Collaborative Digital Design

When the architect meets the software engineer

Luís Santos[1], José Lopes[2], António Leitão[3]
[1]IHSIS (Institute for Humane Studies and Intelligent Sciences), Portugal, [2,3]Instituto Superior Técnico, Technical University of Lisbon, Portugal, [3]INESC-ID, Portugal
[2]http://fenix.ist.utl.pt/homepage/ist158612, [3]http://fenix.ist.utl.pt/homepage/ist13451
[1]luis.sds82@gmail.com, [2]jose.lopes@ist.utl.pt, [3]antonio.menezes.leitao@ist.utl.pt

Abstract. *Increasingly more architects use programming as a means for form finding and design exploration, a tendency that is expected to continue. Even though significant progress has been made in the simplification of programming languages, complex design tasks might still require large coding efforts. We do not think it is wise to force architects to also become experts in programming languages and software engineering. Instead, similarly to what happened with other design and building disciplines, we think that the future of digital design lies in the collaborative effort of architects and software engineers. In this paper we analyze different situations where such collaboration increases productivity and frees the architect to more creative tasks.*
Keywords. *Architecture; Software Engineering; Design; Collaborative process; CAD tools.*

INTRODUCTION

Architecture is a creative discipline that depends on the specificities of the program, site, social and economical circumstances, historical and political background, and the concerns of each architect or design team. Therefore, an architectural project is always unique in its design requirements. However, in some cases, commercial CAD packages are not enough to tackle effectively and efficiently the specificities of each design. As a result, increasingly more designers and architectural practices use programming to conceive specialized digital tools tailored to the given design tasks. The use of programming in design allows (1) design task automation, (2) extension of CAD application features, (3) customization and procedural generation of parametric models, (4) algorithmic exploration of different design options, (5) 3D printing optimization, (6) implementation of digital fabrication protocols, (7) deal with complex models, and (8) pursue exhaustive design exploration through the manipulation of scripted parametric models. Unfortunately, programming is still limited to a small number of architects and design teams.

It has been suggested that architecture students should learn programming (Leitão et al., 2010; Burry, 1997) and, in fact, several architecture curricula already follow this recommendation. Learning programming requires learning a programming language, and it is only reasonable that architecture curricula adopt languages that can be directly applicable in the production of digital design. As a result, most curricula teach languages that are provided by CAD applications, for example, AutoLisp, RhinoScript, and Grasshopper.

Even though these languages allow a smooth transition from theory to practice, they have two major problems, namely, (1) most of these languages lack fundamental pedagogical qualities, and (2) professionals become locked-in to specific CAD languages and tools not only because it is difficult to reuse programs written in different languages and to migrate to different CAD tools (Lopes, 2011), but also because professionals find it difficult to learn and adapt to new systems.

In order to avoid these problems, one would think that students should learn several programming languages. This strategy is actually followed in several software engineering curricula, with courses requiring different programming languages. Unfortunately, this approach is impractical for architecture students due to the extremely small number of programming courses available in typical architecture curricula. On the other hand, increasing the number of programming courses is not a viable solution because it is not wise to try to convert architects into programming experts (Boeykens and Neuckermans, 2009).

In practice, although some architects may specialize in programming, the majority will only master programming up to a certain level. As a result, with increasingly complex design tasks, there is a moment when an architect feels the programming challenge he faces is so large that the necessary effort to accomplish it will not pay off. At that moment, it is more productive to invite software engineers to participate in the process.

A software engineer can contribute to the design process in a number of ways, including, custom modeling and analysis tools, management and optimization of scripts made by architects or other professionals, mediation or translation of programs written in a variety of languages, and support to the development of complex programs and algorithms when necessary. As a result, while software engineers are concerned with the programming side of design tasks, architects can focus on the architectural side of those tasks.

In general, the benefit of multidisciplinary teams comprising both architects and software engineers can be translated in the ability of the design team to (1) exhaustively explore the design solution space; (2) make informed decisions from the feedback provided by performance analysis; (3) study, design, and optimize, the fabrication of complex building components; and (4) quickly and economically incorporate changes and variations into complex designs.

Even though the concept of multidisciplinary teams is not new, the fact is that this deeply integrated, collaborative effort between architects, designers, and software engineers, is still exceptional. In practice, only large design offices are currently investing in the formation of these multidisciplinary teams. However, this does not mean that smaller design offices cannot take advantage of this collaboration: outsourcing their specialized programming needs can be a valid solution.

