Moral Design: Patterns in Software Design

David Hutchful

Calvin College

dhutch97@calvin.edu

ABSTRACT

Different methodologies have been proposed and implemented to improve the efficiency of the software design process. The concept of a Pattern Language is one such attempt. Finding its roots in the work of architect Christopher Alexander, and further popularized by the first Design Patterns book by Gamma et al. (affectionately known as the 'Gang of Four') [5], patterns have become an integral part of the software development process. However, as Alexander emphasizes in his keynote address at OOPSLA 1996, there is a moral aspect to the concept of Pattern Language that the software community has largely ignored [3]. In this paper, I will explore this moral aspect of patterns. I will argue that software design can truly fulfill this "moral imperative to build whole systems that contribute powerfully to the quality of life," [3] and I will show how software designers can produce these moral systems.

Keywords

Pattern, pattern language, moral design, moral challenge, *Alexander end*, *object-oriented end*, software design, architecture.

1. INTRODUCTION

R. Clayton, a moderator at the Georgia Tech software engineering seminar on patterns, provides a simple model for understanding how software engineers perceive patterns [3]. He uses two

definitions of patterns taken from the Webster's Ninth New Collegiate Dictionary: first, a pattern is "a form or model proposed for imitation" and second, a pattern is "a discernible coherent system based on the intended interrelationship of component parts." Clayton sees these two definitions as the pole ends of the "spectrum of views" about software design patterns. He calls the "form or model" end the *object-oriented end* and the "discernable coherent system" the *Alexander end*. In the following paragraphs, I will first examine and describe the *Alexander end* of the spectrum by looking at Christopher Alexander's understanding of patterns in his own field of Architecture. Second, I will examine the meaning of patterns in the *object-oriented end* of the spectrum and lastly, I will propose a modification to Clayton's linear model, which will then serve as the foundation for my discussion of moral design, and what it entails.

2. THE *Alexander end* PATTERN

Patterns have always been an integral part of architecture. Greek, Roman and other civilizations modeled their architectural works on patterns they found in nature. It is fact that some South American architectures were designed, from patterns in nature, to be extensions of nature that served as a point of contact between the physical and spiritual realms.[i] However, in the post world war two era, architects began to frown on the use of patterns: the dominant argument being that patterns hindered creativity and diversity in design.

Despite the unvoiced disapproval of patterns in the architectural community, Christopher Alexander and his colleagues, in their two books, The Timeless Way of Building and A Pattern Language [1, 2], introduced a novel way of thinking about patterns and using patterns in architecture. Alexander's goal was to produce living structure in the world: that is, "towns,

streets, buildings, rooms, gardens, places which are in themselves living or alive," and by their very nature, promoted a nurturing environment [3]. Alexander based his approach to this goal on a very abstract set of ideas,[ii] which I will now attempt to describe.

First, Alexander believes there is a timeless way of building, which he defines as an ordering process that only occurs "of its own accord, if we let it." [2] To discover this timeless way, he says we must first know the quality without a name. He defines this *quality without a name* as follows:

> There is a central quality which is the root criterion of life and spirit in a man, a town, a building, or a wilderness. This quality is objective and precise, but it cannot be named…The search which we make for this quality, in our own lives, is the central search of any person, and the crux of any individual person's story. It is the search for those moments and situations when we are most alive.[2]

Secondly, Alexander postulates that to define the central quality one must recognize that everything consists of patterns, which are in themselves either alive or dead. Thus, he writes, "each building and each town is ultimately made out of patterns in space…to the extent they [the patterns] are alive, they let our inner forces loose, and set us free; but when they are dead, they keep us locked in inner conflict." [2] He goes on to describe how a room in its entirety becomes more alive when more living patterns are present in it. The room eventually acquires a fiery glow, which enables it to participate in nature: "like ocean waves, or blades of grass, its parts are governed by the endless play of repetition and variety created in the presence of the fact that all things pass. This is the quality itself." [2(ix-xi)]

Thus, for Alexander's project, patterns are the basic building elements from which everything (buildings, towns, gardens, etc) comes to be. More importantly, patterns and pattern languages (connective rules for patterns) provide information on which physical structures and configuration of structures in space make the environment nurturing for human beings. In addition, and equally important, Alexander believed that one characteristic of a good environment is the high adaptability of each part of the environment to its particularities [3]. To achieve this feature, and scalability, in his use of patterns in architecture, Alexander employed a biological model: just as genetic code indirectly generates biological organisms (adapting them to their environment), Alexander planned to use, indirectly, shared pattern languages to develop environments, which effectively generated themselves.[iii] Therefore, for Alexander and his friends, patterns have a twofold function: on the one hand, patterns identify structural features that promote a positive and nurturing environment, and on the other, patterns provide an indirect way for environments to generate themselves. This is the *Alexander end* of the "spectrum of views" on patterns.

