Context Centralised Method for Software Architecture: A Pattern Evolutionary Approach*

Ziyad Alshaikh and Clive Boughton

Department of Computer Science, The Australian National University, Canberra, Australia
{ziyad.alshaikh, clive.boughton}@anu.edu.au

Abstract. *Context* plays an important role in various analysis and design methods in software engineering, and is typically exemplified within data flow diagrams and (sometimes) design patterns. However, we believe that context within software engineering has been largely limited to identifying system boundaries for the scoping of work. In this paper we introduce an approach where the notion of context plays a central role during the analysis, architecture and design phases of software development, providing greater meaning and understanding. Accordingly, we provide a definition of context and how it relates to requirements, architecture and design and then propose a method of requirements elicitation/analysis based on context and its inherent properties for reducing ambiguity, increasing understanding and enabling greater communication. We extend the ideas to include the building of architectures and designs based on context-pattern evolution.

1 Introduction

The term *context* is used implicitly in our daily lives as we converse, travel and observe events. Equally, the concept of *context* is shared among different disciplines and professional practices. In Language, for example, *context* gives more precise meaning to statements/sentences we say/write. In describing historical events, interpretation is made clearer with the political and cultural context of the associated times. Although the term is common we are more aware of *context* in practice than in theory [23]. Following the word *context* to its Latin origins the verb *contexere* means to weave together. Scharfstein [23], a contemporary philosopher, believes that "... no reasoning or action can be understood very well outside of its own context".

In software engineering, *context* has been used, both explicitly and implicitly, at different levels of abstraction within the overall description of systems. In structured analysis, for example, context is used within diagrams to help set system boundaries and to establish the scope(s) of interest/work as well as to help define interactions between the system and its context/environment [13,26]. Hence putting requirements into perspective with regards to context/environment.

We introduce a method based on context which realizes the influence of forces on requirements and designs. Context is defined by Scharfstein [23] as "... that which environs the object of our interest and helps us by its relevance to explain it". We believe, however, that such relevance is applied to the object and to the observer. This means that different observations could result in different contexts. The meaning of colours is

* Appeared in proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFT 2008), Porto, Portugal.

not unified, yet we are able to use them (colours) to emphasise messages; like green to mean go, red to mean stop. Context influence is based mainly on relativity. If we design a chair, the chair's colour would not affect comfort. Colour, however, is relevant to other objects in a house, since colours have to match. Accordingly, we address context in relevance to perception as explicit and implicit, and relevant to force as contextual and non-contextual. If what is perceived of context is correct, this makes the context *explicit*, if not confirmed to be correct it is considered *implicit*. Context also could be enforced strongly, which makes it contextual, if it is weakly enforced or not enforced at all, we regard it to be non-contextual. We consider a context to be contextual as long as its force causes stress on an object [2].

The paper is organized as follows: firstly a review of the existing role of context in software engineering practices for developing precise models of requirements, architecture and design (including patterns and pattern language). Secondly, we propose an approach, a context centric methodology, based on two major activities - the elicitation of contextual requirements and context-pattern evolution. Thirdly, we discuss the advantages and open issues of the method. Finally, we describe some future work to show the use of this new approach in system development in general, and for software architecture in particular.

2 Contextual Requirements

Requirements can be categorized as contextual and non-contextual. However, during requirements capture the two categories are typically not distinguished. Some efforts have nonetheless been directed toward identifying contextual requirements. For example, Potts and Hsi [22] call for a synthesis between two approaches, abstractionism and contextualsim, combining the best features of both. The synthesis is supported by a goal refinement method, which identifies and analyses goals, actors and objects, and transforms these goals into operational requirements, and then identifying and handling obstacles [22]. Another example is the work done by Bübl and Basler [10], where *context* is introduced as constraints linked to requirements. A constraint technique is introduced called context-based constraint (CoCon), which aims to support the evolution of software development by supporting modifiability. Both of the methods realize the importance of *context* in system development. However, both have recognized only one facet. As we've already implied, and will confirm in the following sections, context exists in all stages of software development. However, most methods that are used to develop the various artefacts of a software system fail to recognize its existence.

