1.5 Patterns and Pattern Languages

So far, this literature survey has reviewed project outcomes, procurement, requirements representation and effort estimation. In the third section on RE, it was suggested that use case models might improve procurement if they were included in tender documents. The objective would be to make requirements less ambiguous. This section describes patterns and pattern languages, and suggests that using this approach may be employed to reduce the time needed to construct use case models, making their usage more commonplace.

A pattern is a way of capturing best practice and recording it in a standardised manner so that it can be reused by others who face a similar problem without having to begin from first principles. The notion of patterns, in both structures and relationships between those structures, was first articulated by Christopher Alexander [1, 2] Alexander was an architect, and his work was in the domain of designing buildings and communities. Simply put, he defines a pattern as "a three-part rule, which expresses a relation between a certain context, a problem, and a solution."

Patterns are related to, but different from, idioms, principles, heuristics, or paradigms. An idiom is a language specific way of combining elementary building blocks that holds true for a particular programming language but is not valid more generally. A principle, on the other hand, is an invariant that holds true more globally and is not constrained by a particular language. An heuristic is an aid to decision making, although it makes no absolute claim as to the goodness of the action suggested. Heuristics could be used to choose among alternative patterns. A paradigm is a very abstract pattern [3]. In comparison, a pattern is a more concrete concept that provides detailed, but non-proscriptive instruction, regarding the solution to a standard, recurring problem that is presented in a formalised written structure [4]. They are therefore a mechanism for the communication of concepts designed to be reused. They provide effective 'shorthand' for communicating complex concepts effectively. Patterns are a solution to a problem in a context [5]. They help new developers ignore traps and pitfalls that normally can only be learned from costly experience. Patterns have a context in which they apply. There are advantages and disadvantages to the application of any particular pattern, thus they are subject to opposing forces that must be balanced. All

solutions have costs and these costs are explicitly stated. There is nothing new about the concept behind patterns, as by definition they capture experience. The pattern movement is a systematic attempt to document common abstractions in the domain of software engineering [6].

Coplien says that patterns are the core of techniques, principles and values which have proven useful. On one level, they are just a form of documentation; a tool in the designer's toolkit [4].

A collection of unrelated patterns is termed a 'pattern catalogue'. When patterns are combined, or links are made between related patterns, they form a 'pattern language'. A pattern language is a collection of patterns that build on each other to generate a whole, whereas a pattern in isolation solves only an isolated problem. A pattern language describes a process framework through formal rules. It is a collection of rules that build all members of a family [4]. Through pattern languages, patterns achieve their fullest power. For Alexander, there exists a pattern language capable of defining and designing all houses, from those of Cape Cod, to those found in southern Italy and built of stone [2]. A natural language, such as English, can be seen as a pattern language capable of generating all sentences. A pattern language has structure, but is unlikely to be so finely grained or so formally structured as is found in the grammar that underpins a natural language. A pattern language is not generally hierarchical; rather it is a directed graph where nodes have many to many relationships [4]. However, in the example presented by Cunningham [7] of a pattern language for competitive development in the software industry, the work is expressly described as being 'top-down or hierarchical' [7] implying that pattern languages may take different forms and need not necessarily conform to a graph structure.

A *design pattern* can be thought of as a particular type of pattern that exists in the domain of software engineering, concerned with solutions in the realm of object-oriented programming. There are many domains in which patterns may be described; however, design patterns were the first to be articulated through the application of Alexander's original concept and applied to software. They are liable to be equally applicable at other stages of software engineering, and indeed across many unrelated disciplines. In future, pattern languages are likely to emerge in more narrow application domains, (e.g. financial transaction processing) and to continue to grow in importance as the discipline matures [4]. The long term

goal of the pattern movement is to develop handbooks for software engineers [6] because they are an expression that is both accessible and semi-formalised.

Underpinning the whole pattern movement is the explicit attempt to harness patterns of best practice to the improvement in software quality. It is intended that this should directly improve productivity and customer satisfaction while indirectly contributing to a reduction in development costs through shorter project timescales. It is envisaged this should be possible through a 'short circuiting of the discovery interval' thereby avoiding rework through the reuse of intellectual capital. Patterns strive to bring knowledge into the open where in the past they have been part of the discipline's folklore [4].

