A Re-Examination of Alexander’s Theory and Process of Design for Software Development

Andenet Emiru
Nik R. Hassan, Ph.D.
Department of Finance and Management Information Sciences
University of Minnesota Duluth

ABSTRACT

The goal this research is to re-examine the current popularity of using software patterns to enhance the quality of software development within the software industry. This study reviews the software development landscape and suggests that because the software patterns movement is going farther away from Christopher Alexander’s original pattern language and design theories, the current software patterns movement will not significantly impact the software industry. This study proposes a combination of Alexander’s pattern language and current component-based development approaches in developing software called “component patterns.” Four systems analysts were interviewed to explore the viability of using components patterns in their development process. The results show that component patterns hold promise for improving the process of software design.

Introduction

The objective of this research is to re-examine the use of Christopher Alexander’s theory and process of design (Alexander, 1964, Alexander, 1979, Alexander, et al., 1977, Alexander, et al., 1987) as an approach for improving the effectiveness of systems development. Alexander’s theory of design has inspired the software industry and has generated a great deal of interest in the use of Pattern Languages for software development (Beck, et al., 1996, Schmidt, 1996, Schmidt, et al., 1996). This paper reviews the software development landscape and briefly analyzes from first principles the origins of various approaches, including the genesis of software patterns. This paper compares the theory presented by Alexander with the way in which it is used by the pattern community. Four systems analysts were interviewed to explore the viability of using components and patterns in their development process. The second part of the research explores the efficacy of what this research terms as “component software patterns” (or “component patterns”), an alternative to software patterns for software development. Component patterns are
A Re-Examination of Alexander’s Theory and Process of Design for Software Development

generic software patterns that exist as documentation at a higher level than abstract software patterns, physical data structures or algorithms. This research argues that component patterns are closer to what Alexander describes as the pattern language for architecture. In this part of the research, the communicative value of these component patterns was rated by the systems analysts. The expectation of this research is that component patterns hold promise to improving the process of software design.

Research Agenda

According to Christopher Alexander (1964), an artifact is well designed because of the result of years of actual application of forms to context and the ensuing adjustment and self-adaptation that has taken place to ensure fit. Alexander defines forms as the object of design (the artifact to be built) and defines context as the demands put on the artifact to function in its environment. Alexander defines his preferred process of design as the “unselfconscious” process of fitting form to context. The current software development approach follows a “self-conscious process” in which major changes in form are forced onto context and years of application experience are effectively ignored. Contrary to this, an “unselfconscious” approach of software design instead builds upon years of actual successful application of software code to user requirements rather than following the traditional multi-phased approach of analysis followed by designing according to the results of the analysis, and later implementing to fit the results of the design. The thesis of this research is that the unselfconscious process of software design is more effective because only small changes and adjustments are performed on existing patterns of code which have already worked to satisfy requirements. It is also the thesis of this research that this “unselfconscious” process of software development does not follow the traditional “build from scratch” approach, but starts from an existing, working physical software “base” and, that complex software programs are built on top of simpler already-working software components by adjusting these components to user requirements as they are constructed. In this way the single process of design combines the many phases of traditional software development of investigation, analysis, design and implementation, resulting in a single, more manageable process of fitting form to context. The agenda of this research is to re-examine the process of software development in the light of this theory of design and to explore the viability of combining both the rational multi-phased technological approach with the intuitive single-phased humanistic approach using software components.
Research Outline

To accomplish the above mentioned goals, this paper will cover the following topics. The research background section will briefly describe the major approaches in software development and their underlying theories. The section on Alexander’s design theory and software patterns explores in some detail the influence of Alexander’s theories on the software pattern community and critically examines the differences between current software pattern approaches and Alexander’s original theories. This section will also introduce the new concept of component patterns and the manner in which they fit within Alexander’s theories. The section on component patterns and software development explains the benefits of component patterns and the reasons they will work better than current approaches (including here the way in which phenomenology is the answer to the design process). The research design section explains the strategy for the survey, the instruments that will be used to perform the research and the process by which the results will be analyzed. Finally the research results section will summarize the results of the survey that was performed on 100 systems analysts and users.

Research Background

Design is defined as “the process of defining the architecture, components, interfaces, and other characteristics of a system or component” by the IEEE Computer Society. Creative design is defined by Taylor (1959) as “the process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realization.” Software design in the traditional sense is defined by Pressman (1997) as the iterative activity translating the requirements into a “blueprint” for constructing the software.