3. THE *object-oriented* PATTERN

History can retell several stories about the radical altering of methodology in a particular discipline due to the infusion of knowledge from another, often unrelated, field of study. This was the case with Software Design (SD). Although, the introduction of the concept of patterns, from Architecture, did not radically change SD methodologies, it began a strong movement of pattern developers and users who now dominate the SD community. The difficulty of designing reusable object-oriented software contributed greatly to the success of patterns in SD. As Gamma et al noted, "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. When they find a good solution, they use it again and again." [5] Patterns provided an efficient and systematic way of documenting these recurring solutions – "they help[ed] designers reuse successful designs by basing new designs on prior experience."[5] Thus, for software designers, a pattern is a recurring solution to a recurring problem in object-oriented software design.

Describing the nature of patterns in SD, John Vlissides, the fourth member of the Gang of Four, points out that design patterns "do not describe new or novel designs; they are *mined* from successful object-oriented designs. To constitute a pattern, a problem must recur, and the same solution must have been discovered independently and applied repeatedly in real systems" [6]. Furthermore, he identified four main functions or benefits for using patterns. First, patterns provide the capacity for design reuse, which Vlissides considers more powerful than code reuse. Second, a common language for design develops as each pattern name contributes to the vocabulary of that language. Third, referring to the patterns in documentation makes it easier for others to understand what you did and why. Lastly, Vlissides argues, "patterns help you restructure your system whether or not you used patterns up-front. Systems designed with patterns make it easy to transform an application of one pattern to an application of another. Systems designed without patterns in mind can use patterns as targets for class refactorings."

Software designers have centered most of their efforts on documenting patterns: Gamma et al, in producing the most definitive work on patterns in SD, stated that their goal was to "capture design experience in a form that people can use effectively."[3] Ultimately, SD has been extremely successful in this endeavor: software designers have discovered several patterns, and

have successful implemented these patterns in various software projects. This is the *Object-oriented end* of the "spectrum of views" on patterns.

4. EXAMPLES OF PATTERNS AND PATTERN FORMATS

Having given a brief summary of the origination and understanding of patterns in both the Architecture and Software Design fields, I will now complement this summary with a concrete example from each discipline and a description of the format used to document the patterns found in each discipline.

4.1.1 Format of an *Alexander end* pattern

Alexander, in his book Pattern Language, provides a practical language scheme that one can use to produce living structure. In addition, he provides a guide for using the pattern language. This guide is implicit in the format Alexander uses to present his patterns. The format itself is as follows: first, there is a picture, which represents an archetypal example of the pattern. Second, an introductory paragraph explains the context of the pattern, that is, how the particular pattern completes other larger patterns. Following this are three diamonds that indicate the beginning of the definition of the problem. The fifth section provides a detailed explanation of the problem and empirical evidence that supports the explanation.

Next, in bold, is what Alexander calls the "heart of the pattern": the solution to the problem described in the problem-definition section. After that, three more diamonds marking the end of the main body follow a diagram depicting the solution. The ninth and last section connects the pattern to other smaller patterns in the language that complete the pattern in question. The format

serves two purposes: first, it shows how each pattern connects to other patterns – this allows the user to see all 253 patterns as constituting one language. Second, it presents the problem and solution for which a pattern exists so that the user can modify the pattern without losing the essence of the pattern [1].

4.1.2 Example of an *Alexander end* pattern

See appendix A for an example of Alexander's patterns. It is worthy to note that Alexander's patterns are in themselves made up of other patterns. Thus, the example given both consists of various other patterns and it contributes to a larger pattern.

4.2.1 Format of an *object-oriented end* pattern

Gamma et al provided the first consistent format for documenting design patterns in SD [5]. The format has thirteen sections – together, they define the pattern name, the problem in question, the solution to the problem and the consequences that arise from the use of the pattern. See appendix B for a description of the template used to document patterns in SD.