A Brief Pattern History

Christopher Alexander first articulated the concept of a pattern, and inter-related patterns (pattern languages) as templates for the design of better buildings and communities. The most well-known example of the application of his pattern-based approach is in the design of the University of Oregon. Alexander prophesised that the concept of pattern languages was not restricted to the field of architecture and fully expected the approach had application in other fields [1, 2].

Some of the earliest work in software patterns was done by Kent Beck and Ward Cunningham. Beck had been exposed to the ideas of Alexander as an undergraduate at the University of Oregon (Alexander had used his patterns in the design of that institution). His room mates were studying architecture and so had naturally been exposed to Alexander's work. Beck read the 'Timeless Way of Being' [2] standing up in the university book shop over the course of several months. The things Alexander did not like about architects seemed to be the same things Beck did not like about software engineers and so he was drawn to the idea that the principles might be reapplied in this new domain. In trying to assist a company with the design of a new user interface, and following Alexander, who said that the occupiers of a building should help to design it, he tried the same approach in the design of a application's user-interface. Working with his friend, Ward Cunningham, they devised a simple pattern language, comprised of five component patterns (although not presented in a formal pattern structure), which are summarised below.

Window per task: there will be a specific window for each task the user must perform. *Therefore*, All information needed to perform the task will be available in the *Few Panes* of the window.

Few Panes: To understand complex things one often must *see* it from several points of view, *therefore* different points of view (called simply View) must be provided by dividing the area of your *Window Per Task* into panes.

Standard panes: one must learn to operate each kind of pane offered in the *Few Panes* of every window, *therefore* cast each pane into the format offered by one of a few standard panes. Limit these to Text, List, Table and Waveform.

Nouns and Verbs: Things exist while action happens, *therefore* put lists of things (nouns) in a list pane (one of *Few Panes*) which persists through interactions. Put actions (verbs) in 'Short Menus' which pop-up and then disappear as the action commences.

Short menus: elements of pop-up menu must be visually searched repeatedly, *therefore* make them short, fixed and single-level [8].

Beck and Cunningham were pleased with the results of the application of the language in the design of their user interface. The results were reported at OOPSLA 87 but their approach did not have the immediate impact they had hoped for [4].

Erich Gamma was using ideas about recurring design structures in his Ph.D. thesis, without being clear how they could best be communicated. In 1991, prior to the European Conference on Object-Oriented Programming (ECOOP) in Zurich, Gamma and Richard Helm agreed on a set of candidate patterns that could be put together to form a pattern catalogue. These patterns became the basis of their later seminal book 'Design Patterns' [9] in 1995, with the further participation of Ralph Johnson and John Vlissides who would become known as the 'Gang of Four' (GOF).

In August of 1993, (prior to the publication of the GOF book), Kent Beck and Grady Booch sponsored a mountain retreat in Colorado to discuss the implications of patterns in software engineering. They discussed Alexander's ideas and how they could be applied by designing a building on the site of the meeting where ideas about object-oriented programming could be discussed between practitioners and

customers. The group continued to meet and became known as the Hillside Group. They met in April 1994 to plan the first Pattern Languages of Programming (PLoP) conference, which spawned equivalent conferences around the world which continue to this day.

Pattern Forms

There are effectively three different styles of pattern presentation. In general, all the different forms include the following elements:

Name: a pattern name is important because it draws the reader's attention and acts as the first clue as to whether a particular pattern is likely to be relevant in the solution to the current problem under consideration. The name encodes the patterns meaning. Some examples of pattern names are *Window Per Task*, *Ambassador*, or *Remote Proxy*, *Leaky Bucket Counter*, or *Fool Me Once*. As is evident from the examples, a name may recall an obscure analogy. A name may take the form of a noun or noun phrase [1] such as *A Place to Wait* or a verb or a verb phrase [10] such as 'Get Result'.

Intent: a summary of the Problem section that states what the pattern does, and the particular problem it resolves.

Problem: this section describes the problem to be solved. It is a concise statement that helps the reader to decide whether to read further. It serves as a primary index. In some forms, the *forces* are combined into the *problem*.

Context: the context section includes a history of patterns that have been defined before the current pattern being described was considered. It specifies aspects of size, scope, market, language or, in fact, anything else that, if changed, would invalidate the pattern. Context is crucial to understand what patterns work together. Context weaves patterns together into a pattern language. This section tends to mature with experience.