Before reviewing the process of software design, it is important to understand what constitutes design. McPhee (1996) explains in detail design theories and how they have affected software design. A brief summary of McPhee (1996) is provided. Designing a mathematical theorem can be as much a design process as building a house. McPhee lists general characteristics of design as (1) it starts with a need (a motivation for design) and requiring an intention, (2) involving some kind of transformation of a form that can be used to guide the implementation of an artifact, plan or process. Simon (1981) defines design as the restructuring of a current situation to achieve some preferred situation, (3) it involves a generation of new ideas (an element of creativity), (4) it satisfies a set of internal and external constraints derived from “functional and performance specifications of the artifact, limitations of the medium and process by
which the artifact is rendered or produced and aesthetic criteria on the form of the artifact” (Mostow, 1985). It is a problem solving and decision making process, (5) the results of which becomes a scheme for implementing an artifact (a blueprint), and (6) it always involves diversity and evolution (change).

McPhee divides software design theory into two major approaches, the formalists’ approach and the pragmatists’ approach. The formalists base their work on theoretical foundations of mathematics and computing science, while the pragmatists will use any other methods beyond the formal techniques as long as they work. Both approaches try to capture the complex conceptual constructs and somehow manage the transformation of these constructs into physical systems. Regardless of their inclinations, proponents of both approaches agree that software design is a human endeavor and designers will need to interact with users at some point in their processes to determine the suitability of their design.

McPhee categorizes general design theory into either the “scientific” approach to problem solving (natural science) or the social approach (human science) to problem solving. The first often involves a rational, logical and analytical mode of design, whereas the second is described as an artistic, intuitive and idiosyncratic approach to design. This conflict can be demonstrated by the decision of the “design science” movement to basically ignore the “mysterious” characteristic of design, and instead declare that the subjectivity of design must be overcome to make design “good.” In the first approach, design is assumed to be similar to natural sciences in that there is something to be found and discovered as the process of design proceeds. The criticisms against this approach are (1) that design is concerned with “how things ought to be” (Simon, 1981) whereas science is “concerned with how things are.” Consequently, the scientific method of design is less likely to result in a creative design because there is little collaboration between the skills of the designer and the vision of the customer; (2) since design cannot be described completely, they cannot be completely analyzed; and since thorough analysis is what science requires in order to be successful, the scientific approach to design cannot work; (3) the scientific, rational approach may result in a premature selection of courses of action, excluding better choices or alternative interpretations; and (4) design is a social activity involving communicative, cultural and political, (hence interpretive) approaches, so objective scientific approaches cannot work well for design. Additionally, the current tools of the scientific method of design (data flow diagrams, entity relationship diagrams, class diagrams, interaction diagrams) do not encourage any rich communication between designer and user because the level of abstraction of these models are beyond the typical user’s ability to visualize and conceptualize.
According to McPhee, in software design, the formalist/scientific approaches to design focus on the way in which software developers should proceed when engaged in the design of a software system, and typically specify a series of steps to be applied during design. Examples of this approach include the classic waterfall development process (Boehm, 1976), the spiral process model (Boehm, 1988), the System Development Life Cycle (Royce, 1970, Yourdon, 1982), Structured Systems Analysis and Design (DeMarco, 1979, Ross and Schuman, 1977), data decomposition (Jackson, 1975, Warnier, 1974) and other approaches that result in a set of hierarchical, discreet, temporal chunks of activities and deliverables. The “design” process itself, according to this formalistic approach, is placed as one of the many phases within the overall series of steps resulting in a deliverable for the next step. Despite its rigidity, there are many advantages to using this approach, most of which relate to the benefits of its “divide and conquer” characteristic; the organized, systematic and disciplined decomposition of the larger, complex problem; the ease of estimating the time to completion of individual tasks; the ease of measuring project progress; and other management and engineering-related benefits.

McPhee states that the humanistic and social approach emphasizes the economic, psychological, sociological, organizational, philosophical, political and aesthetic issues surrounding software design. This approach acknowledges the interpretive processes that take place as software is developed; the flexibility and adaptability required in the rapidly changing and unstable tasks, process and environments; and the recent changes overtaking the software industry, such as the demands placed by the ubiquity of computers in society. Other reasons for choosing the humanistic approach includes better handling of the development process itself. Because constantly changing requirements the nature of almost every software design project, the formalistic method is unable to handle the highly interactive, interleaved and loosely ordered tasks of software design. McPhee emphasizes that the humanistic approaches can better manage any imbalance that might surface during design, address inadequacies at different levels of abstraction, and better handle the globally un-deterministic activity that characterizes software development. The humanistic approaches acknowledge that as social beings, human designers cannot be objective as in natural sciences because they are themselves a part of the design process, contributing their idiosyncrasies to the process of design. The process of design is seen as an interpretive process where meaning is discovered and agreed upon among human agents, and where meaning is understood in the context in which it is interpreted. This context includes the anticipated meanings or preconceptions that are carried by the participant in the design process and the learning process that takes place as a result of understanding the goals,
A Re-Examination of Alexander’s Theory and Process of Design for Software Development

constraints and requirements. Other authors in the information systems field who have written on the humanistic approach to software development include Boland et al. (1982) Lytinen (1987) and Lee (1991). One of the goals of this research is to explore this interpretive process that takes place among participants in the software development activity.

