Lawnmower

Designing a web-based visual programming environment that generates code to help students learn textual programming

Gabriel Wurzer[1], Burak Pak[2]
[1]Vienna University of Technology, Austria, [2]Sint-Lucas School of Architecture, KU Leuven LUCA, Belgium
[1]http://www.iemar.tuwien.ac.at, [2]http://www.architectuur.sintlucas.wenk.be
[1]gabriel.wurzer@tuwien.ac.at, [2]burak.pak@architectuur.sintlucas.wenk.be

Abstract. *Learning programming can be a challenging task for design students, especially when code is to be entered in textual form. Visual programming languages, such as McNeil's Grasshopper, have helped students to engage in scripting without having to deal with lower level syntax that is often hindering them in expressing their thoughts. However, the problem with learning how to program textually is only postponed: When switching to a new platform, students may be forced to learn coding from scratch, and, even worse, to do so in a textual environment that is yet unfamiliar. Our idea is simple: Connect visual programming with textual coding, using code-generation as means. Using this approach enables students to think visually, and see the results textually. An added bonus is the possibility to use debugging, a feature that is yet lacking from Grasshopper. By this way, our language aims to enable students to gradually move from visual to textual programming in a comfortable manner.*
Keywords. *Visual Programming; Structured Code; Teaching; Code Generation.*

OVERVIEW

In this paper, we are going to present a programming platform called "Lawnmower", which is a web-based learning environment for architecture students that allows the automatic translation of visual programs into textual code. In perceiving this transformation, students visually observe the similarities and differences between flow-based and block-based programming styles and use their existing knowledge about Grasshopper when transitioning to text-based programming. The major motivations behind our proposal are:

- The limits of the existing visual programming environments and especially Grasshopper (also observed by Leitão and Santos (2011) and Cabecinhas (2010)).
- Our own findings from a survey amongst students, teachers and professionals (reported in the following sections).
- The need for facilitating social learning in programming (Celani and Vaz, 2012).
- The potentials of web 2.0-based environments for facilitating social learning (Pak and Verbeke, 2012-forthcoming).

Based on these motivations, we are going to elaborate on the following subjects to reveal the details of the proposed platform:

- As a pre-step, we will briefly discuss the Visual Programming Languages and map *the exact differences between block-based and flow-based languages*, as these affect the overall design of our visual programming language to a large extend (see "Block-based versus flow-based programming languages").
- We apply these observations when designing a combination between block-based and flow-based programming language for the sake of *generating textual code out of a scripting graph*. In this context, our work does not seek to replicate available flow-based language patterns; instead, we have conducted surveys among students and teachers and asked how they would visually represent constructs commonly found in textual programs (see "Language design").
- Another important factor is the "*social learning experience*" of our platform (see "Capabilities of Lawnmower as a Social Learning Environment"). Being web-based, various communication modes and styles are supported, e.g. allowing students who are disconnected from the physical teaching environment to share and comment code, embedded into an online course management.

At the time of preparing this paper, our implementation is in a preliminary stage. Some details of our current system will be provided as-is, yet these details are likely to change during the course of the implementation.

BACKGROUND: VISUAL PROGRAMMING LANGUAGES

Visual programming languages (VPLs) have been used as learning tools for a long time.

For example, both Alice (Conway 1997) and StarLogo TNG (Klopfer, Scheintaub et al. 2008) enable students to program using a drag-and-drop metaphor, in which programming blocks are arranged on a virtual paper sheet. One of the main advantages of this approach is the avoidance of syntax errors, since dragged programming language elements are only allowed to be dropped if they fit the expected type (e.g. cannot drop a string onto a function requiring a number value as input).

Instead of the block-based metaphor, the VPLs used in form generation (e.g. McNeil Grasshopper, Generative Components by Bentley) follow a flow-based approach, in which multiple components are linked together to form a transformation chain, from input data (e.g. sliders) to generated geometry. Each component can take arbitrary numbers of inputs and outputs (as opposed to only one output in classical programming). As further differentiation, the latter approaches are interactive, with form generation happening in parallel to code development.