In the following sections, we present examples of well-known cases of multidisciplinary teams and we show practical cases where this collaboration is beneficial for the design process.

RELATED WORK

The eighteenth century was a turning point where architecture and engineering became separate fields. Since then, architects have always worked collaboratively with other experts, therefore, it is not surprising that the effective use of programming in the design process also requires the collaboration of software engineers.

Some well-established architectural and engineering practices present interesting case-studies of these recent collaborative design teams, namely, Specialist Modelling Group (SMG) of Foster and Partners (Peters and DeKestellier, 2006), Advanced Geometry Unit (AGU) of Arup (Walker, 2004), Computational Design & Research (CDR) and the Advanced Modelling Group (AMG) of Aedas, and BlackBox Studio of Skidmore, Owings & Merrill (SOM). These are examples of multidisciplinary teams comprising architects, mathematicians, engineers, and programmers.

For example, SMG is a specialized group with the purpose of developing project-driven research and design workflow consulting (Whitehead, 2011). SMG lead architect, Hugh Whitehead, says the work at SMG can be divided in two major areas, namely, (1) development of procedures for data transfer and exchange, and (2) development of data analysis mechanisms to support decision making in the design process.

Team members at SMG are called "architects plus". They are professionals originally trained as architects that specialize in digital techniques, such as, 3D modeling, scripting, and digital analysis tools. For SMG, this kind of organization provides a better support to design teams because team members have a designer mindset and, as result, they have a clear understanding of the design brief and its actual requirements. This approach aims at improving the workflow of the design process, not only by conceiving tailored tools for specific problems on demand, but also by giving advice to designers so they can devise better strategies and choose adequate parameters and procedures.

The specificity of each design and the uniqueness of every design team result in avoiding formula solutions. As a result, the collaborative process focuses more on the applicability of digital technology to inform design workflow, to select the correct type of information which, then, becomes a stimulus to design (Whitehead, 2011). Design teams make use of the tools provided by SMG and provide feedback, which is used to improve those tools. As a result, design teams are more motivated to participate in this process, where not only they learn how to use those tools but they also become actively involved in their development.

Moreover, the fact that younger generations of architects are more aware of programming is beneficial to SMG because they require less training and can pursue more individual work, thus relieving the group which, in the end, has more time for research (Whitehead, 2011). As a result, the role of SMG becomes one of mediating different design teams and external consultants, reinforcing the idea of a multidisciplinary environment where different expertise, experiences, and points of view, are focused on finding the answer to multi-criteria problems.

This mediation and interpretative role are shifting the workflow concerns of SMG to code management, sharing, and reuse. In this case, a software engineer is particularly important because, by being an expert in programming, he is capable of improving code in order to make it more modular, abstract, and flexible, so it can be reused and adapted to different design tasks.

AGU of Arup is another example of a multidisciplinary team. Arup is a large and well-established engineering office and AGU is a research and consulting group that provides internal and external consulting. The group, lead by architect and engineer Charles Walker, has a set of skills that comprehend engineering, architecture, mathematics, physics, and programming. The main goal of AGU is to find new or less conventional solutions by exploring different strategies, such as, algorithms, generative systems (e.g. fractals), and non-linear structures (Walker, 2004), and to systematically apply these strategies to design geometrically complex forms and structures. Similarly to SMG, AGU faces each design as a unique problem, thus providing project dependent consulting. As a result, AGU comprehends a small number of permanent team members and according to the requirements, size, and complexity, of each project, a larger ad-hoc team is assembled for the duration of one or more stages of design for that project.

Even though current software is enough for most problems, there are some tasks that require custom software (Walker, 2004). In this case, AGU will try to extend (whenever possible) existing CAD and structural analysis software with additional components that target specifically those tasks. If this is not possible, then AGU will develop new tools. This shows that AGU focuses not on developing new technology but instead on reusing software as much as possible, with the objective of accelerating the design process. The unusual composition of team members at AGU and the application of inno-

vative digital tools in the design process suggest a new organic structure, which independent architectural practices can adopt to remain competitive in the race to tackle complex and large projects.