Examples of such humanistic approaches in software design include the many exploratory approaches towards software design. McPhee states that often, these exploratory techniques are used within formalistic approaches to either supplement formal methods (Boehm, et al., 1984), or initiate the process of design; e.g., during the initial discovery of objects in object-oriented approaches. Examples of such techniques and models include prototyping (Boehm, et al., 1984, Naumann and Jenkins, 1982), Class-Responsibility-Collaboration cards (Beck and Cunningham, 1989) and use case diagrams (Jacobson, et al., 1992). Human-computer interaction (HCI) studies and graphical user interface (GUI) development, which focus on the more humanistic (cognitive and psychological) aspects of interaction, rely heavily on prototyping as a means of design and validation. Historically, the development of HCI and GUI owe their origins to another area in software development that was taking place at the same time as the humanistic approaches were being explored --- the development of object-oriented programming (OOP).

The concept of OOP was developed in Norway in the 1960s when Chrystian Nygaard and his associates developed the Simula language (Dahl and Nygaard, 1966, Nygaard, 1962, Nygaard and Dahl, 1963). This led to further research by Xerox at its Palo Alto Research Center in the 1970s, resulting in the development of Smalltalk-80 (Goldberg and Kay, 1976, Kay, 1977, Kay and Goldberg, 1977) and graphical-based interfaces (Johnson, et al., 1989) that later became interfaces for Microsoft Windows and Apple operating systems (Kay, 1987). OOP changed the process of design by introducing object-based concepts such as encapsulation, inheritance and polymorphism. Consequently software design could be viewed beyond low-level data structures and algorithms into higher level abstractions of objects, which could in turn be built into components and frameworks. With OOP, it was possible to design reusable components and frameworks that contained objects with properties and behaviors, sending messages to other objects, resulting in a more flexible and modular architecture. Since the late 1960s (McIlroy, 1968), software components was hailed as the solution to the software construction problem and OOP made it possible to build software from reusable components similar to how complex integrated circuits was constructed from simpler components (Cox, 1986).

In its early stages, component-based software development took the form of the development of frameworks of software modules. The First
International Workshop on Component-based Software Engineering defined a software component as a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces (Brown and Wallnau, 1998). Kozaczynski (1999) defined a component as: “A component is a part of a system that is a unit of design, construction, configuration management and substitution. A component conforms to and provides the realization of a set of interfaces in the context of well-formed system architecture.” The first widely used framework was the Smalltalk-80 user interface framework called the Model/View/Controller (MVC) framework (Johnson, 1997a). It divides the typical user interface into three kinds of components; models, views and controllers. These three components worked in concert to display views (the view component) of an application object (the model component) and allowed users to interact (the controller component) with the application object, using as many different views as required. Newer and more complicated frameworks such as the Macintosh framework, the Andrew Toolkit, InterViews, OpenStep and the Microsoft Foundation Class (MFC) were built from this original framework. Each framework comes with a component library that contains concrete subclasses of the classes in the framework. Frameworks are, therefore, a kind of domain-specific architecture that provides a reusable context for components (Johnson, 1997b). Unfortunately, the progress of frameworks and component-based software development did not go far beyond user interface components (Nierstrasz and Meijler, 1995) and only recently has interest been shown towards combining business concepts with software components (Fingar, 2000, Herzum and Sims, 2000, Kozaczynski, 1996). Research in component software has expanded into various different abstractions of components, from the repositories of commercial-off-the-shelf (COTS) components that are being researched at the Software Engineering Institute (SEI) at Carnegie Mellon, to the concept of the factory of business components by Herzum et al. (2000) to the very abstract business patterns being developed by IBM researchers (Adams, et al., 2001).

This evolution from formalistic approaches to humanistic approaches does not only underscore a difference in how the process of design in undertaken, but also reflects another aspect of the evolution in the process of design. Formalistic approaches define the process by which design should take place, but not the product of the design (the software program itself). This is reflective of the Cartesian ideal of separating the subject from the object that permeates the natural sciences and the “scientific” method. Humanistic approaches appear to not only define the process, but also propose the product. For example, exploratory approaches such as the CRC cards technique actually combine the process
of analysis with synthesis. The cards become the objects. This aspect of combining the process of design with the product of design is also reflected in another development in the evolution of software design – the concept of software patterns (Gamma, et al., 1995).

**Alexander’s Design Theory and Software Patterns**

Inspired by Christopher Alexander’s work on architectural designs, the so-called Gang of Four (GoF) wrote the first book on software patterns entitled Design Patterns (1995). They defined software patterns as: “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” (p. 3).