Context was first identified and explicitly used as part of systems structured analysis approaches, as in the work by DeMarco [13]. In structured analysis the system is abstracted as a set of data inputs and data outputs and a series of processes and flows of data, which are depicted using Data Flow Diagrams (DFD) to help understand the interconnected processes/processing at different levels within a system. It is the purpose of the context diagram to show identified terminators or external entities that interact directly with the system, and to also show identified data that passes between the system and the external entities. In some cases, to understand the context diagram better, an event list is constructed to better describe stimuli and responses of the system, al-

lowing a more dynamic view. In order to focus on identifying important processing and data flow within DFDs, the context diagram does not include any information that either indicates the frequency of interaction between the system and an external entity or the size/amount of data being transferred, which could have an effect on quality of service. Hence the context diagram is a device for setting system boundaries and identifying interacting external entities, ignoring any other requirements related to quality or constraints. Real-time extensions to the Demarco analysis method were made separately by Ward and Mellor [25] and Hatley and Pribhai [18] whereby, for the latter, frequency of system inputs and outputs together with the concept of response times were included. Both extensions included the concept of event/control flows as separate from data flows. However, both methods also retained the context diagram, but with the inclusion of event/control flows. In Object Oriented Analysis (OOA) the term context has not been used explicitly, although as with event-lists, the use-case approach can provide context both implicitly and explicitly. Similarly to event-lists, use cases describe a sequence of interactions between the system and the external entities called actors [26]. However, unlike the (DFD) context diagram, sequence diagrams and collaboration diagrams are not used in the early stages of analysis, but rather in conjunction with the more detailed system/class state models [21]. A quick review of the change from using *context* in structured to object oriented analysis, leads us to conclude that more contextual requirements have been realized. In addition, the focus shifted from analysing the system as a whole to focus on a single instance of usage of the system through the use of scenarios [26]. In both cases, the use of context didn't go beyond defining system boundaries and describing interaction between entities.

Context is realized, in software architecture, without adopting the term. In the review of software architecture analysis methods by Kazman et al. [19], the importance of realizing context is stressed as being the first criteria for analysing any software architecture analysis method. The term context is joined with goal identification, which reflects the view that context is synonymous with scope as used in other analysis methods. Additionally, the term *context* takes a new dimension, when it is used to reflect on which state a system is in [19]. Context is used also to mean constraints [19]. The view of context between state and constraints indicate the loose use of the term. Another view is presented by Bosch [9], who uses context to define boundaries and interfaces with external entities, as used by traditional analysis methods. The view is extended by linking functional and non-functional requirements to each interface defined by the context [9]. Therefore, realizing the importance of the role of *context* in architecture requirements. Requirements, however, are not all significant to architecture. Only requirements that have an impact on quality are identified [8]. Yet, it is not clear how to determine whether a requirement is significant or not. Experience and judgment are essential [8]. Nevertheless, requirements with impact on modifiability [20] and usability [7] which could be achieved through architecture [14], we believe can only be identified through understanding of *context*.

Design patterns in part are based on the architectural work of Christopher Alexander and his colleagues [5] in the book ”A Pattern Language”, where design solutions are captured from existing houses and cities to form a language for design. Although the same concept is adapted for software, a pattern language was not formulated to the

extent that complete programs can be designed [17]. Alexander [4] expressed his view that software design patterns should be developed to work together and be aimed at improving human life. Although each architecture pattern by Alexander et al.[5] has included a description of the context before introducing the pattern, this structure was not followed by all in software patterns. In the patterns by Gamma et al. [17], the context is replaced by other aspects, intent and motivation, that introduce each pattern. In Buschmann et al. [12] context is presented in the beginning of each pattern followed by a description of the problem the pattern solves. In the pipe and filter pattern, for example, the context statement is *processing data streams*, followed by an example from the real-world. Recently more attention has been given to addressing the issue of context in patterns. In the fifth volume of POSA [11], it is argued that context descriptions must be precise, but some context descriptions are so general they could be easily omitted. Many patterns such as the BRIDGE pattern, have been applied out of their defined context [11]. Such issue reveals the same sort of misunderstanding of the role of context, as in other approaches. We believe that the definition of context has not been given enough attention, not addressing what has been identified as the dilemma of context [23].