4.2.2 Organizing *object-oriented end* patterns

Gamma et al, recognizing the possibility that the number of patterns might increase, developed a system for classifying and cataloging patterns [5(9-11)]. They classified patterns using two criteria: first, the purpose of the pattern and second, the scope of the pattern. The former refers to what a pattern does – Gamma et al identified three main purposes: creational, structural and behavioral. The latter criterion indicates whether a pattern is primarily concerned with relationships between classes or relationships between objects (see *figure 1* below). Gamma et al

acknowledge that there are several ways to organize patterns and encourage others to develop such systems since they believe that that will "deepen [one's] insight into what they [patterns] do, how they compare, and when to apply them" [5].

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Façade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Figure 1 [5] (the numbers in parenthesis refer to the page in [5] where the pattern can be found)

4.2.3 Examples of *object-oriented end* patterns

As *figure 1* indicates, there are three main purposes for *object-oriented end* patterns. I will examine one pattern example from each purpose. Note that all examples and descriptions come from Gamma et al [5]. First, Singleton is a creational pattern (that is, deals with the creation of objects) that "ensures a class only has one instance, and provides [a] global point of access to [that class]". Second, the Façade pattern is a structural pattern (deals with the composition of classes or objects) that "defines a higher-level interface that makes the subsystem easier to use." Lastly, the Memento is a behavioral pattern, which provides, "without violating encapsulation, [a way to] capture and externalize an object's internal state so that the object can be restored to this state later." Behavioral patterns represent the ways in which objects or classes interact. All three types of patterns are documented in the manner described in appendix B.

5. THE MORAL CHALLENGE

OOPSLA'96 was both the culmination of years of work done on patterns in the software design community, and the first official intersection of Architecture and Software Design. In my opinion, it was also a turning point where the community of software designers failed to turn. Christopher Alexander expressed his dismay at the inadequacy of Architecture to effectively, and in a timely manner, provide the kind of nurturing environment he had envisioned. He then gave the software design community a glimpse of the potential and the resources they possess that allows them to make a moral impact. Rather than accept the moral baton, the software design community stuck to their guns and continued to practice their one-sided use of patterns, albeit, rather successfully. However, I would now like to return to the turning point and pick up the baton.

Alexander made an insightful observation, the present realization of which should serve as a motivating force for the software design community to accept their moral responsibility: "I am convinced that the equivalent of the genes that act in organisms [for generative purposes] will have to be – or at least can be – software packages, acting in society" [3]. With the current explosion in computer use and the Internet revolution, with features like .NET, I think it is time software designers realized the social responsibilities they have acquired due to the pervasive nature of their products. Back in 1990, Mitchell Kapor issued his *Software Design Manifesto* calling on the software development community to improve on the design and usability of their software products. I would like to add my voice to Kapor's and urge software designers not only to aim for user-friendly products, but also products that ultimately have a moral impact on the user.

6. MEETING THE MORAL CHALLENGE

Thus far, I have given a historical account of the meaning and use of patterns and pattern languages in both the Architectural and Software design fields. My approach was to use a linear-spectrum model proposed by Clayton. I would now like to modify Clayton's model by proposing that rather than seeing the two perceptions of patterns as pole ends of a linear spectrum, one should mentally bend this spectrum to create a circular shape: that is, make the two ends coincide. The Clayton model, I believe, accurately represents the current software design perception of the two views (the *Alexander end* and the *object-oriented end*) of patterns as distinct. I argue that this current perception is wrong and greatly inhibits the capacity of patterns to have a moral influence. I believe that acknowledging and leveraging the interdependence of these two perceptions of patterns will provide the software community with a moral approach to software design. Basing my arguments on Alexander's thoughts on why patterns failed in Architecture and the new insights he has on the use of patterns, I will show how software designers, and indeed the whole software engineering field, must begin to consider a wider and intentional application of patterns and pattern languages.