Design patterns are described by the GoF in the following paragraphs: “One thing expert designers know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. Whey they find a good solution, they use it again and again.” (p. 1) “Design patterns make it easier to reuse successful designs and architectures…Put simply, design patterns help a designer get a design “right” faster….the design patterns in it capture only a fraction of what an expert might know…It doesn’t have any application domain-specific patterns.” (p. 2) Patterns are, therefore, essentially the product of design, because patterns define the form by which a certain problem (the context) is solved. Although patterns suggest a blurring of the process of design with the product, there are still two very critical implications to the statements made by the GoF. First, these statements suggest that software patterns are still abstract solutions and; unlike software components defined earlier, are not physical implementations (although example code of such patterns is offered by most authors of software patterns). The idea behind using patterns is to offer designers different ways in which to solve problems they face when designing and constructing systems based on the experience of others. Second, the software design patterns as defined by the GoF are meant solely for designers to improve their design process as soon as requirements for the design are made clear. The problem has been defined and they suggest that these patterns may be useful during the “design phase” of the software development life cycle, to encourage code reuse or as a communicative tool between designers. Although software design patterns as described by the GoF to a certain extent assist in the process of finding fit between forms (the software) and context (albeit more generic requirements), they do not address the process of uncovering user requirements.

These critical implications suggest major differences between software patterns as defined by the GoF and the Pattern Language made popular by Alexander. First, the patterns proposed by Alexander are
essentially concrete, physical implementations, albeit couched in generic terms. All of the patterns proposed by Alexander are far from being abstract. Consequently, they are easily understood by any layman who wishes to build dwellings and structures. In fact, it was Alexander’s intent to democratize the knowledge of architecture for the layman so that any one can build a home following tested architectural principles. Second, because Alexander’s patterns are written in a language that is understandable by both the architects and their customers, it is possible for both “designer” and “user” to have a productive discussion and presumably come up with creative ideas during the process of design. If it could be assumed that the objective of the process of design is ultimately to satisfy the customer, the language used by both of them needs to communicate all the critical elements of the design and also describe to the customers what they will eventually get in the end. Alexander defines patterns as “the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.” Patterns are the recurring solutions to the problems of design. People learn patterns by seeing them and recall them when need be without a lot of effort. Patterns link together in the mind, so that one pattern leads to another and another until familiar problems are solved. Alexander assumes that physical clarity cannot be achieved in a form until there is first some programmatic clarity in the designer’s mind and actions and for this to be possible, the designer must first trace his design problem to its earliest functional origins and be able to find some sort of pattern in them. Alexander’s Pattern Language communicated exactly to anyone how best to build and construct, and opened up the knowledge of architecture to the layman. With current changes in the software industry and the ubiquity of computers, a similar series of patterns for software development should be developed so that not only the developer should benefit, but more so the beneficiary of the software, i.e., the users.

Now that the difference between Alexander’s patterns and the GoF patterns are made clear, a difference in the theory of design proposed by Alexander and current developments in software development needs to be examined. In his Synthesis of Form, Alexander explained his theory of design. He describes form as the ultimate object of design (p. 15) and that design has to do with the goodness of fit (a clearly humanistic aspect of his design theory since he is saying that one knows its right when one feels or sees fit). He defines design as the effort to achieve fitness between form (solution to the problem) and context (that which defines the problem) (p. 15-16). “The form is a part of the world over which we have control, and which we decide to shape while leaving the rest of the world as it is. The context is that part of the world which puts demands on the form; anything in the work that makes demands of the form is context. Fitness is a relation of mutual acceptability between these two”. (p. 18-19) He stresses that
since people cannot give an adequate description of the context (an inherent quality of almost all software development projects), designers will-instead of actually trying the form out in the real world context (trial and error)-they replace it with a symbolic method, because trial and error is too slow and too expensive (another parallel with traditional software development processes). According to this definition, the process of uncovering requirements in software development should be defined as part of the process of design since, as Alexander notes, users cannot be expected to give a full description of the context; because if they could do this, there would be no problems of design.