SMG and AGU are successful examples of multidisciplinary teams comprising architectural and software engineering skills, showing that this collaborative effort is a promising organizational decision from which we can expect better, more effective, and more efficient design processes. It is our understanding that, due to the complexity of custom digital tools, algorithms, parametric models, and the growing need of code reuse and optimization in digital supported design, the role of software engineer within design teams and the communication between designers and software engineers are becoming critical.

COLLABORATIVE DIGITAL DESIGN

Collaboration between architects and software engineers must be supported by a common vocabulary. Architects without programming notions cannot effectively describe their design ideas to a software engineer. Conversely, software engineers without architecture notions cannot understand architectural requirements. More specifically, we believe that software engineers that want to collaborate with architects should learn geometry, representation techniques, and 3D modeling.

In spite of the initial knowledge each must have about the work of the other, there is a natural tendency for this knowledge to grow: the collaborative experience becomes also a learning experience. With time, the software engineer specializes in architectural problems and the architect becomes more aware of the capabilities and limitations of programming.

We will now describe a range of situations that we have been experiencing in the last few years, where the complexity of the programming task required collaboration between design teams and programming experts.

Situation 1: Automation

There are tasks that are repetitive by nature. For example, consider the creation of a 3D model for an urban fabric based on building boundaries and elevation points (Figures 1 and 2). Most architects that accomplished this task consider it annoying and time-consuming because, for each building, the architect has to visually locate the elevation point contained in its boundary and then extrude the boundary up to this elevation. This sequence of operations must be repeated a large number of times, being an excellent candidate for automation. Automating an extrusion operation is an easy task that most architects with a modicum of training in scripting languages for CAD applications quickly accomplish. However, instructing a computer to visually locate points inside polygons might not be straightforward for an architect, although it should be easy for a software engineer acquainted with computer graphics algorithms. The algorithm relies on testing the containment relation between a polygon and point, which can be implemented by a predicate, i.e., a function that returns true if the point is contained in the polygon and false otherwise. To this end, we can use the algorithm presented in Figure 3. Then, for each polygon, the set of points must be filtered according to the containment predicate and three situations can occur: (1) the resulting set of points is empty, meaning that the polygon does not contain any points; (2) the resulting set contains a single point, meaning that the point describes the elevation of the polygon; and (3) the set contains more than one point, meaning that there are different elevation possibilities.

At this moment, the software engineer consults with the architect to understand what should be done in each case. The answer might be that he should not be concerned with cases (1) and (3) because the architect guarantees that they do not occur in practice, or it might be that the program should report an error for these cases, or report an error for the first one and use some kind of elevation average for the third; or any other possibility. In any case, given that it is the architect that decides

Figure 1
Plan of urban fabric with elevation points.

Figure 2
The 3D model of the urban fabric.

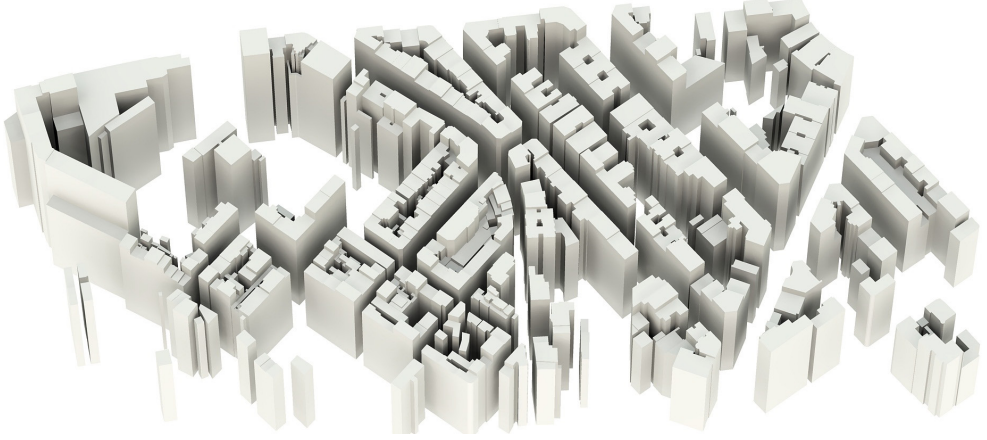
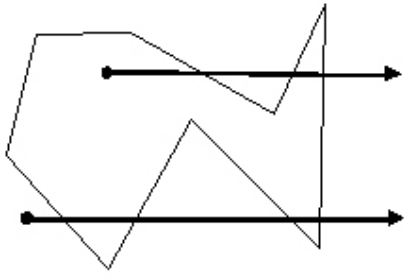


Figure 3
Testing if a point is inside a polygon: compute a ray that starts on the point and count the number of intersections with the polygon edges. If the number is odd, the point is inside the polygon, otherwise it is outside the polygon.



the course of action, either the software engineer implements the architect's choice or, even better, he implements all options and a way for each architect to provide their own choice.

