischer Kunst- und Kulturverlag, Vienna, 1997.
5. Woodbury, R., Aish, R. and Kilian, A., Some Patterns for Parametric Modeling, in: Lilley, B. and Phillip, B. eds. *27th Annual Conference of the Association for Computer Aided Design in Architecture*, Dalhousie University, Halifax, 2007, 222-229.
6. Dijkstra, E. W., Go To Statement Considered Harmful, *Communications of the Association for Computing Machinery*, 1968, 11(3), 147-148.
7. NATO Science Committee, *Software Engineering*. NATO Science Committee, Garmisch, 1968.
8. Mall, R., *Fundamentals of Software Engineering*, 2nd edn., Prentice-Hall, New Delhi, 2004.
9. Brooks, F., *The mythical man-month: essays on software engineering*, Addison Wesley Longman Inc., Massachusetts, 1975.
10. Wong, Y. and Sharp, J., A Specification and Design Methodology Based on Data Flow Principles, in: Sharp, J. ed., *Dataflow computing: Theory and Practice*, Ablex Publishing, Norwood, 1992, 37-79.

11. Green, T. and Petre, M., Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework, *Journal of Visual Languages & Computing*, 1996, 7(2), 131-174.

12. Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Massachusetts, 1995.

13. http://www.grasshopper3d.com/forum/ [data recorded from May 2009 to October 2011].

14. Lewis, C. and Rieman, J., *Task-Centered User Interface Design: A Practical Introduction*, Self published, 1993.

15. Nielsen, J., *Usability Engineering*, Morgan Kaufmann, San Diego, 1993.

16. Nielsen, U., Guerrilla HCI: Using Discount Usability Engineering to Penetrate the Intimidation Barrier, in Bias, R. G. and Mayhew, D. J., *Cost-Justifying usability*, Morgan Kaufmann, California, 1994, 245-272.

17. Burry, M., *Scripting Cultures: Architectural Design and Programming*, Wiley, Chichester, 2011.

18. Davis, D., Burry, J. and Burry, M., Untangling parametric schemata: enhancing collaboration through modular programming, in: Leclercq, P., Heylighen, A. and Martin, G. eds. *Proceedings of the 14th international conference on Computer Aided Architectural Design*, University of Liege, Liege, 2011.

Daniel Davis, Jane Burry and Mark Burry

Royal Melbourne Institute of Technology
Spatial Information Architecture Laboratory
Melbourne, Australia

D. Davis, daniel.davis@rmit.edu.au