“Design is a problem because we are trying to make a diagram for something we do not clearly understand.” (p. 21). “The task of design is NOT to create a form which meets certain conditions, but to create such an order in the ensemble that all misfits are resolved …The form is the ensemble part which we have control over. It is only through the form that we can create order in the ensemble.” (p. 27) Good fit happens as a result of years of actual application of forms to context and the ensuing adjustment that has taken place to ensure fit. Alexander refers to this as the “unselfconscious culture.” “To achieve in a few hours at the drawing board what once took centuries of adaptation and development, to invent a form suddenly which clearly fits its context – the extent of the invention necessary is beyond the average designer. A man who sets out to achieve this adaptation in a single leap is not unlike the child who shakes his glass-topped puzzle fretfully, expecting at one shake to arrange the bits inside correctly…His chances of success are small because the number of factors which must fall simultaneously into place is so enormous.” (p. 59)

Most view this approach as a “scientific” approach (McPhee, 1996) because it reflects the Popperian falsification approach in which misfits are eliminated, resulting finally in an overall fit within the ensemble. “We should find it almost impossible to characterize a house which fits its context. Yet it is the easiest thing in the world to name the specific kinds of misfits which prevent good fit. A kitchen which is hard to clean, no place to park my car, the child playing where it can be run down by someone else’s car, rainwater coming in…(p. 23) However, there is little “science” in the way he describes the process of discovering that fit, which is really a cognitive and humanistic process. “The only reason we are able to match one thing with another at all is that we rely on a great deal of unexpressed information contained in the statement of the task, and take a great deal for granted.” (p. 25)

It is this paper’s opinion that Alexander has successfully combined both rational and humanistic approaches in his theory of design. This is not unlike the Feyerabend’s relativism (Feyerabend, 1978, Feyerabend, 1987), where disciplinary knowledge claims are viewed as contingent upon the
particular beliefs, values, standards, methods, and cognitive aims of its practitioners. As proposed by Alexander, the process of software design should not be separated into “analysis” and “design,” rather that design is cognitively a single process where an “analysis” of the object to be designed is a realization that the form (what we will call the object design pattern) does not fit the context (or requirement) of the design and therefore requires a change (the result of the design process) to what has already worked. The process of software design therefore is a process of building upon previous designs and components that fit the context. Changes in the design are necessary only when there is a change in context, which should normally result in minimal change to the form.

At first glance this conclusion appears to agree with the opinions of those that support component-based software engineering. However, there are several major differences. First, component-based software engineering deals with concrete components of software that work in the context of well-formed system architecture. It is the objective of this group of software developers to produce software components that can be plugged into existing architectures or frameworks such as COM, CORBA, .NET or J2EE (Brown, 1998, Seacord, 1999, Wallnau, 1999). Hence, there will always be a constant struggle with and dependence on framework compatibility and standards (Arsanjani, 2002), which will not help with the progress of the field (this is clearly illustrated by Microsoft’s decision to “abandon” COM/DCOM in favor of XML and .NET). Second, the component-based community continues to use traditional rational, multi-phased methodologies that imprison software components in the “vicious” circle of analysis paralysis (Crnkovic, et al., 2002, Kuzmanov, 1999). In contrast, this research proposes an approach that will build complex systems by plugging designs (rather than components) into existing working systems, using component patterns rather than physical software components. Ultimately, each design will need to be coded into a physical binary component, however, because the pattern already addresses the context and function it was designed for, the process of coding becomes much easier and faster. These patterns are termed “component patterns” because they bridge the gap between the abstract pattern of the GoF and the physical components of component-based software engineering.

**Component Patterns and Software Development**

This section will explain why using component patterns in software development will dramatically improve the process of design. Most of the arguments for component patterns are based on the general theory of design. For explanatory purposes, an elementary set of component patterns for a simple catalog-type application is proposed in Table 1. The first
column describes the specific business function that the component pattern is performing. The second column describes the physical binary component that will be implemented for that business function. The third column provides the name of the component pattern that is used to implement the binary component, and the last column provides a description of the component pattern. For example, in order to manage items that will be displayed by the catalog, at least five components need to be implemented, the “Product database” component, the “Product” component, the “Categories database” component, the “Categories” component and the “Product Manager” component. The “Product” component is implemented using the “Selling Item” component pattern. The “Selling Item” component pattern describes various alternative patterns for managing different kinds of items that are to be offered for sale. In a situation where products are being purchased as opposed to being sold, perhaps a “Purchasing Item” component pattern can be used to implement the “Products” components that are to be purchased. Once these product-related components are implemented successfully, the “Catalog” related components can be implemented to display products online. Later, if the business decides to implement an auction-based system, both components, the “Product” and “Catalog,” form the basis for building a more complex application involving, say, a “Bidding Engine” that is implemented using a “Bidding Transaction” component pattern. The process of design proceeds in this manner, building, on top of existing components, other components that add more functionality to the whole system. There should be no reason to go back and modify the Product or Catalog components because they have worked successfully in the context of meeting requirements. Should there be any change to the context that requires a modification of the original base component, such a modification is expected to be minimal.
### Table 1: Simple Catalog Component Patterns