6.1 Defining Moral Design

Before I delve into the project of moral design, I would like to establish the goals of the moral approach and explain why patterns are essential to this project. The goal of moral design is to produce software[iv] that effectively complements the user's current physical and virtual environments thereby allowing the user to experience a sense of wholeness. This goal requires that software be both technically efficient and morally satisfying. The concept of a Pattern Language is one that, I believe, will allow software designers to achieve this goal. As the use of

patterns in the *object-oriented end* demonstrates, patterns can create technically robust and efficient software systems. In addition, the *Alexander-end* demonstrates that the accurate use of patterns results in living structure, which brings about a sense of wholeness. Thus, by combining these two ends, as I previously proposed, one can see why the concept of patterns provides the solution to this moral project.

6.2 Practicing Moral Design

Back to the project: it should be obvious that the current use of patterns in Software design satisfies the technical demand of the moral project. The question that remains to be answered is how do we use patterns in software development to create a living structure, that is, how do we incorporate the *Alexander-end* into software design? To do this, we need to revisit OOPSLA'96, specifically, Alexander's revised view of patterns.

6.2.1 Alexander's revised pattern theory

Alexander admitted that his initial view and use of patterns "had fallen far short of the mark I [Alexander] had intended." [3] Patterns had allowed people to get a better control of their environment – there were certain isolated features, for example, improved daylight, which were being introduced. However, according to Alexander, these pattern-based buildings were not exactly profound buildings – they seem to have no moral influence. Alexander and his colleagues realized that there was something more fundamental missing from the pattern language.

Fifteen properties:

What they discovered in their search for this missing piece was a “deeper level of structure and a small number (15) of geometric properties that appeared to exist recursively in space whenever buildings had life.” [3] Alexander identified these fifteen properties as being present in all successful patterns (see appendix C for a list of the fifteen properties). He goes on to argue, based on extensive empirical research, that these fifteen properties provide the “ability to be precise about the nature of living structure, in just precisely such a way that it is connected, not only to all mechanical function, but also to the depths of human feeling” [3]. Thus, concerning software development, these fifteen geometric properties (or others that must be identified for the realm of software) will provide the necessary structure that connects the functional demand (technical efficiency, application accuracy, etc) of a software application to its moral demand (enhancing user wholeness).

Centers:

Besides the fifteen properties he mentions, Alexander also identifies certain entities, which he calls, centers. Centers, according to Alexander, “are the primitive elements of all wholeness” [3]. He explains that centers are either alive (in varying degrees) or not and are defined recursively in terms of other centers. The fifteen geometric properties referred to above provide the connective relationships between centers that eventually lead to the recursive formation of other centers.[v] Alexander believes that this concept of centers and the relationship between centers provided by the fifteen properties offers “a complete and coherent picture of all living structure” [3] and that these centers form the building blocks of all life in buildings.

Additionally, Alexander explains that these centers are different from patterns: he describes patterns as being certain "structural invariants that appear within these centers under very, very particular conditions" [3]. With regards to software development, Alexander suggests that the concept of objects as used in Object-oriented programming might be similar to the idea of centers he is proposing. This connection is based on his view of centers as 'field-like structures', which are the 'focal organizing entities' at the core of all structures.

Structure-preserving transformations and generative schemes:

Lastly, Alexander, in his presentation, discusses what he calls "structure-preserving transformations." This is the process by which living structure is produced. Alexander describes this process as an "unfolding wholeness" in which one "[maintains] the whole in each step, but gradually [introduces] differentiations one after the other" [3]. Alexander supports his view by claiming that nature uses this same process of structure-preserving transformations: this is why, he argues, nature, when left alone, produces living structure. This unfolding wholeness is made possible by what Alexander calls a generative scheme.[vi] A generative scheme is a "generative process that is defined by sets of instructions that produce or generate designs" [3]. Compared to the original concept of pattern language, the "generative language" is dynamic and context sensitive: it begins with an existing context and allows users to design and develop things that relate to that context or environment. Alexander claims in his presentation that the designs that are generated by these generative schemes are "guaranteed, ahead of time, to be coherent, useful, and to have living structure" [3].

To recap, Alexander, reflecting on the failure of his original concept of patterns and pattern language, identifies certain elements, which when present, ensure the design and development of

living structures that have a profound (moral) effect on human beings. Alexander states that the process of structure-preserving transformations produces living structure. At the core of all living structure are entities called centers, which are the source of life or wholeness in a structure. Centers, by definition, consist of other centers and the nature of the relationship between the centers that come together to recursively form another center determines the degree of life in the newly formed center. Alexander identifies fifteen properties, which when present in the above-mentioned relationship, cause the resulting center to be alive. Lastly, a generative scheme allows and guides this unfolding wholeness process to occur. Together, these elements allow one to develop a coherent structure that is living and morally beneficial to its inhabitants.