Forces: patterns are not rules that are followed blindly. They must be understood and tailored to a specific need. It is important to understand what 'tradeoffs' are likely to have to be made. The forces section helps to understand how a pattern can be applied

effectively. A single pattern can be applied many times without it ever resulting in exactly the same outcome. In software the term ‘force’ is used figuratively, because there are rarely physical forces that must be balanced. Forces determine why a problem is difficult.

Solution: the solution section must detail what specific action is to be taken, but it must remain general enough to address a broad context. Some patterns provide only a partial solution with paths to other patterns.

Sketch: the sketch conveys the problem’s structure. This is the section where an example solution structure is presented, normally through the employment of diagrams. A sketch is not a graphical specification *per se*, but it provides the foundation of a specific specialised solution to the specific problem under consideration by the practitioner. Readers may interpret diagrams too literally which may be avoided through the employment of hand drawn diagrams.

Resulting context: The resulting context wraps the pattern up; it describes the situation that exists once the pattern has been applied. It describes what forces have been resolved, and what new problems may have arisen. This section contains instructions as to what patterns other patterns may be further applied.

If a pattern is like a play, by analogy, the context introduces the characters and the setting. The forces provide a plot and the solution provides a resolution of the tension built up in the conflict of the forces [4].

There are alternate forms of a pattern’s structure, yet they all share much the same content. The *Alexandrian* form is the original upon which the variations are based. In the Alexandrian form the component sections are not strongly delimited. They always include the word ‘therefore’ immediately preceding the solution. Each pattern starts with an introductory paragraph that states the patterns that must already have been applied to make the ensuing pattern meaningful. *Alexandrian* patterns have star ratings which indicate how much confidence the author has in the pattern, presumably based on experience of their application. This form has the distinct advantage of being brief, at the expense of being detailed and may be more suitable for patterns of organisation with a lesser emphasis on programmed solutions.

The *GOF* form is specially adapted for OO software design. Typical pattern sub-headings of this form are introduced below.

Motivation: a scenario that illustrates the problem and how the class structure solves it. This section is intended to help understand the more abstract description that follows (similar to *Sketch* in the ideal form) .

Applicability: describes situations in which the pattern should be applied.

Structure: presents a graphical representation of classes.

Participants: details the classes that participate in the pattern.

Collaborations: describes how participants work together to carry out their responsibilities.

Consequences: states how the pattern supports objectives and what 'trade-offs' must be accepted.

Implementation: describes the pitfalls, hints, techniques and specific issues that may be encountered when implementation is undertaken in any particular programming language.

Sample code is the section where language code fragments or pseudo-code may be represented.

Known uses presents examples where the pattern has been applied successfully in real applications.

The *Portland Form* is primarily an 'online' form that takes advantage of hypertext links to make connections and associations between patterns. It is similar to the Alexandrian form and was first presented at PLoP 94 [3]. It is written in a narrative style as opposed to the outline form introduced in the idealised structure and refined in the GOF form.

Types of patterns

It has already been shown that patterns exist to describe architecture and implementation problems in OO programming. An excerpt example of a complete design pattern is presented below from the GOF book [9].

Name: Singleton

Intent: Class has one instance and a global access point

Motivation: An accounting system will be dedicated to serving one company. Class is responsible for tracking its sole instance. Class ensures no other instance can be created.

Applicability: One instanceAccessible to clients from well-known access point

Structure

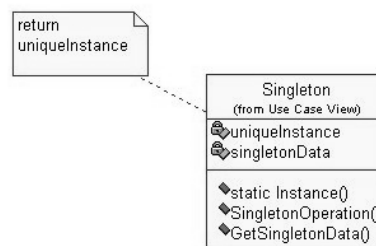

Figure 1.5.1: The structure of the Singleton pattern is represented in diagrammatic form.

Participants: Singleton

Collaborations: Clients access a Singleton instance solely through Singleton's instance operation.

Consequences (benefits): Controlled access. Reduced name space – no global variables. Singleton class may be sub-classed. Population at runtime. Easy to change your mind and limit to x instances

Implementation: Hide the operation that creates the instance behind a class operation that guarantees only one instance is created. The constructor is protected. Once a singleton is created it is mapped to a registry. When a client wants it, it consults the registry. Therefore it is not necessary to go through the class every time to find the instance.