3 Context Centralised Method for Software Architecture

To recognize *context* in our methods it is important to realize that *context* itself is problematic [15], or as Scharfstein [23] puts it "the problem of context is too difficult for philosophers or anyone else to solve". The context problem is described as an element of design that cannot be properly defined [2]. Alexander [2] attempts to resolve this issue by realizing a set of forces that create form. Only by designing to temper the influence of such forces, will the form fit the context. Forces are seen as a result of context, yet other forces might not be recognized relative to their influence. Forces with less influence would be deemed non-contextual. Accordingly, it is practical to account for forces that have strong influence while recognizing that context is dynamic, it is important to account for change as well.

Most of the contextual information and our reaction to its influence seem to be common sense. This takes us back to the idea presented earlier that we are more aware of context in reality than in theory. What is true about reality in life is also true about software; the view of context has changed gradually as presented earlier, a change that derived more contextual requirements. It is not clear how much context-information is needed, to reach better requirements. Still we realize that there is a certain limit to our ability in handling the overwhelming amount of contextual information of our surroundings. As the amount of information to analyse in a project grows by a factor proportional to the contextual elements recognized, we doubt the ability of current methods to handle such challenges. The fact that everything is realized in a context [23] means that nothing exists outside of context. Therefore, it is important to account only for those aspects of context that are significant.

3.1 Method Overview

We propose a method for architecture development that helps to identify necessary changes to pattern structure to realize quality through context, by applying a two step

approach. The first step is to analyse the dynamics of context, where context statements, either described for requirements or design, should be verified and their significance established relative to their environment. This is achieved through the Context Dynamics Matrix. The second is based on evolving patterns, either architecture-patterns or design-patterns, through context derived requirements. The requirements are represented in separate domains, mainly functional and non-functional requirements domains, to be linked to patterns using an Aspect-Oriented Thinking (AOT) approach [16] to software requirements. In the following sections, an overview of the two approaches is presented.

Context Dynamics Matrix. In the Context Dynamics Matrix (CDM) [6], we acknowledge four states of requirements in relation to context; the states are realized based on contextual or non-contextual requirements derived from explicit or implicit context, resulting in the matrix depicted in Figure 1 . A certain requirement is contextual relative to the demand of the context for this requirement, the demand is represented as a force, where the weaker the force the less demand of the context, the less contextual it becomes, until it becomes non-contextual, when the force is easily neglected. This could be seen in inflated performance requirements, in network bandwidth for example, rates could be exaggerated beyond reasonable bounds. Such requirement would be non-contextual or contextual based on actual need. In both cases context could be explicit or implicit. An explicit context is the direct result of an observation, or undisputed statement(s) like: an apple is red, the sky is blue. Implicit context, however, is the result of an observation, like: sign is red (Explicit), which means danger (Implicit). The choice between explicit and implicit context is influenced by cultural differences and/or norms of conduct.

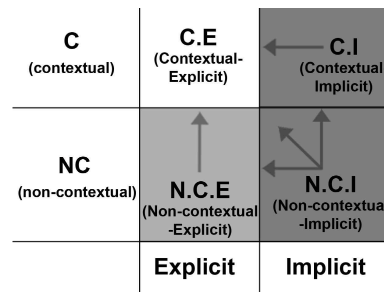


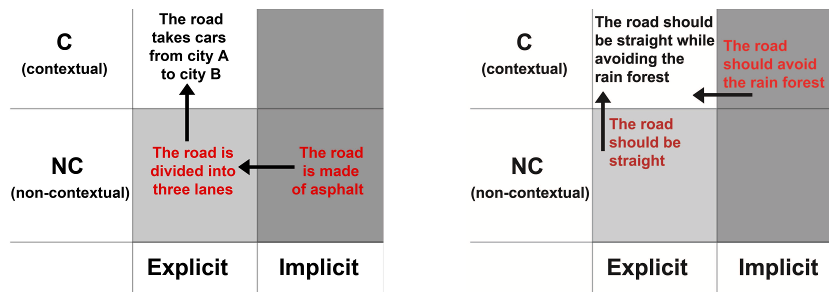
Fig. 1: A 2x2 matrix of possible contextual states, any one of which might apply to a situation under consideration.