The difference between these two programming styles is subtle, but important: visual programming languages (VPL) are flow-based, whereas textual languages are block-based. In the first case, data flows along the graph network and is transformed by subsequent nodes. In the second case, data exists in the form of variables, each defined in a block of code. Blocks are essentially containers for data and commands, structured hierarchically to form a tree. A variable can be accessed and modified by a command if it is found in a same branch (this is also called *data visibility*). Therefore, we can conclude that in flow-based languages, data travels along a transformation chain, while in block-based programs a transformation of data is performed by executing commands in a hierarchical fashion, acting on variables that stay in the block where they were defined. We will give more details on this point in the next sections, since it has implications on the overall design of the Lawnmower VPL.

BLOCK-BASED VERSUS FLOW-BASED PROGRAMMING LANGUAGES

Flow-based languages use tokens to carry data

In flow-based programming languages (FBPL), data flows through a network of nodes and edges. Regardless of dialect used (e.g. business process diagram, Grasshopper), the underlying technique is using a so-called token to represent program execution, which is a pointer to the node being currently executed: Once a program is started, the token is set onto a node, which executes contained command and passes the token on to next node that is connected.

In case that there are multiple outgoing edges, the token is either be duplicated (i.e. every next node gets a copy), or it is up to the node itself to determine to what node the token is passed (this is generally called decision, or "dispatch" in Grasshopper). Vice versa, tokens coming from multiple nodes can be merged to form a single token.

The data used by the node for performing calculation is either contained in the token, or it is globally available: in the first case, the token carries a value with it, e.g. "x", which the node takes to compute a new value "y", which is again stored in the token. In the second case, the value is available in the form of special parameter nodes that do not perform computation, but allow the user to enter values. For example, Grasshopper offers nodes with contained sliders or nodes representing a constant value, which can be hooked up to nodes requiring input.

In the preceding description, it has been said that a program starts by setting a token onto a node (more precisely: a processing node performing computation). However, it is yet unclear which of all the nodes should act as such. In business processes, a start node is explicitly defined (i.e. there is exactly one), while Grasshopper presumably uses graph analysis to come to the conclusion which processing nodes have no incoming edges, thus being start nodes. The question of multiple versus single start nodes will be important when trying to transition from visual programming to textual programming, which is due to be presented in the next section.

In block-based languages, data does not travel

- In the case of block-based programming languages (BBPL), there is exactly one entry point for a program – in which the necessary parameters (if any) are to be supplied. For example, sin(x) starts e.g. when invoking sin(1), which passes 1 as parameter x. Programs are structured hierarchically, as sets of nested blocks: A block is a list of statements (for simplicity, one can think each statement being a line of code).
- The whole program forms the topmost block, in which statements such as if and while open their own (nested) block. Thus, a (tree-)hierarchy of blocks within blocks is established.
- From a data standpoint, nested blocks can access values defined either locally (in the block itself) or blocks further up the hierarchy, which is called visibility.

The latter fact is of special significance when comparing BBPL and FBPL languages: BBPL have a pointer to the current statement being executed, however (and in contrast to FBPL), this does not carry any data. Statements can access a value if it is visible to the current block.

Summary of differences

Transforming visual into textual programs is hard, because the FBPLs are occupied with routing tokens containing data along a graph, while BBPLs executes lines of codes in nested blocks defining data visibility. Our attempt will show how such an approach can nevertheless be done, when transferring some of the concepts found in BBPLs into a FBPLs.

LANGUAGE DESIGN

The Lawnmower VPL has been elaborated through a user-driven development process, using questionnaires and mock-ups. In this section, we first report the findings from our user survey, before describing the design options that make up the language.

Survey

In January 2012, we have launched a questionnaire to collect information on designers' perception of Grasshopper's interface, functionalities and their ideas on its future development (see Figure 1). It included specifically targeted questions on the representation and use of "dispatch nodes" as well as loops and "f(x) nodes" (i.e. expressions). 54 users have responded to the questionnaire, among which 67 % identified themselves as students, 15% as teachers, 20% as architects and 11% as 'other'.

According to the results of our analysis, 37% of the participants have expressed that they often, frequently or always face problems while using the interface, while 38% percent claimed that they sometimes experience problems. The distribution of this data according to user types can be found in Figure 1a. The major reason of problems according to the users were: lack of their programming knowledge (62%), complexity of the language (14%), lack of debugging option (21%), interface (17%), incompatibility with other scripting environments (12%) and other reasons such as incompatibility of versions, lack of programming concepts on component scale and not fully explained data types (15%) (see Figure 1b).