Specifically, the class of OO programming problems addressed by Design Patterns [9] can be simplistically understood to be those that combine part of a 'software architecture'. There is no definitive definition of software architecture; certainly none that has been universally agreed.

The most straightforward, and shortest definition of architecture may be 'the way the parts work together to make the whole' [11]. It is a useful characterisation to make the distinction between architecture that supports NFRs, and that which supports 'functional' requirements. A functional requirement is one that would be recognised as a task by a user of the system, and may be represented profitably with a use case. A NFR, on the other hand, supports the delivery of the functional requirements; it is the 'how' by which behaviour is experienced, as opposed to the 'what' of the available behaviour. Therefore, a single NFR (the system will be available 23 hours a day, 7 days a week) supports many functional requirements (login, new booking, modify departure time, etc.). Having made this distinction between the two fundamental types of requirement, it has been suggested that design patterns solve problems in the domain of NFRs [11-13]. Design patterns can be seen as a mechanism for describing a NFR architecture.

IBM have done some interesting work with patterns, in the realm of software and hardware architectures, that help in the specification of the way machines should be configured to support the level of service that is desired. In his book, Lord [14] draws upon the experience of IBM in both hardware and software. He states that although development projects differ, they have much in common. Typically 20% of an application development project is unique, while fully 80% of a project can be approached with well-proven software and techniques. IBM identify the following patterns to help in the categorisation of development effort [15, 16].

- Self-service: applications where users are interacting with enterprise transactions and data.
- Collaboration: applications where tools facilitate communication among users.
- Information aggregation: applications where tools extract information from other data sources.
- Extended enterprise: applications that integrate programmatic interactions among organisations.

IBM could be said to have described a pattern catalogue that describes a type of network and hardware architecture. Additionally, as a software vendor, the IBM patterns describe a service architecture that helps in the specification of such software services as offered by 'application servers' such as Websphere. Websphere is an example of a class of service 'container' that supports an execution framework (in this

case, Sun's Java EJB framework). Components built under the EJB framework need software support, and patterns help in the specification of the number of copies required and in the ideal number of servers (dependent on the geography of the enterprise).

From the single perspective of having defined a pattern form for communication, describing solutions to recurring problems, regardless of domain, is liable to be helpful, because structure allows the reader to focus on content, without being distracted by questions of form. Patterns occur in such diverse domains as training, project management, and organisation [4]. In fact, there is no reason that the pattern form cannot be applied to any domain where it may be useful. The restriction on this statement being that the pattern should be of *sufficient* utility; which only becomes clear over time.

There are many tasks to be performed in software engineering apart from answering questions of implementation. Fowler coined the term 'Analysis Patterns' in his book of the same name [17]. These describe groups of concepts that represent a common construct in business modelling. They may be useful in only a single domain (such as healthcare), or they may span more than one business domain. By this approach, OO techniques can be used to formulate conceptual models in analysis that begin to demonstrate similarities to business process engineering.

Robertson introduced the notion of 'requirements process patterns' and she proposes how they might be discovered, represented, and reused [18]. These patterns can be categorised by name, input and output definitions. Each pattern focuses on the processing of a single use case. Context analysis and use case partitioning provide a way of reusing relevant patterns early in the lifecycle. Unfortunately, she neglects to follow the pattern form, and thereby makes her argument difficult to fit into the existing artefacts of the wider pattern community. Cybulski agrees that there is merit in considering patterns for the capture of recurring requirements as the basis of a dictionary of problem-solving approaches [19].

Without the incorporation of use cases, one of the most serious problems in dealing with early artefacts is their insufficient formality, lack of structure and incompleteness. Requirements documents are commonly expressed in natural language (e.g. English). Processing such information is difficult and it is not easily reused. It has been suggested that textual artefacts would greatly benefit from more general representation schemes [20]. It could be that use cases are ideal for the formulation of requirements patterns. Patterns can

be thought of as 'mined' from experience. In the arena of requirements capture, to achieve requirements reuse a taxonomic knowledge structure is required. Patterns are a natural form for the representation of a taxonomic structure because they include a mechanism that allows for cross-referencing. The majority of approaches to requirements reuse have focused on the semantics of requirements documents, with the attendant ambiguity contained in natural language expression. Thus, similar requirements may be expressed using distinct writing styles, different grammatical structures and inconsistent terminology. To counter this problem, requirements patterns may offer a solution [19].