6.2.2 Applying Alexander's revised theory

I believe Alexander's new insights provides us with the tools need to design and develop moral software. I will now propose a theoretical development process that is based on Alexander's ideas and seeks to fulfill both the functional and moral demands of moral design.

Centers:

First, I will go further than Alexander and argue that objects are the centers, as Alexander defines the term center, of software applications. This implies that in any project, one must identify and represent as objects - from the specification for the project - the various entities required to make the project functional.[vii] These objects, as centers, could consist of other objects. As such, the definitions of each object must be carefully done.

Fifteen properties:

Second, the nature of the relationships between the various objects must be determined and checked to see if they conform to some set of properties similar in nature to the fifteen Alexander identified for Architecture. Due to the theoretical nature of this project, I have not been able to determine exactly what these properties will have to be in SD. However, I postulate that each of these will either be, specifically, a universally agreed upon relationship between certain kinds of objects or more generally, certain principles that, by taking into consideration both the physical and virtual worlds of the user, guide the whole design process. Already, work is in progress to translate and apply Alexander's fifteen properties in other fields of computers that are also interested in the concept of patterns: for example, John Thomas, an IBM T.J. Watson researcher, is currently applying Alexander's fifteen properties to the field of Human Computing Interface (HCI) design (see appendix D). According to Alexander, the fifteen properties he identified are what make patterns alive, thus, I believe it is essential that software designers discover what these properties are in SD if we are to develop software applications that generate a sense of wholeness in people.

Structure-preserving transformations and generative schemes:

Thus, after identifying the objects and the relationships between the objects, Alexander argues that we will have general view of the whole that will be the end product. With this whole in mind, developers can then implement the process of structure-preserving transformation where gradual improvements (actual coding) are made to the software. Lastly, what guides and allows this whole process should be what Alexander calls a generative language. The equivalent of this in software development would be a "language" that takes into consideration the context or environment for which the software is being developed.

This idea of using a language construct to manage a software development process is not new: Thomas Erickson proposes this concept in his article *Lingua Francas* for Design: Sacred Places and Pattern Languages as a communication medium (a common ground) between the different parties involved in a development process. He suggests that by using pattern languages as meta-languages, one can produce lingua francas, which are specific to the project at hand. What I am suggesting here is a language that would serve as a common ground, but that would be sensitive to the elements that both exist in and extend beyond the software or hardware the application is expected to interact with. I am thinking here of going beyond the traditional factors considered in interface design to considering external factors of the nature that the field of Human-Computer Interface scholars and researchers are beginning to consider in their work.

Thus, the language will provide, in one sense, a lingua franca that serves as a point of reference for different stages and parties of the development process and in another, as a guide that will seek to keep the project within the context specified in the language. Consequently, each software development project will have its own generative language, which would consider the context in which the software application would be used and would try to make each stage of the development process adhere to specifications of this context.

I believe that by considering these issues discussed above, software developers can achieve the level of success Alexander and his colleagues were beginning to experience in terms of developing living structures, which brings about a sense of wholeness in human beings. By combining the *object-oriented end* and the revised *Alexandrian end* uses of patterns languages,

software developers gain the ability to develop software products that are both functional and “contribute powerfully to the quality of life."

7. USING MORAL DESIGN PRINCIPLES IN INDUSTRY

Although I am proposing a purely theoretical process for moral software development, it is worthy to mention here that some members of the software development community are actively involved in putting these ideas into practice. Urban Code, an Ohio based software company, is currently involved in using this concept of pattern languages to develop whole software systems. Their understanding of this concept is closer to the idea of a generative language I discussed above, as opposed to the original idea of Pattern Language proposed by Alexander.