The CDM approach could be applied to all sorts of situations across different disciplines. For example, consider the case of building a road connecting two cities. We can analyse the general requirements based on the context dynamics matrix in Figure 1. First we are presented with these set of requirements: *The road is to allow cars to travel quickly between city A and city B. The road is divided into six lanes - three in each direction and should be straight.*

For the first category we begin with the following statement: *The road is to allow cars to travel quickly between city A and city B*. The statement resembles a mission statement of the road, differentiating it from other roads, like a road connecting two industrial complexes, which makes it contextual and explicit. The following statement: *The road is divided into six lanes*, is also explicit since it specifies how many lanes the road must have, however, it is not clear whether it is a contextual statement or not. One way to clarify this statement is to determine the basis of the choice of three lanes, like if the road regulations demand that any road between two cities of population of more than 100,000 combined would require a connecting road of six lanes. The result of such a finding will move the statement from being non-contextual-explicit to contextual-explicit, as shown in Figure 2a.

The requirements statement leaves the road materials un specified, if the material is assumed to be asphalt, a new statement is added:*the road is made of asphalt*. Unless the statement is verified to be correct, it would be implicit. Any assumption derived from previous experience or an interpretation of meaning would be considered implicit. It is also non-contextual until the material is linked to either a constraint enforced by a regulation or a contextual reason is revealed. This is not to say that if the statement is non-contextual it would not be implemented. In the case of conflict between requirements, however, a non-contextual statement would be overruled by a contextual statement and an implicit statement would be overruled by an explicit one. If the statement is confirmed true, this will move the statement from non-contextual-implicit to non-contextual-explicit as shown in Figure 2a.

The second category of requirements concerning the road's shape starts with this statement: *the road between city A and city B should be straight*. The statement is explicit about the shape of the road, but a question might be raised if this requirement is contextual. The statement is non-contextual but explicit until the shape of the road is linked to a context. To make the statement contextual and explicit, the environment has to be studied, providing a justification for the decision, accounting for the forces



(a) The first requirements category showing two statements, one is moving from implicit to explicit context, the other is moving from non-contextual to a contextual state.

(b) The second requirements category for the road's shape synthesized to a contextual-explicit state using a contextual matrix

Fig. 2: Context Dynamics Matrix used to analyse two requirement cases.

applied by the context. An example of such a force is a rain forest in the way of the road. In considering the forces strength, which is in this case protecting the ecological stability of the rain forest, leads to consider the statement non-contextual. Therefore, a modification of the requirement is derived: *the road should avoid the rain forest*, which is implicit but contextual. It is contextual because it is derived directly from the context and it is implicit because it has not been verified formally. The conflict between the two statements, the original and the derived, is resolved by the new statement: *The road between city A and city B should be straight while avoiding the rain forest*. The new synthesised statement is now contextual and explicit, and thus less prone to ambiguity, as shown in Figure 2b.

Requirements, as presented earlier, can be moved from one state to another depending on the context. While the same principle applies, in some cases, requirement state changes are unlikely to happen on a short period of time. Requirement decisions, as in road construction projects, are governed by standards which are regulated and approved by road commissions. It is, however, important to follow the state of requirements until all of the requirements are clarified.

Context-pattern Evolution. Patterns for Architecture and Design are intended to codify knowledge, either in structure or in code [24]. Alexander [3] remarks that patterns can be realized in a "million different ways" , yet software patterns are mostly realized in one form. Design-patterns, although very useful in dealing with specific design issues, provide narrow focused solutions, resembling templates more so than patterns. The emphasis on context and forces must balance the emphasis on structure [11], allowing a pattern to be more responsive to change, hence enabling it to evolve.

In some cases there has to be a synthesis between design-patterns and architecture patterns, allowing a pattern to be realized in different ways, it can be generalized as an architecture pattern in one instance, or localized to be a design-pattern for code, in another. Some well recognized patterns are already synthesized, like the MVC pattern, which has been listed as an architecture pattern [12] and a design pattern [17]. We believe that all patterns, design or architecture, share deep structures that serve a common function. It is not *function* which differentiates between design and architecture patterns, it is *realization* . For example, the function of caching could be realized in different ways, in object orientation through a PROXY pattern, in structured programming as a procedure, or in architecture as a client-server. It is, however, the context which leads one to decide how to realize a given pattern.