Other approaches to requirements categorisation and representation have been earlier described, such as those of Jackson and Sutcliffe. Although these alternate approaches are not described in a pattern form, both make reference to patterns, and could conceivably be represented as patterns had this been their authors' intention. (Admittedly the structures themselves do not contain cross references to other structures, as is the case with patterns.) Apart from a difference in the form of their description, Jackson's problem frames and Sutcliffe's Object System Models are intended to act as catalysts for the recognition of commonality.

In many spheres, language can be as much a barrier as an enabler. The English language only contains a finite number of words, so the same word has different connotations. It becomes necessary to say a few words regarding what is meant by 'problem' and 'solution' in the discussion of patterns. How does a requirements pattern confine itself to a statement of the problem when a *solution* is a defined component of the pattern form? A requirements pattern, which describes a problem, provides only a solution to the problem's representation.

Some patterns solve problems, while others provide guidance as to how processes should be carried out. Use cases were introduced earlier in section three of this literature survey where some guidance was provided on how they should be constructed. A more formal treatment of this subject is undertaken in [21] that describes a rich pattern language for the production of good quality use cases and patterns for producing complete and coherent use case models. This book features the pattern *Clear Cast Of Characters* which is used to identify the actors a system must interact with and the roles they must play. *Breadth before depth* describes the need to develop an overview of use cases first and then to progressively add detail.

User valued transaction to identify the valuable services the system deliver to the actors to satisfy their business purposes. *Complete single goal* states that each use case must address one complete and well-defined goal, while recognising that goals exist at different levels of abstraction. *Verb phrase name* states that a use case name must contain an active verb phrase that represents the goal of the primary actor. *Precise and readable* ensures the use case text is readable enough for stakeholders, yet precise enough for developers to work from. There are many patterns described in this book for the generation of use cases (too many to list in their entirety). The language is useful in the description of well-formed models as opposed to the production of models that are functionally representative.

Biddle takes the approach that there exists a candidate set of use cases that appear commonly enough to be captured in a pattern language. Developed according to the Portland model, this language defines such use cases as required for standard reporting. Systems require comprehensive reporting, a fact that is too often overlooked. To some, reporting may be considered trivial or boring, yet essential to others. Therefore, reporting use cases are common and must be properly modelled. Equally, Biddle recognises that CRUD functionality is a reality for the production of database driven systems and therefore must be modelled [22]. He succeeds in suggesting an initial candidate set of use cases that is very useful in stimulating the initial modelling process.

Patterns may be divided into business patterns and software patterns. A business pattern can be defined as that which is useful in the formulation of needs and the definition of functional specifications. A software pattern is equivalent to a design pattern; it is the expression of a technical solution that leads to implementation. It may then be possible to provide a mechanism for mapping one to the other from which *design patterns* can be mapped to reusable components. This approach would allow for the specification and construction of new systems based on reusable artefacts following the heuristics contained in an all encompassing pattern language that existed for this purpose. In an example of this approach, patterns based specification was applied in the construction of a ‘Product Information System’ (PIS) for car production on behalf of Peugeot [23].

Pattern Summary

Alexander defines a certain essence of patterns as being ‘that quality without a name’ where the pattern is elegant, containing nothing extraneous, yet capturing the essence of both the problem and solution in a clear and unambiguous manner. Patterns may transcend domains. They are intended as guides to humans, not to machines. They should not be thought of as a means to turn everyday workers into experts but only as a way of helping the inexpert avoid errors. Patterns are not theoretical; they are about ‘real stuff’. They capture proven practice, rather than postulates or theories. The objective of pattern’s expression is to celebrate what works, not recommend action on the basis of intuition, argument and aspiration. As a minimum a good pattern should feature three examples that show different successful implementations. The pattern community has been described as having ‘an aggressive disregard for originality’; success is more important than novelty. The emphasis is on writing and the clarity of communication [4, 6].