7.1 Implementation

Maciej Zawadzki, CEO of the company, in response to my questions regarding their understanding of patterns, described how they used patterns and the benefits they were realizing from such practice. First, he explained that instead of thinking of pattern languages as “mere collections of patterns that loosely fit together,” one should think of them as one would of any other language: they consist of grammatical constructs. The constructs in this case would be the patterns, which contribute to what he calls the "pattern programming language." Second, to implement this language-oriented view of patterns, Zawadzki described a “Pattern Oriented programming system” in which a single instance of a pattern (a construct in the language) “provides an abstraction for a collection of classes where the relationships between the classes adhere to the pattern.” Thus, these patterns (constructs) are at even higher abstraction level than classes in Object-Oriented programming.

7.2 Benefits

Zawadzki gave two examples concerning the benefits of using this pattern-based development system. First, he mentioned increased productivity: instead of coding 25,000 lines of object-oriented code, developers using the pattern system were able to produce equivalent length (and equally functional) code with 500 lines of a pattern-oriented code. Second, since the patterns in SD capture the best solutions to problems, the pattern system assures quality and accuracy down to each line of code.

Urban Code is not developing moral software, however, they are implementing, in one form or the other, certain concepts of the development process I am proposing: for example, for each project, they define a language structure composed of patterns, that is, a "pattern programming language," that relate to the functional demands of the project. This is very similar in nature to the context-sensitive generative language I proposed. I believe the success of Urban Code, and other companies like it, in using this new pattern-based approach to development, as opposed to the current, static use of patterns, shows the feasibility, from a technical and economical standpoint, of implementing the moral design process described above.

8. SUMMARY

In this paper, I have tried to argue for a fresh understanding and implementation of the concept of pattern language based on Christopher Alexander's experiences with patterns in architecture. In addition, I have argued and shown how software developers, using this revised concept of patterns, are capable of producing moral systems that bring a sense of wholeness to people. Moreover, I have insisted that software developers, due to the pervasive nature of the products

they produce, have inherited a moral obligation to humanity to pursue moral design. The concept of moral design I have discussed in this paper does not encompass a definitive solution for fulfilling this moral imperative. Rather, it provides a starting place, informed by experiences from the discipline of architecture, from which others can better define and articulate a process by which software developers can “build whole systems that contribute powerfully to the quality of life."

9. ACKNOWLEDGEMENTS

The ideas and thoughts expressed in this paper are deeply rooted in the works of Alexander, et al [1, 2, 3] and Gamma et al [5]. Any synthesis of the concepts expressed in the original works is my own and may not meet the approval of the authors. I am also grateful to the following people for their time and valuable input during the process of writing this paper: Dr. J. Thomas, Maciej Zawadzki, Dr. C. Young, Stephen Schultze and my research advisor, Dr. K. Vander Linden.

10. REFERENCES

1. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. A. A Pattern Language. New York: Oxford University Press. (1977).

2. Alexander, C. The Timeless Way of Building. New York: Oxford University Press (1979).

3. Alexander, C. The Pattern Language Page. http://www.patternlanguage.com/archive/ieee/ieeetext.htm (Oct. 1996)

4. Clayton, R C. CS 8112 Winter 1995 - Software Engineering Seminar. Georgia Tech. 28 http://www.cc.gatech.edu/computing/classes/cs8112m/wi95/claytonsummary.html (Jan. 2002).

5. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass: Addison-Wesley (1995).

6. Vlissides, J. Design Patterns Page. http://www.research.ibm.com/designpatterns/example.htm (Aug. 2001)

Appendix A

Pattern 78 from Alexander's book, A Pattern Language

78 HOUSE FOR ONE PERSON

…the households with one person in then, more than any other, need to be part of some kind of larger household – THE FAMILY(75). Either build them to fit into some larger group household, or even attach them, as ancillary cottages to other, ordinary family households like HOUSE FOR A SMALL FAMILY (76) or HOUSE FOR A COUPLE (77).

Once a household for one person is part of some larger group, the most critical problem which arises is the need for simplicity.

The housing market contains few houses or apartments specifically built for one person. Most often men and women who choose to live alone, live in larger houses and apartments, originally built for two people or families. And yet for one person these larger places are most often uncompact, unwieldy, hard to live in, hard to look after. Most important of all, they do not allow a person to develop a sense of self-sufficiency, simplicity, compactness, and economy in his or her own life.

The kind of place which is most closely suited to one person's needs, and most nearly overcomes this problem, is a place of utmost simplicity, in which only the bare bones of necessity are there: a place, built like a ploughshare, where every corner, every table, every shelf, each flower pot, each chair, each log, is placed according to the simplest necessity, and supports the person's life directly, plainly, with the harmony of nothing that is not needed, and everything that is.