To illustrate the link between context and patterns we present a case of transferring an image to a client, using object oriented software, according to the following context: *transfer one image upon request from a client to browse and save*. Initially, the pattern consists of a sender and a receiver, as depicted in Figure 3a, and is assumed not to encounter any retarding forces, given that the size of images are small and the time to process a request is minimal. If the context changes, the pattern has to adapt by changing its method of delivering the images to the client. The pattern, for example, should change when images get larger, resulting in a longer transfer time, assuming that everything else remains constant. Therefore, the context changes to: *transfer one large image upon request from a client to browse then save*. To adapt, the pattern's structure is redesigned, where the browsing and saving process are separated. Rather than browsing

the full image, a smaller size image is browsed first, then the image is downloaded later if needed. The separation is possible because the browsing activity is mentioned in the context statement. Alternatively, if the images are not browsed, then the design would not be useful. Consequently, the image object would split into two objects, one holding the full size images and the other holding the reduced size images, applying the PROXY pattern as demonstrated in Figure 3b. The client therefore would browse faster, while the speed at which the full size image is retrieved would not be affected. The structure serves its purpose until the context changes again. The absence of restrictions on viewing images is an absence of a force which does not exert any influence that calls for the structure to be changed. If we change the context again: *transfer large images upon request from an authorized client to browse and then save*. A new force exerts its influence causing stress on the structure which has security consequences. As a result the structure must be re-factored. A new strategy is used to separate the proxy objects into two SINGLETON objects, one encapsulates all the images, and the second encapsulates the unrestricted images, as depicted in Figure 3c. The two SINGLETON proxy objects restrict the number of objects used, applying more control over content. The decomposition of the proxy object is a result of the changed context of authorized and non-authorized users. The response to the change of context results in (perhaps) a viable solution. The ability, however, to meet the goals or qualities expected, either stated in functional or non-functional requirements could be disputed. To address this issue, we need to determine the functional/non-functional impact(s) of certain forces on a pattern. In addition, to the type of qualities a solution is aiming to achieve. We adapt the AOT [16] approach to link a pattern to at least two domains, the functional domain and the non-functional domain.

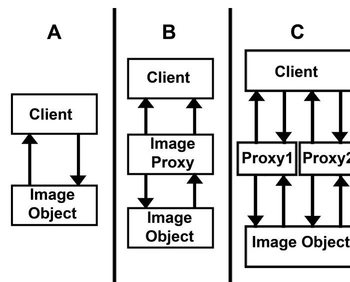
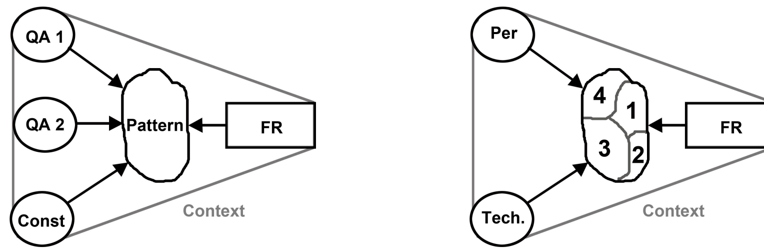


Fig. 3: The evolutionary growth of a pattern according to a changing context.

3.2 Method Description

The method is separated into two levels of abstraction, which can be considered as stages of development in the course of the systems life cycle. The first level is analysing requirements through context, using the context dynamic matrix, the requirements are derived from or modified and purified by the matrix. The resulting requirements from the context are a set of forces and expected outcomes linked to design/architectural



(a) Four forces influencing the shape/form of a pattern.

(b) An RCS evolving pattern according to function, performance and technological forces.

Fig. 4: Patterns influenced by forces realized from different domains.

patterns. The second level is to consider the separate functional and non-functional domains/contexts.

As represented in Figure 4a, an architectural pattern is influenced by four contextual forces - three non-functional forces and one functional force. The number of forces could vary. However, a pattern is a direct result of at least two forces, a functional force and non-functional force or quality-oriented force. On the one hand the functional force represents what a pattern has to achieve, on the other hand, the non-functional force(s) represents quality goals in achieving what the pattern has to do. If a pattern is not able to achieve what it has to do in accordance with quality goals, then a trade-off must be made between the quality or the functional goal.