Drawing on the adage that ‘knowledge is power’ it is remarkable that talented individuals should want to share their experience with a wider audience. Individuals with key technical knowledge may take the view that their expertise is a source of their job security and perhaps even their identity. On the other hand, there has always been a tradition of sharing knowledge in science, with the caveat that those who reuse intellectual capital should scrupulously adhere to the custom of citing the author of the original work. Certainly, from a chronological perspective, written knowledge outlasts experts (knowledge unshared dies with the person) and outpaces them geographically. Although hype surrounds all new methods in software engineering, the pattern community takes the view that hype should be actively discouraged [4] so as to minimise an ‘inflation of expectations’.

Patterns are not going to make experts redundant, nor will they make adequate designers, great designers. Patterns are not going to allow machines to do the work that was previously done by people. Patterns are not a *silver bullet*. Indeed, the openness of the pattern community contains within it some disadvantages. It is not always obvious that the pattern form is wholly applicable especially when a topic area might be considered too specialised as is the case with the pattern language described for the construction of an online auctions management system as presented at PLoP 2001 [24]. In this example it is hard to see how what is presented can rightly be described as a pattern language as opposed to being a software

specification because it does not address a sufficiently generalised problem. In this case, the work would fail a test of wide applicability (were such a test to exist).

When there are many patterns that address the same problem, it may be difficult to choose between them, especially when it proves impossible to state objectively which one is superior in a particular context. Conversely, there may be a wide *corpus* of problems for which no patterns have been defined. It is difficult to know what patterns can be ‘mined’ without having a reasonable idea of the scope of the entire topic under consideration. The subject of software engineering, for instance, is very wide, yet the pattern collections that are available do not come close to addressing all the processes and structures that might usefully be tackled. There is no academic journal devoted to the dissemination of patterns, which is perhaps because there is no academic community qualified to judge the quality of individual patterns. This limits the distribution of patterns, while at the same time provides no guidance as to their quality. The concept that patterns should be able to cite three examples of their application is a sound approach, yet there is no mechanism to capture their future application and consequent results to ensure they are, in fact, delivering value. This gap suggests that no mechanism exists for their further evolution and subsequent improvement and so a pattern may be published and become an isolated (perhaps, irrelevant) artefact. A serious problem with patterns is that there is no defined indexing service. It has been suggested that the name of a pattern, or its problem statement should act as a hook into its discovery, but this is a hit and miss approach that is inefficient for those seeking a particular pattern to a particular problem without the benefit of knowing if such a pattern even exists.

Whatever the limitations, it is clear that many practitioners receive measurable value from working with patterns. Fowler comments on the subject of some customer’s resistance, and others enthusiasm, when he says that for those who are wedded to the notion that they are unique he keeps hidden the fact that he is applying patterns. To others, the fact that the problems they face have been encountered before, comes as a relief, further amplified by the knowledge that a tried and tested solution is being applied [17].

Patterns are not driven by the academic world, they are driven by industry. This has its advantages and disadvantages. On balance, patterns that are clearly useful will survive and evolve, while those that are of

marginal interest will be forgotten. What is important is that they offer practitioners a form of expression, after which the pattern must live or die by its own utility.

Most excitingly, if it can be said that patterns generate architectures, and it has been shown there are different kinds of patterns, it follows that those different patterns will generate different kinds of architectures. Patterns of hardware may generate hardware architectures [16], design patterns may generate NFR architectures [11], and so it stands to reason that requirements patterns are capable of producing reusable functional architectures. Although these functional architectures have not yet been articulated, both Jackson's and Sutcliffe's work have laid the foundation for further progress in this area [25, 26].

[1.] Alexander, C., Ishikawa, S., M., S., Jacobson, M., Fiksdahl-King, I., and Angel, S., *A Pattern Language - Towns, Buildings, Construction*. (1976), New York, N.Y., Oxford University Press (OUP)

[2.] Alexander, C., *The Timeless Way of Building*. (1979), New York, NY, Oxford University Press (OUP)

[3.] Viljamaa, P., *The Patterns Business: Impressions from PLoP 94.* ACM Software Engineering Notes, (1995). **20**(1) Amercian Computing Machinery (ACM)

[4.] Coplien, J., *Software Patterns*. (1996), N.Y., N.Y., SIGS Books & Multimedia

[5.] Beck, K., Crocker, R., Meszaros, G., Coplien, J., Dominick, L., and Paulisch, F., *Industrial Experience with Design Patterns.* Proceedings of ICSE-18, (1996)