*Component Patterns Guide the Transformation of Form*

The theory of design suggests that design involves some kind of transformation of a form that can be used to guide the implementation of an artifact. Software components are actual binary implementations of an artifact, rather than a guide, and are therefore inextricably bound to the framework of implementation. Consequently, the chore of transplanting one component from one framework to another no longer becomes a trivial matter because of the many dependencies that are formed as a result of the

<table>
<thead>
<tr>
<th>Business Function</th>
<th>Component Name</th>
<th>Component Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage Items</td>
<td>Product database</td>
<td>Table for simple non-assembled selling item</td>
<td>Contains various database patterns for keeping simple products or parts to be sold. Performance: Medium</td>
</tr>
<tr>
<td></td>
<td>Product</td>
<td>Simple non-assembled selling item</td>
<td>Contains patterns to manage Table for simple non-assembled selling item</td>
</tr>
<tr>
<td></td>
<td>Categories database</td>
<td>Item Organizer Table</td>
<td>Contains patterns that manage the Item Organizer Table</td>
</tr>
<tr>
<td></td>
<td>Product Categories</td>
<td>Item Organizer</td>
<td>Contains patterns that manage both Selling Item and Categories table</td>
</tr>
<tr>
<td></td>
<td>Product Manager</td>
<td>Item Manager</td>
<td>Receives input to create, edit and delete items from table. Performs checks to make sure creation, edition and deletion is allowed</td>
</tr>
<tr>
<td>Search/Browse electronic catalog</td>
<td>Search engine</td>
<td>Search form</td>
<td>Provides an interface for inputting search words</td>
</tr>
<tr>
<td></td>
<td>Search engine</td>
<td>Search result</td>
<td>Provides an interface for displaying search results</td>
</tr>
<tr>
<td></td>
<td>Search method</td>
<td>Search Pattern A – Simple text search</td>
<td>Searches Item Table for search words using a specific search algorithm</td>
</tr>
<tr>
<td></td>
<td>Search method</td>
<td>Search Pattern B – Performs tree search based on categories</td>
<td>Searches Item Table using a different search algorithm</td>
</tr>
<tr>
<td></td>
<td>Catalog</td>
<td>Selling Display Templates</td>
<td>Displays how the catalog will look like and where graphics and data fields are placed</td>
</tr>
<tr>
<td></td>
<td>Catalog</td>
<td>Selling Display Processor</td>
<td>Performs the logic for displaying categories and items on the Catalog Templates</td>
</tr>
<tr>
<td></td>
<td>Catalog Manager</td>
<td>Selling Metadata Manager</td>
<td>Performs the logic to manage and edit the categories and tree structure – Catalog Categories table</td>
</tr>
</tbody>
</table>
implementation of that component within a particular framework. The longer a component stays within a framework, the more likely it is to become dependent on that framework. Software patterns offer a solution at a slightly higher level of abstraction (closer to data structures and algorithms) and cannot be used directly in any particular business context. Hence, the task of matching software patterns to business functions becomes non-trivial, and the designer is often left alone to figure out which software components are most appropriate for any particular business function. Business patterns such as those proposed by the IBM researchers (Adams, et al., 2001) are at too high an abstraction to be of immediate use in code production. Component patterns exist somewhere in between. They are essentially patterns and therefore can act as a guide to implementation, but at the same time, because they are specific enough to the context of their implementation (e.g., Selling Item), the designer is no longer left alone to figure out whether they are suitable for any business problem. With the help of users, designers will be able to select those that are most suited to the context, and programmers can more easily code them into binary components.

**Component Patterns Support Generation of New Ideas**

A major advantage of component patterns as opposed to software patterns is their communicative value. The “pattern language” that is exhibited by component patterns is much closer to what Alexander proposed in his Pattern Language. Component patterns share the same ideal that Alexander had when he developed his Pattern Language --- to enable the layman understand architectural principles for building dwellings and structures. Similarly the naming of component patterns should be in a language that encourages users to communicate more easily with designers during the process of design. The language should enable the designers to explain to users all the advantages, disadvantages, limitations and boundaries each component pattern offers to the application and hence to the business function. Unlike current software modeling methods, there should not be any need for users to attend a week-long course just to understand data flow or UML diagrams. Although these models are indispensable in the process of design, they will not necessarily become the standard language of communication between designer and users. Instead the language to be used will be component patterns.

As soon as communication between designers and users is enhanced, this will lead to a tremendous deluge of creativity and generation of new ideas in the process of design will occur. The implementation of new business-related ideas can be represented using a combination of several component patterns and new component patterns may be created in
the ensuing process. Designers will be spared from the “vicious” cycle of analysis paralysis, because practical business considerations can play a larger part in the process of design. The MIT Process Handbook Project is working in this direction, cataloguing the many thousands of possible patterns of business processes in an effort to generate new ideas for implementing innovative business structures.