This is an example of a real situation where the final result was an AutoLisp script written by a software engineer for an architect that then was used by many other architects, saving them enormous amounts of time.

Situation 2: Translation of algorithms

Many programming tasks can be solved by reusing existing algorithms. Unfortunately, in many cases, these algorithms are written in programming languages which are not commonly used by designers, such as C or Fortran, and significant effort might be required to learn such languages just to translate those algorithms. On the other hand, translation of algorithms between languages is an ordinary task for a software engineer.

As a real example, consider the creation of meshes from sets of points, an activity that is frequent in 3D modeling but not directly supported by all CAD tools. An architect that needs to produce such mesh might search the literature or the internet for an adequate algorithm, and he will probably find the Delaunay algorithm. However, it is very unlikely that he finds the algorithm written in a language that is ready to be executed in the CAD tool he is using. Most probably, the algorithm will be written in some of the most used programming languages, such as C, Java, or C++. However, these are not the languages that most architects learn. Instead, they learn CAD scripting languages, such as AutoLisp or RhinoScript, and without previous experience, it can be difficult to understand other languages, which have very different syntax, semantics, and pragmatics. Moreover, given the effort needed to learn just a single language, it is not surprising that programmers tend to become addicted to the first language they learn and do not think it is worthwhile to learn a new one just to be able to translate some algorithm.

This is another situation where collaboration with a software engineer can be a serious time saver. Software engineers should know several programming languages and, more importantly, they should be able to learn new ones when necessary. This means that the necessary effort for a software engineer to learn an algorithm written in a "foreign" language and rewrite it in the language required by the CAD tool used by the architect should be considerably smaller than the equivalent effort required for the majority architects for the same task. The same can be said for the translation of scripts from one CAD tool to another.

```
#include <iostream>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/convex_hull_2.h>
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;
int main()
{
Point_2 points[5] = { Point_2(1,0), Point_2(7,2),
Point_2(9,1), Point_2(6,5),
Point_2(4,1), Point_2(3,9) };
Point_2 result[5];
Point_2 *ptr = CGAL::convex_hull_2(points, points+5, result);
std::cout << ptr - result << " Convex hull" << std::endl;
return 0;
}
```

Figure 4
C++ program for computing the convex hull of a set of points.

Situation 3: Using libraries

Certain design tasks require an entire software library. For example, the Computational Geometry Algorithms Library (CGAL) is a free, open-source library that provides a large number of algorithms and data structures for solving complex geometric problems, such as computing convex hulls or Voronoi diagrams. However, the library is written in C++ and makes extensive use of its complex template system, being almost hermetic to non-software engineers. Figure 4 presents a small C++ program using CGAL for computing the convex hull of a set of 2D points. Although this task has a lot in common with situation 2 and it would be possible to translate the convex hull algorithm written in C++ to the AutoLisp language, the effort needed to do that might be considerable because CGAL is a huge library, with many interdependencies that make the translation effort highly non-cost effective. So, in this situation, we recommend a different approach that, once again, is doable by a software engineer but might represent a tremendous effort for an architect: create bindings between the library and the architect preferred language. These bindings allow architects to use the data types and program constructs of their preferred language to interact with the library, effectively hiding it and, as Figure 5 illustrates, making it much easier to deal with it.

There are several technological approaches for implementing these bindings, including foreign function interfaces, sockets, COM, and others, depending on the support provided by the languages and libraries, and we have explored some of these approaches in the past, e.g., to connect CGAL to the Racket language and to allow AutoLisp to be used as scripting language for Rhino 3D. As an interesting side-effect of this effort, we have been noticing that after the initial development work, some architects start to look at the actual binding implementation and some of them even make improvements to the bindings, extending them and/or correcting bugs.