[6.] Schmidt, D., Johnson, R., and Fayad, M., *Software Patterns.* Communications of the ACM, (1996). **39**(10) Amercian Computing Machinery (ACM) http://www.cs.wustl.edu/~schmidt/CACM-editorial.html

[7.] Cunningham, W.: *Episodes: A Pattern Language of Competitive Development*. in *Pattern Language of Programming (PLoP 95)*. (1995), Monticello, Illinois

[8.] Beck, K. and Cunningham, W.: *Using Pattern Languages for Object-oriented Programs*. in *Object-Oriented Programming, Systems, Languages, and Applications Conference (OOPSLA87)*. (1987), Orlando, Florida: ACM Press

[9.] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns - Elements of reusable object-oriented software*. (1995), Upper Saddle River, N.J., Addison Wesley Longman

[10.] Meszaros, G., *A Pattern Language for Improving Capacity of Real-time Systems*. (1996), Reading, MA., Addison Wesley

[11.] Beck, K.: *Patterns Generate Architectures*. in *Proceedings of the 8th European Conference on Object-Oriented Programming*. (1994), p. 139-149: Lecture Notes in Computer Science, Springer Verlag

[12.] Gross, D. and E., Y., *From Non- Functional Requirements to Design through Patterns.* Requirements Engineering, (2001). **6**(1): p. 18-36, Springer Verlag

[13.] Cybulski, J., *The Formal and the Informal in Requirements Engineering*, 1996, www.dis.unimelb.edu.au/staff/jacob/publications/ informal-reqs%5CFormal-Informal-IEEE-Fmt.pdf

[14.] Lord, J., *Facilitating the Application Development Process Using the IBM Patterns for e-Business*, 2001, International Business Machines (IBM), Somers, NY http://www-106.ibm.com/developerworks/patterns/guidelines/lord.pdf

[15.] Bloor, R. and Hanrahan, M., *Patterns of Experience*, (2001), International Business Machines (IBM), Somers, NY, http://www-106.ibm.com/developerworks/patterns/guidelines/bloor.pdf

[16.] Adams, J., Koushik, S., Vasudeva, G., and Galambos, G., *Patterns for e-business - A Strategy for Reuse*. (2001), Somers, NY, IBM Press

[17.] Fowler, M., *Analysis Patterns - Reusable Object Models*. (1997), Menlo Park, CA, Addison Wesley Longman

[18.] Robertson, S., Requirements Patterns via Events/Use Cases, The Atlantic Systems Guild, *accessed*: 2003, http://www.systemsguild.com/GuildSite/SQR/Requirements_Patterns.html

[19.] Cybulski, J.: *Patterns in Software Requirements Reuse*. in *3rd Australian Conference on Requirements Engineering ACRE'98*. (1998), p 135-153, Deakin University, Geelong, Australia: Department of Information Systems, The University of Melbourne http://www.dis.unimelb.edu.au/staff/jacob/publications/awre-98%5Cpaper.pdf

[20.] Cybulski, J., Neal, R., and Allen, J., *Reuse of Early Life Cycle Artefacts.* Annals of Software Engineering, (1998). **5**: p. 227-251, Kluwer Academic

[21.] Adolph, S., Bramble, P., Cockburn, A., and Pols, A., *Patterns for Effective Use Cases,* Agile Software Development. (2002), Upper Saddle River, N.J., Addison Wesley Longman

[22.] Biddle, R., Noble, J., and Tempero, E.: *Patterns for Essential Use Cases*. in *Australasian Pattern Languages of Programming (KoalaPLoP)*. (2001), Melbourne, Australia

[23.] Gzara, L., Rieu, D., and Tollenaere, M., *Patterns Approach to Product Systems Engineering.* Requirements Engineering, (2000). **5**: p. 157-179, Springer-Verlag

[24.] Re, R., Braga, R., and Masiero, P.: *A Pattern Language for Online Auctions Management*. in *European Pattern Language of Programming (EuroPLoP)*. (2001), Irsee, Germany: EuroPLoP

[25.] Jackson, M., *Problem frames,* ACM Press. (2001), Harlow, Pearson Education Ltd., Addison Wesley

[26.] Sutcliffe, A., *Domain analysis for software reuse.* Journal of Systems and Software, (1998). **50**: p. 175-199, Elsevier Science