Component Patterns Satisfy Internal and External Constraints

The design process should satisfy internal and external constraints derived from functional and performance specifications of the artifact. The classic example provided by Alexander (1964) was the design of the kettle. Centuries of adaptation and implementation to actual context have resulted in the simple but elegant and very functional design of the kettle. Internally the kettle should be made from material that is heat resistant and malleable, but must be cheap enough to allow mass production. Externally, the user should be able to add water to it easily, heat the water by placing the kettle over a stove, carry the kettle easily in one hand and be able to pour the water out without too much trouble. These constraints resulted in a form having a handle and a spout. Even when external constraints such as the invention of electricity took place, and it was necessary to add another interface to the kettle to allow a heating element to heat the water, none of the other internal and external designs needed to be changed. The electric kettle is essentially the same design as the centuries-old wood stove kettle. The component pattern that was used to implement the kettle could be called the “Kettle” component pattern and only minor changes to that component pattern may be necessary when designing say a physical “Whistling Kettle” component. That same “Kettle” component pattern has satisfied all internal and external constraints and requirements to implement its function.

The same cannot be said about software components. Software components satisfy only constraints specific to its framework. Similarly, software patterns satisfy at best one or two problems and do not go as far as to address the business context. To do this, several software patterns may be required, and the ensuing combination may still suffer from additional unanticipated problems because the combined pattern has not undergone the test of time. If it did and successfully passed that test, it would be called a component pattern.

Component Patterns Solve Whole Business Problems

Unlike software components and software patterns, component patterns solve whole units of business problems. For example, in order to
solve the problem of tracking information on products to be sold, the “Selling Item” component pattern has been tested to handle every conceivable business function of keeping information on such products. However, if the item to be sold was an assembly of parts that required some form of configuration and manufacturing in order to sell it, a slightly different version of that pattern, perhaps called the “Selling Composite Item” component pattern, will need to be used. The business function of tracking such an item is fulfilled by that component pattern. Software patterns do not address whole business problems, only programming problems. Recent work on business components by Herzum and Sims (Herzum and Sims, 2000) is addressing this need.

Component Patterns are Flexible to Changes

The major problem with software components is change. Any addition or changes to the surrounding framework affects the portability of any software component. Software patterns and components share the same quality of not being subject or dependent on changes to either technology or to the business environment. Because they are abstract guides to implementation, they thrive on change and will grow with newfound technology.

Preliminary Research

A simple instrument was developed to test the viability of using component patterns in comparison with traditional data flow diagrams and entity relationship diagrams. The questionnaire is provided in the Appendix. Four systems analysts were interviewed using the questionnaire. The ratings of the items in the questionnaire were averaged and the average rating between the diagrammatic presentation and the component patterns were compared. Several questions that were not documented in the questionnaire were raised by the system analysts and a discussion ensued on the general idea of a component pattern. The results of the preliminary research are described in the following sub-sections.

Better Performance with Component Patterns

Out of the four analysts interviewed, three of them concluded that the description provided by the component patterns were much easier to understand and represented a more complete description of the requirements. Although one system analyst felt the diagram was more understandable, there was only a slight difference in the average points given by that system analyst.
Difficulty in Understanding Component Patterns

There was a general difficulty for the systems analysts to understand the concept of component patterns. This is partly due to the different terminologies used by different analysts in describing the same objects. One analyst commented that the component patterns the research was referring to was similar to their company’s software frameworks.

Difference between Component Patterns and the Traditional Development Process

One system analyst raised the issue of the difference between the proposed component patterns method and their company’s development process. The system analyst claimed that they are already reusing their software components and refitting the components to new customers’ requirements. He also claims that because their development group understands their customers’ needs, there is no significant gap of understanding between the technical group and the customers.

Choice of methodologies is irrelevant

Another system analyst commented that the choice of methodologies is not significant in the development process. What is important is making sure the customers get what they want regardless of the methodology or diagramming method used. What is really significant is the quality of the people performing the development process. The people need to listen to customers, understand the business needs and not just the technical needs of the customers.

Component Patterns May Not be Sustainable

One important issue with component patterns, or any general patterns for that matter, concerns the financial sustainability of the business model surrounding patterns.

Most companies will not want to share their patterns with any other company and even if they did, they still need to figure out how to make sustainable profits from generic patterns. The model of sharing patterns could probably work within a large organization, but not necessary among different independent firms.
Conclusion and Further Research

Based on the preliminary research performed, we can conclude that component patterns do provide some value to systems analysts at least compared to traditional diagramming methods. Further research need to be performed to develop a better representation method for component patterns that describe not only the components but also the process of development. Further research also needs to be performed to clearly define the value of component patterns and the development process most suitable for their development and implementation in organizations.
Appendix

Software Development Questionnaire

The objective of this study is to explore the viability of an alternative approach to software development that will enhance the quality of the software development process. There are two sections to this questionnaire. The questionnaire should be answered by a technical person in charge of software development (e.g. Head of Development or Senior Systems Analyst).