Figure 5
AutoLisp program for computing the convex hull of a set of points.

```
(print (convex-hull '((1 0) (7 2) (9 1) (6 5) (4 1) (3 9))))
```

Situation 4: Interaction between applications

Analysis and performance simulation tools are becoming increasingly important to provide data into to the design process as means to support decisions and evaluate solutions. Therefore, an extremely important case of collaboration between software engineers and design teams occurs when interaction between a CAD system and other applications, such as, building energy simulators or structural analysis tools, is necessary. For example, associating performance data to parametric models is necessary to assess the impact of variable and parameter modifications in the overall performance. And associating different systems requires software engineers to implement specialized connection components, and the integrated design data helps designers in decision making.

The interaction between applications can also be useful in goal-oriented design where a generative system iteratively produces, analyses, and compares, different solutions to find the one that maximizes the objective set for a specific performance behavior. In most cases, goal-oriented systems are composed of (1) a modeling application that produces geometry, (2) simulation software that runs performance based analysis, and (3) an optimization algorithm that guides the search process. Because these components must communicate between them, software engineers have to conceive and implement communication protocols, sometimes reusing existing technologies, such as, foreign function interfaces, ActiveX, and remote procedure calls. These techniques are commonly taught in the software engineering curricula and they are in the scope of software engineering tasks, but not in the usual concerns and skills of an architect.

A fine example that illustrates this situation is the collaboration between Kristina Shea, a mechanical engineer, and Robert Aish, a software engineer

(Shea, 2005). Together, they connected Bentley Microstation and GenerativeComponents (GC) with eifForm, a performance-based generative design system for structural design (Sass, 2005). With this integration, eifForm users can take advantage of the richer modeling and visualization mechanisms of Microstation, including the programming capabilities of GenerativeComponents. On the other hand, Microstation users can take advantage of performance or optimization analysis, for example, to perform structural mass reduction on their Microstation models. The interaction between these applications was accomplished via XML. With this technology, one of the applications serializes the geometric models to XML format and the other application executes the converse process. While architects are better prepared to make architectural and design decisions, software engineers are well prepared to choose an adequate technology for a given programming problem or set of software requirements. For example, the choice of XML technology suggests that one of the most important software requirements is easy data sharing across different applications.

Situation 5: Development of large scale programs

In general, complex problems can only be solved by complex programs. One of the fundamental goals of software engineering is precisely to allow the development of these programs in the most economical way. To this end, many techniques have been proposed and successfully used, for example, object-oriented programming, design patterns, and model-driven development. Due to time restrictions, these techniques are hardly addressed in the architecture curricula. As a result, architects spend unnecessary efforts developing programs that could be much more effectively developed by software engineers.

As an example, consider that a designer needs a programming environment in which he can write Generative Design programs that create geometric models in the most popular CAD applications. Note that this designer does not want to be limited to a specific CAD application. Instead, he wants to be able to interoperate with multiple CAD applications because, for example, his work colleagues deal with multiple tools. Conceiving such programming environment entails designing adequate software architecture, generalizing the functionality of CAD applications, abstracting portable functionality and emulating non-portable features, establishing connections between different systems, serializing data, and so on. These tasks are suitable for a software engineer, but quite unusual for an architect.

At this moment, architects are not capable of overcoming these problems by themselves because they do not possess the necessary knowledge to distinguish between problems that result from legacy systems, language implementation details, and inadequate programming approaches, from those that are intrinsically complex from the computational point of view. Therefore, architects usually approach all problems in the same way: they write programs that implement a given task and at the same time circumvent the limitations of CAD applications, making them unnecessarily complicated. A software engineer can propose solutions to overcome those secondary problems, thus allowing designers to focus on their design task.

For example, a software engineer can select an appropriate programming language and develop a geometric algebra embedded in that language such that the problem of arguments being consumed by operations no longer applies. This requires understanding memory management and language evaluation models. He can also advise multiple dispatch, a feature of object-oriented programming, as a mechanism to implement generic operations, such that operations, such as, union, intersection, and subtraction, work for all geometric shapes. Finally, the software engineer can also conceive procedures for detecting special cases, such as, empty and universal regions, that typically fail in CAD applications, and implement algebraic equivalence rules to find valid combinations for operations.