In our view, the lack of programming knowledge and wish for a debugging option indicates that it is not always obvious how the data is passed around, i.e. to foresee the outcomes of the graph network. In this context, several issues might exist in the semantics of nodes, which are used in several meanings:

- As *processing nodes*, which receive input and produce output.
- As *dispatch nodes*, which distribute flow along its two outputs, for cases in which a set condition is true or false.

The dispatch nodes require some more attention, as 80% of the users who often and always experience problems in Grasshopper rated their representation as being "difficult". The same majority (80%) also expressed their willingness to use a "repeat" component (i.e. a loop) and think that the expression designer for F(x) nodes can be improved. In our view, the latter aspect seems to be rooted in the breach of metaphor experienced when having to enter formulas for F(x) nodes as text (i.e. mathematical notation): Formulas could well be entered graphically, by using a flow network of mathematical components (e.g. sin(),+, -) in the same way as the rest of Grasshopper. A further point not connected with the graphical representation is that of lacking comprehensibility of data flow (and therefore the call for step-by-step debugging). Data in Grasshopper is mostly exchanged via lists containing geometrical objects. These lists are used as a replacement for loops, in the following manner: Instead of building up an object iteratively by consecutively adding a point, all points are immediately generated and modified as they pass through the flow graph. Understanding the modifications on lists of objects can be hard to understand, and requires a certain amount of knowledge of set theory (e.g. when intersecting two lists). If there would be an explicit loop statement, simple data types could be used to iteratively build up an object on a point-by-point, which we find preferable for didactic purposes.

Defining the component layout

Based upon the conducted survey, an initial design of the Lawnmower VPL has been created, which is current under intensive mock-up testing and preliminary implementation. The design options given in the following subsections acted as a starting point and conceptual basis on which development is based.

Like Grasshopper, Lawnmower uses nodes represented as rectangle for its flow graph. Inputs and outputs are depicted as circles (*ports*). There are two basic types of components:

- *Data components* hold a value of a specified data type (refer to Figure 2a). If the input port is set, the data component displays supplied value. In all other cases, the data component allows users to enter a value using sliders, spinners and text boxes as widgets.
- *Code components* perform calculation and act

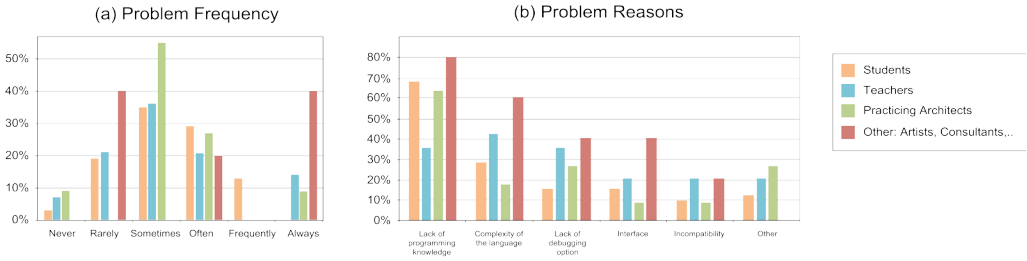


Figure 1
(a) Frequency of problems according to the user types (b) Problem reasons according to the user types (Each respondent could choose multiple responses).

recursively as containers for other code components (see Subsection "Combining Block-based and Flow-based"). There are two fundamentally different view modes for code components: *definition mode* (see Figure 2b) shows the component with the contained commands, while *call mode* (see Figure 2c) hides these internals. In all cases, one or more inputs and one possible output (constraint of textual languages) are given as nested data components (visible in Figure 2c).

Combining block-based and flow-based

As mentioned, code components form a hierarchy: The program itself is the topmost component, in which the following subcomponents may be used:

- *Code block* – an area in which components and connecting edges are drawn (Grasshopper calls this the "canvas"). Each such code block is transformed into a line in the textual language. This essentially means that Lawnmower enforces a "one expression per line" policy, and there will likely be a multitude of code blocks in sequence (also see "Code Generation"). The addition of such code blocks might be done explicitly (i.e. by the user) or implicitly (each component creates a code block for itself).
- *Conditional block* (see Figure 3a) – resembles an "if-then-else" construct found in textual languages and consists of (1.) a condition block "if", which has an embedded data component of type Boolean, used to select either the following (2.) "then" or (3.) "else" block for execution.
- *Loop block* (see Figure 3b) – a condition block that executes a following regular block ("body") while the condition evaluates to true (pre-test loop).