1) Name of Company: __________________________________________

2) Company Size: _______ employees

3) Business area: (Circle one) Education, Manufacturing, Services

4) Number of information systems staff _________ employees

Assume that you are planning to develop an online catalog for your new business. The catalog should be able to display, search and browse information about all the products you are selling and display the products in convenient categories for the buyer. Read the following diagrammatic and textual descriptions. Rate how well each description represent the complete set of specifications for the online catalog system based on.
Using Data Flow and Entity Relationship Diagrams

Please rate the above diagram according to the following criteria:
<table>
<thead>
<tr>
<th>CRITERIA</th>
<th>RATING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strongly Disagree</td>
</tr>
<tr>
<td>1. The language of the model is easily understood</td>
<td>1    2  3  4  5</td>
</tr>
<tr>
<td>2. The model correctly represents the requirements</td>
<td>1    2  3  4  5</td>
</tr>
<tr>
<td>3. The model completely represents the requirements</td>
<td>1    2  3  4  5</td>
</tr>
<tr>
<td>4. The purpose of the model is clear to me</td>
<td>1    2  3  4  5</td>
</tr>
<tr>
<td>5. The model communicates the details of the requirements</td>
<td>1    2  3  4  5</td>
</tr>
<tr>
<td>6. I can explain the model to others easily</td>
<td>1    2  3  4  5</td>
</tr>
</tbody>
</table>
Using Textual Descriptions of Components

<table>
<thead>
<tr>
<th>Business Functions</th>
<th>Component Name</th>
<th>Component Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage Product</td>
<td>Product database</td>
<td>Table for simple non-assembled selling item</td>
<td>Contains database patterns for keeping simple products or parts to be sold.</td>
</tr>
<tr>
<td></td>
<td>Product</td>
<td>Simple non-assembled selling item</td>
<td>Contains code patterns to create, edit and delete items from the database “table for simple non-assembled selling items”. Performs checks to make sure creation, edition and deletion is allowed.</td>
</tr>
<tr>
<td></td>
<td>Categories</td>
<td>Item organizer table</td>
<td>Contains database patterns that keep the categories or other meta-descriptions of items. Each item can be part of any category and a category will hold many items</td>
</tr>
<tr>
<td></td>
<td>Product Categories</td>
<td>Item Organizer</td>
<td>Contains code patterns that create, edit and delete the information from the database “item organizer table”. Performs consistency checks to make sure that no creation or addition anomaly occurs.</td>
</tr>
<tr>
<td>Search/Browse electronic catalog</td>
<td>(Part of ) Search engine</td>
<td>Search form</td>
<td>Provides an interface for inputting search words</td>
</tr>
<tr>
<td></td>
<td>(Part of ) Search engine</td>
<td>Search result</td>
<td>Provides an interface for displaying search results</td>
</tr>
<tr>
<td></td>
<td>(Part of search engine) Search method</td>
<td>Search Pattern A – Simple text search</td>
<td>Searches Item Table for search words using a specific search algorithm</td>
</tr>
<tr>
<td></td>
<td>(Part of search engine) Search method</td>
<td>Search Pattern B – Performs tree search based on categories</td>
<td>Searches Item Table using a different search algorithm</td>
</tr>
<tr>
<td>Catalog</td>
<td>Selling Display Templates</td>
<td>Displays how the catalog will look like and where graphics and data fields are placed</td>
<td></td>
</tr>
<tr>
<td>Catalog</td>
<td>Selling Display Processor</td>
<td>Performs the logic for displaying categories and items on the Catalog Templates</td>
<td></td>
</tr>
</tbody>
</table>

Please rate the above diagram according to the following criteria:
<table>
<thead>
<tr>
<th>CRITERIA</th>
<th>RATING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strongly Disagree</td>
</tr>
<tr>
<td>7. The language of the model is easily understood</td>
<td>1   2   3   4   5</td>
</tr>
<tr>
<td>8. The model correctly represents the requirements</td>
<td>1   2   3   4   5</td>
</tr>
<tr>
<td>9. The model completely represents the requirements</td>
<td>1   2   3   4   5</td>
</tr>
<tr>
<td>10. The purpose of the model is clear to me</td>
<td>1   2   3   4   5</td>
</tr>
<tr>
<td>11. The model communicates the details of the requirements</td>
<td>1   2   3   4   5</td>
</tr>
<tr>
<td>12. I can explain the model to others easily</td>
<td>1   2   3   4   5</td>
</tr>
</tbody>
</table>
Works Cited


A Re-Examination of Alexander’s Theory and Process of Design for Software Development