A good example of changing patterns according to realized forces is the Real-time Control System (RCS) reference architecture model [1]. The architecture-pattern has evolved over 20 years in four stages, from RCS-1 to RCS-4, as depicted in Figure 4b. The RCS started in the 70's as a simple state machine in RCS-1 which, depending on a feedback, input commands were chosen to determine behaviour. As the technology changed in the 80's, a sensory processing algorithm was added to the second model - RCS-2. This change allowed the function of the RCS model to expand, and it was applied in the Army Field Material Handling Robot (FMR) and the Army TMAP semi-autonomous land vehicle project. In the late 80's the RCS-3 model was refined by adding a third model called the World Model view which carries the task of planning and sensory processing. The change in technology of that time offered new opportunities and demands for better performance. The processing features of the RCS-3 were enhanced further in the RCS-4 in the mid 90's. Stronger processing power capabilities and memory storage allowed the addition of a Value Judgment (VJ) system, which computes cost, benefit, risk of planned actions, and assigns values to objects.

4 Advantages and Open Issues

The method so far presented provides several advantages that we believe should enable software architecture and software in general to be more responsive to change. However, we also recognise that there are a couple issues to be resolved in order that

the method to remain advantageous. In the following sections both the advantages and open issues surrounding the method are discussed.

4.1 Method Advantages

We summarise some of the advantages of adapting the CDM approach on requirements, architecture and design in the following points:

1. Contextualised and Unambiguous Requirements: The use of CDM to analyse context in any stage of development, gives the chance for requirements to be verified and made clear. Such analysis allows for unrealized requirements to be derived as well. The communication gap between stakeholders on one side, and system developers on the other, could be bridged using the approach.
2. Gradate Contextual Requirements based on Influence: Requirements are classified according to contextual forces. Such a process allows systems to achieve targeted goals effectively. Decision makers, developers and other stakeholders, can focus better on critical requirements whilst still being aware of the relative importance of less critical requirements. This can lead to projects being managed with clear goals and objectives.
3. Determining Patterns of Change: Identifying forces and realizing their impact on the system, provides the opportunity to determine patterns of change. In the RCS example by Albus [1], a pattern of change brought about by the contextual influence of technology and its subsequent impact on performance could be identified. This allows the environment and its force changes to be expected, allowing forward planning and effective resource management.
4. Joining Architecture and Design Patterns: The evolution of the reference architecture by Albus [1], enabled numerous pattern changes at different levels. The differences between patterns could be bridged through identifying forces and their impact(s). Through designing for context, it is easier to understand what influences architecture and its components structurally. Thus leading to a more transparent architecture.

4.2 Open Issues

There are a couple of issues that need further attention in order for the method to be applied effectively. The first issue is measurement of forces to determine order of strength and weakness and thus, influence. Context is considered influential according to force/impact, which is determined in a judgemental fashion, perhaps based on personal experience. An objective way to measure contextual forces has not yet been determined. Without an objective measure a fully gradated contextualisation might be difficult. The second issue is how to account for the inherited complexity of context. Context is inherently complex. Especially given the fact that each element of a context has its own context, which may result in an endless series of contexts. Every implicit context is an open door to further interpretations. Context is also made difficult by relativity, since it takes its meaning relatively. A point of reference must be established to define context for an object/situation. If we take the example of rain, we can understand

a person who is not thinking of going out might be oblivious to the rain context, she/he is indifferent. The matter, however, is different for somebody who is planning to go out. Therefore, knowing of such easily changeable forms of context is limited by experience when there is no objective way to easily identify and analyse context.

5 Conclusions and Future Work

The aim of our research is to establish a leading role for context in requirements elicitation, analysis and specification and to enable controlled software architectural adaptation and evolution based on dynamic context analysis. A change in perspective to the way *context* is being utilized in software engineering, we believe, would promise the realisation of more adaptive and elastic software systems. The affect of such change would lead to a more uniform view of patterns, realising common deep structures between architecture-patterns and design-patterns, allowing both types of patterns to be integrated horizontally and vertically in the process of developing software.

The development of software architecture/design for long-lived systems is an ongoing process which spans years and ever-changing environments/contexts. To avoid the early retirement of software systems requires planning and foresight including an understanding that it is context that causes requirements to change and systems to adapt or retire/perish. Although not fully realised such holistic approach of system development is plausible but currently not a part or our research.