Dealing with space constraints

Previously, lack of screen space that is common to VPLs has been addressed by using Zoomable User Interfaces (ZUIs). Lawnmower extends this concept by also incorporating earlier work on techniques known as Fisheye/Focus+Context visualizations of source code (Furnas 1986):

- Code components are collapsible/expandable (Figure 3c). There is always one block that is *in focus* and therefore expanded. All other components stay collapsed (*context*).
- Data components visible to the block in focus are shown as possible inputs.

All in all, these functionalities aim at enabling concentrated work on the current piece of code while helping to conserve space.

Code generation

The Lawnmower concept of code entry is visual, but nevertheless line-based: Each code component stands for a single line within the textual code. The lawnmower editor must therefore check syntactic validity of the entered graph, in order to guarantee that the generated source will be valid and fits on a line. Figure 4 gives a general outline of the used checking rules:

- **Connectedness rule**. The entered graph must be connected, i.e. it must be "one graph" in which all nodes are reachable by an edge path.

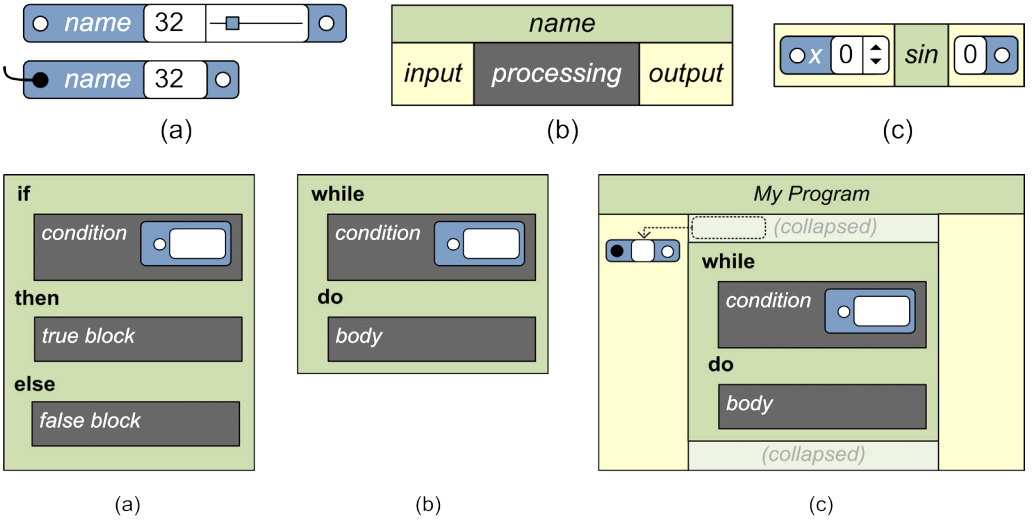


Figure 2
Language Design. (a) Data components (b) Code components in definition and (c) call mode.

Figure 3
Code Component Types. (a) Conditional block (b) Loop block. (c)Using collapsible code blocks to conserve space.

- **No cycle rule.** Figure 4a shows the statement "i = i + 1" as lawnmower graph. However, it is unclear from the depiction how the update to i should happen. It could either mean that "i is *to be created* having the value of itself plus one" (a paradox, since it does not exist at that time), or it could mean that "i *should take* the value of itself plus one" (effectively deactivating the initialization value, and thus becoming undefined). Disallowing cycles is therefore a necessary, but not sufficient measure to establish some order.
- **Single setter rule**. We explicitly allow *at most one data update* per data component (refer to Figure 4b): i is created and immediately set to 0. Consequently, there can be no more statements inside the same code block. In the next code block, i is referenced two times: one time in the form of a "get" (as input to "+"), and one time in the form of a "set". The code editor must therefore explicitly distinguish declaration/initialization and reference to a variable. This is exactly in line with what was introduced in Figure 3c.

Figure 4c provides an additional example of a valid lawnmower code component, using two gets and one set. The code generation regards the set as the left side of an assignment ("len"), the rest as right side ("sqrt(x*x + y*y)" through traversal of the graph). Graphs that contain no setters are regarded as non-assignments, generating as a function call (e.g. "redraw()"). Lastly (and: trivially), the empty graph is valid as well - it is translated into an empty line.

The actual generation of textual code in the target language happens through exchangeable adapters. Currently, we take VBA for AutoCAD as our language target. However, additional language targets may be added at will, enabling students to "script once and run many" environments, which is in line with recent tendencies in the field (Leitão and Santos 2011).