Naturally, this process results from the collaborative work of designers and software engineers. For

example, Rosetta (Lopes, 2011) is an example of a system where problems, such as those presented in this section, have been overcome. However, we are constantly observing how architects work and collecting feedback, so that we can identify more problems and implement solutions that aim at simplifying the programming effort.

CONCLUSIONS

In this paper we discuss the utility of collaborative work between architects and software engineers. Despite the benefits of this collaboration, it is desirable that architects continue to learn programming not only as an improvement of their own skills and in the management of digital tools, but also as a means to formulate and model problems that they want to tackle in their designs. Moreover, the increase of programming courses taught in architecture curricula and the popularization of parametric models and the implementation of algorithmic procedures in architectural praxis are relevant factors that cannot be disregarded.

Architects are becoming more proficient in the use of programming as an additional approach to automate, explore, and develop designs. Nevertheless, we believe that it is not necessary for architects to become programming experts. Computer science is an independent field of knowledge with a different culture, concerns, and objectives, than architecture. Architects can master programming techniques up to the level that is sufficient for their immediate needs. However, there are programming tasks that require a great amount of additional programming techniques and the effort needed for the architect to learn and master these techniques might not pay off. For these tasks, a deeper collaboration between architects and software engineers can be highly useful.

As we discussed in this paper, this multidisciplinary, collaborative work is already a reality. Well-known and established Architecture, Engineering and Construction (AEC) firms have specialized multidisciplinary teams where the software engineer or expert programmers put their expertise in the development of digital tools suitable for the design workflow. Designing in such a multidisciplinary environment becomes not only a collaborative effort, where each party shares its knowledge and creativity in a complementary fashion, but also a learning experience for all those engaged in the process.

This paper presented five situations where a collaborative effort between architects and software engineers allowed a more productive design process. These situations included automation of basic design tasks, translation of algorithms, simplified use of libraries, interaction between applications, and development of large scale programs.

We believe this need for collaborative work will increase in the future. Nevertheless, it will only pay off if a strong culture of communication between architects and software engineers is developed. Not only must architects continue to obtain programming skills but also software engineers should learn about architectural culture and praxis in order to understand the architectural design process and the needs and concerns of architects.

ACKNOWLEDGMENTS

This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under the projects PEst-OE/EEI/LA0021/2011 and PTDC/AUR-AQI/103434/2008.

REFERENCES

Boeykens, S and Neuckermans, H 2009 'Visual Programming in Architecture: Should Architects Be Trained As Programmers?', *Cultures et Visions – Proceedings of the 13th International CAAD Futures Conference*, Université de Montréal, Montréal, Canada, pp. 41–42.

Burry, M 1997 'Narrowing the Gap Between CAAD and Computer Programming: A Re-Examination of the Relationship Between Architects as Computer-Based Designers and Software Engineers, Authors of the CAAD Environment', *Proceedings of the 2nd Conference on Computer Aided Architectural Design Research in Asia (CAADRIA)*, Hsinchu, Taiwan, pp. 491–498.

Leitão, A and Cabecinhas, F and Martins, S 2010 'Revisiting the architecture curriculum: The programming perspective', *Future Cities – 28th eCAADe Conference Proceedings*, Switzerland, pp. 81–88.

Lopes, J and Leitão, A 2011 'Portable Generative Design for CAD Applications', *ACADIA 11: Integration through computation – 31st ACADIA Conference Proceedings*, Canada, pp.193–203.

Peters, B and DeKestellier, X 2006 'The Work of Foster and Partners Specialist Modelling Group', *Bridges London Conference Proceedings*, University of London, London, UK, pp. 9–12.

Sass, L and Shea, K and Powell, M 2005 'Design Production: Constructing freeform designs with rapid prototyping', *Digital Design: The Quest for New Paradigms – 23rd eCAADe Conference Proceedings*, Portugal, pp. 261–268.

Shea, K and Aish, R and Gourtovaia, M 2005 'Towards integrated performance-driven generative design tools', *Automation in Construction*, 14(2), pp. 253–264.

Walker, C 2004 'Emergence: Morphogenetic Design Strategies', Castle, H. (eds.), *Architectural Design*, 74(3), pp. 64–71.

Whitehead, H and de Kestelier, X and Gallou, I 2011 '*Distributed Intelligence in Design*', Kocatürk, T and Medjdoub, B. (eds.), John Wiley & Sons, UK, pp. 232–246.