So far the work that has been done covers the first stage of the methodology, consisting of analysing context using the Context Dynamics Matrix. In order to establish the efficacy of the method a case study will be performed where the matrix will be used to analyse usability requirements for the Electronic Voting and Counting System (eVACS) user interface. This system has been used for both the Australian ACT and Federal parliamentary elections where different user interfaces were required. The case study should demonstrate how the matrix clarifies requirements for developers and other stakeholders, reflecting its usefulness in developing a highly usable interface. The context-pattern evolution approach would be demonstrated through two pattern views, a static view and a dynamic view. In the static view the aim is to show how *form* in one instance could achieve quality through a set of refined contextual requirements. In the dynamic view, a couple of instances would demonstrate the progressive change in *form* as a response to the change of *context*. In adapting the dynamic view over the static view, the software process moves from adapting a single act of creativity to a continuous creativity process.

References

1. Albus, J. S. (1992). A reference model architecture for intelligent systems design. In Antsaklis, P. J. and Passino, K. M., editors, *An Introduction to Intelligent and Autonomous Control*, pages 57–64, Boston, MA. Kluwer Academic Publishers.
2. Alexander, C. (1964). *Notes on the Synthesis of Form.* Harvard University Press.
3. Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.
4. Alexander, C. (1999). The origins of pattern theory: The future of the theory, and the generation of a living world. *IEEE Software*, pages 71–82.

5. Alexander, C., Ishikawa, S., with Max Jacobson, M. S., Fiksdahl-King, I., and Angel, S. (1977). *A pattern language : towns, buildings, construction*. Oxford University Press.
6. Alshaikh, Z. and Boughton, C. (2008). The context dynamics matrix. To be published.
7. Bachmann, F., Bass, L., Klein, M., and Shelton, C. (2005). Designing software architectures to achieve quality attribute requirements. In *IEE Proc.-Softw*, volume 152, page 153:165. IEE.
8. Bass, L., Bergey, J., Clements, P., Merson, P., Ozkaya, I., and Sangwan, R. (2006). A comparison of requirements specification methods from a software architecture perspective. Technical report, Software Engineering Institute , Carnegie Mellon.
9. Bosch, J. (2000). *Design & use of software architectures*. Addison-Wesley, London.
10. Bübl, F. and Balser, M. (2005). Tracing cross-cutting requirements via context-based constraints. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 80–90, Washington, DC, USA. IEEE Computer Society.
11. Buschmann, F., Henney, K., and Schmidt, D. C. (2007). *Software-Oriented Software Architecture On Patterns and Pattern Languages*. John Wiley & Sons, Ltd.
12. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture A System of Patterns*. John Wiley & Sons.
13. DeMarco, T. (1979). *Structured Analysis and System Specification*. Yourdon Press Upper Saddle River, NJ, USA.
14. Dey, A. K. (2001). Understanding and using context. *Personal and Ubiquitous Computing*, 5:4–7.
15. Dilley, R. (1999). *The Problem of Context*. Berghan Books.
16. Flint, S. (2006). *Aspect-Oriented Thinking An approach to bridging the disciplinary divides*. PhD thesis, Australian National University.
17. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley.
18. Hatley, D. J. and Pirbhai, I. A. (1988). *Strategies for Real-Time System Specification*. Dorset House.
19. Kazman, R., Bass, L., Klein, M., Lattanze, T., and Northrop, L. (2005). A basis for analyzing software architecture analysis methods. *Software Quality Journal*, 13:329–355.
20. Kazman, R., Bass, L., Webb, M., and Abowd, G. (1994). Saam: a method for analyzing the properties of software architectures. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 81–90, Los Alamitos, CA, USA. IEEE Computer Society Press.
21. Mellor, S. J. and Balcer, M. J. (2002). *Executable UML A foundation for Model-Driven Architecture*. The Addison-Wesley Object Technology Series.
22. Potts, C. and Hsi, I. (1997). Abstraction and context in requirements engineering: Toward a synthesis. *Annals of Software Engineering*, 3:23–61.
23. Scharfstein, B.-A. (1989). *The Dilemma of Context*. NYU Press.
24. van Lamsweerde, A. (2000). Requirements engineering in the year 00: A research perspective. *icse*, pages 5–19.
25. Ward, P. T. and Mellor, S. J. (1986). *Structured Development for Real-time Systems*. Yourdon Press.
26. Wiegers, K. E. (2003). *Software Requirements*. Microsoft Press.
27. Yourdon, E. (1989). *Modern Structured Analysis*. Prentice-Hall International Editions.