Figure 4
Syntactic Check. (a) Cycles cause disruption of temporal order and overwrite initialization values. By using (b) the single setter rule and distinguishing between declaration and later reference to a variable, (c) the mapping between a lawnmower code block and a line of textual code becomes possible.

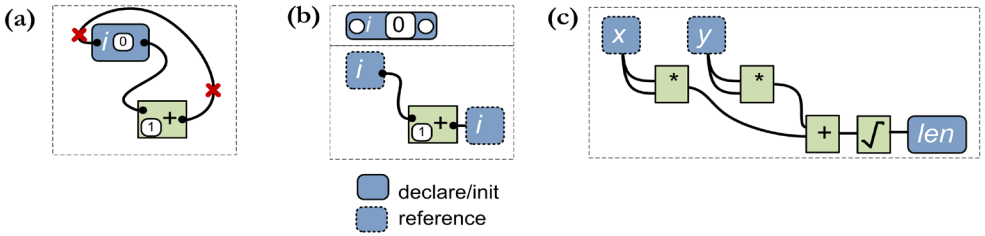
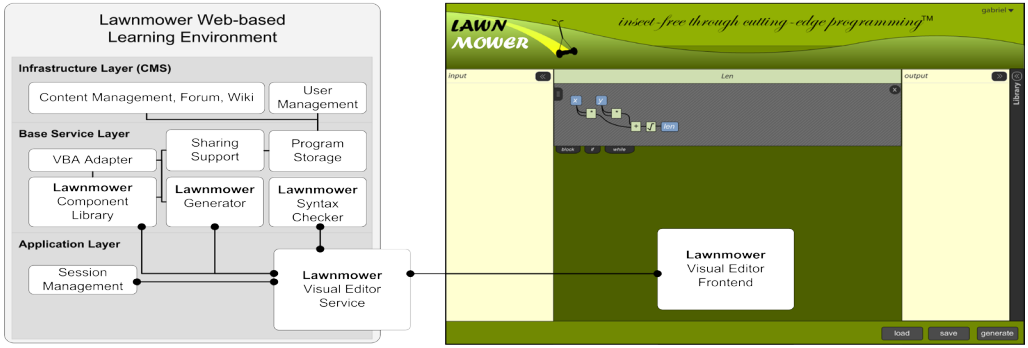


Figure 5
Lawnmower integrated with a content management system to support learning activities, code management, storage and distribution and evaluation.



CAPABILITIES OF LAWNMOWER AS A SOCIAL LEARNING ENVIRONMENT

As referenced in the first section, besides providing various novel functionalities, Lawnmower also aims at enabling learning as a social process (Brown and Adler 2008). It promotes "community building" and "social learning" and intends to facilitate reflective learning-in-action (Schön 1987) in a novel pedagogical context, in which various communication modes and styles are supported. It is a web 2.0-based environment tightly linked with a content management system to support learning activities, code management, storage and distribution and evaluation (Figure 5). By this way, we aim to encourage integration with design studios, design computing courses and short-term workshops.

In line with the aim stated above, Lawnmower allows the teachers to initiate a learning topic linked with a learning theme. The students can enroll into this theme and use Lawnmower editor to accomplish the tasks provided by the teacher(s). It acts as an evaluation medium by the use of a dedicated module and the timeline which keeps a track of the individual and collaborative development processes. This evaluation is not limited with grading but also planned to include a critical evaluation from the students' side such as the general mood, the course material and perceived quality of teachers' performance (Wurzer, Alacam and Lorenz 2011).

Configuring Lawnmower with the functionalities above allows various kinds of social practices to take place and creates various opportunities for learning programming:

- First of all, the system can potentially facilitate the development of code through asynchronous collaboration and thus motivate learning from other students and open resources. Through the use the commenting module, Lawnmower can facilitate collective construction of knowledge through dialogue; as well as learning the activity of learning itself.

- Moreover, while the students are disconnected from the physical teaching environment, they can still learn from and comment on each others' programs and create a collective understanding of the problem(s).
- The online course management module can allow the structured documentation of course materials and various products that are created during the courses. These can be transferred to concurrent or other courses, architecture students and teachers in various geographies. Such a practice can lead the development of a repository of learning tasks and self-contained interactive resources focusing on one area (learning packages) adapted to architecture students' learning needs.
- In the future, Lawnmower may appeal to advanced programmers and include complex applications, thus involve expert knowledge into educational activities.

We are well aware that, besides the content management functionalities, the proposed web environment needs to be supported by various motivational tools. Among the planned ones are the creation of open challenges and development of reward mechanisms such as rating and points; and the timeline-based animation of user created content for better communication. Furthermore, in order to test the field applicability of Lawnmower, we are planning to conduct usability tests based on designer satisfaction, effectiveness, efficiency, freedom from errors, learnability, use sustainability and sociability measures, as introduced in Pak and Verbeke (2011). At all stages, we are also planning to exploit all advantages of web 2.0 environments for crowdsourcing usability.

IMPLEMENTATION

Because of Lawnmower is still being actively developed and tested from a design viewpoint, our preliminary implementation focuses on technological feasibility, foremost: The ability to transfer the programming constructs of the VPL to textual code, depicting components using web technologies, downloading textual programs into a target platform and technologies for sharing programs among students.

- The translation of visual code into textual code happens on the backend of the server, using a stateless service that performs graph analysis on the flow graph. The translated textual code (currently: VBA for AutoCAD) is transferred manually into the target platform using the system clipboard as exchange method. A future implementation would offer more language targets (as e.g. Leitão and Santos 2011) and would also automate transfer of textual code using a plug-in.
- For each user, the state of the Lawnmower editor is held in a session. Clients are connecting with their browser to a webpage, which utilizes AJAX to refresh itself, based on user input and server actions. Rendering of the VPL in the browser happens via use of SVG/VML/Canvas controlled by JavaScript. Storing and sharing of code components as well as collaboration among the users is performed using a message-oriented database.

CONCLUSIONS, APPLICABILITY AND LIMITATIONS

We have presented a cutting edge web-environment called "Lawnmower", which can help students in getting acquainted with textual programming by generating code. Our efforts are based on a conceptual extension of Grasshopper, bringing block-based and flow-based programming techniques together. Our development, which is now undergoing mock-up testing and preliminary implementation, will ultimately produce a social exchange platform in which code can be shared by students, for the sake of education and collaborative learning. Our work is intended for use in classrooms, mainly aimed at form generation. However, we do not strive to create a "production-ready" scripting environment (emphasis is on social learning), nor do we want reverse-engineer textual code into graphs (which is hard, to say the least).

REFERENCES

Brown, J and Adler, RP 2008, 'Minds on Fire: Open Education, the Long Tail, and Learning 2.0', *Educause Review, (January/February 2008)*, pp. 16-32.

Cabecinhas, F 2010, *A High-Level Pedagogical 3D Modeling Language and Framework*, PhD Dissertation, Technical University of Lisbon.

Celani, G and Vaz, EVV 2012: 'CAD Scripting And Visual Programming Languages For Implementing Computational Design Concepts', *International Journal of Architectural Computing,* 10 (1), pp. 121-138.

Conway, M 1997, *Alice: Easy-to-Learn 3D Scripting for Novices*, PhD Dissertation, Carnegie Mellon University.

Furnas, GW 1986, 'Generalized Fisheye Views', *Proceedings of CHI '86*, pp. 16-23.

Klopfer, E, Scheintaub, H, Huang, W and Wendel, D 2009, 'StarLogo TNG: Making Agent-Based Modeling Accessible and Appealing to Novices', *Artificial Life Models in Software*, pp. 151-182.

Leitão, A and Santos, L 2011, 'Programming Languages for Generative design: Visual or Textual?', *Proceedings of the 29th eCAADe*, pp. 549-557.

Pak, B and Verbeke, J 2011, 'Usability as a Key Quality Characteristic for Developing Context-friendly CAAD Tools and Environments', *Proceedings of the 29th eCAADe*, pp. 269 - 278.

Pak, B and Verbeke J 2012 (forthcoming), 'Design Studio 2.0: Augmenting Reflective Architectural Design Learning Using Social Software and Information Aggregation Services', *Journal of Information Technology in Construction (ITCon),* 17.

Schön, D 1987, *Educating the reflective practitioner: Toward a new design for teaching and learning in the professions*. San Francisco: Jossey-Bass (Kindle Version).

Wurzer, G, Alacam, S and Lorenz, WE 2011, 'How to teach Architects (Computer-) Programming', *Proceedings of the 29th eCAADe*, pp. 51-56. Overview