The plan of such a house will be characteristically different from other houses, primarily because it requires almost no differentiation of its spaces: it need only be one room. It can be a cottage or a studio, built on the ground or in a larger building, part of a group household or a detached structure. In essence, it is simply a central space, with nooks around it. The nooks replace the rooms in a larger house; they are for bed, bath, kitchen, workshop and entrance.

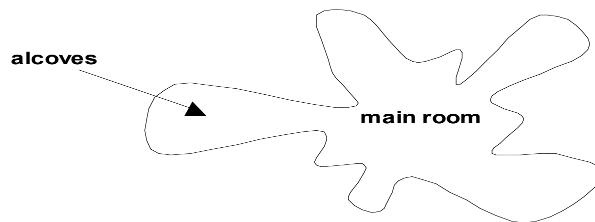
It is important to realize that very many of the patterns in this book can be built into a small house; small size does not preclude richness of form. The trick is to intensify and to overlay; to compress the patterns; to reduce them to simple expressions; to make every inch count double. When it is well done, a small house feels wonderfully

continuous-cooking a bowl of soup fills the house; there is no rattling around. This cannot happen if the place is divided into rooms.

We have found it necessary to call special attention to this pattern because it is nearly impossible to build a house this small in cities-there is no way to get hold of a very small lot. Zoning codes and banking practices prohibit such tiny lots; they prohibit "normal" lots from splitting down to the kind of scale that a house for one person requires. The correct development of this pattern will require a change in these ordinances.

Therefore:

Conceive a house for one person as a place of the utmost simplicity: essentially a one-room cottage or studio, with large and small alcoves around it. When it is most intense, the entire house may be no more than 300 to 400 square feet.



And again, make the house an individual piece of territory, with its own garden, no matter how small-YOUR OWN HOME (79); make the main room essentially a kind of farmhouse kitchen-FARMHOUSE KITCHEN (139), with alcoves opening off it for sitting, working, bathing, sleeping, dressing – BATHING ROOM (144), WINDOW PLACE (180), WORKSPACE ENCLOSURE (183), BED ALCOVE (188), DRESSING ROOM (189); if the house is meant for an old person, or for someone very young, shape it also according to the pattern for OLD AGE COTTAGE (155) or TEENAGER'S COTTAGE (154) …

Appendix B

Documentation Template (Format) for Software Design Patterns

Template section	Section description
Pattern Name and Classification	The name and classification* of the pattern.
Intent	Specifies the function of the pattern.
Also Known as	Other names by which the pattern might be known.
Motivation	An example scenario, which presents a design problem and shows how the pattern solves the problem.
Applicability	Specifies the circumstances in which the pattern can be applied.
Structure	A graphical representation of the classes in the pattern.
Participants	The classes and/or objects participating in the design pattern and their responsibilities.
Collaborations	How the participants collaborate to carry out their responsibilities.
Consequences	What are the trade-offs and results of using this pattern.
Implementation	What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?
Sample Code	Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.
Known Uses	Examples of the patterns found in real systems.
Related Patterns	Patterns that are closely related to or are used in conjunction with this pattern.

This template was developed by Gamma et al to catalog patterns found in Software Design. Most of the descriptions of the template sections come from Gamma et al's book [5].

* See section 4.2.2 and figure 1 for how the patterns are classified.

Appendix C

A list of Alexander's 15 fundamental properties taken from his book The Nature of Order*

1. Levels of scale.

2. Strong centers.

3. Boundaries.

4. Alternating repetition.

5. Positive space.

6. Good shape.

7. Local symmetries.

8. Deep interlock and ambiguity. .

9. Contrast.

10. Gradients.

11. Roughness.

12. Echoes.

13. The Void.

14. Simplicity and Inner Calm.

15. Not-separateness.

* The book has not yet been published. I received this list from J. Thomas who is reviewing a pre-publication version of Alexander's latest work.

Unfortunately, the definitions of these properties tend to be long narratives. Consequently, I have decided only to provide the terms Alexander uses to describe these properties.

Appendix D

J. Thomas's translation of Alexander's 15 properties for Human Computer Interface Design*

1. Levels of scale. This seems fairly straightforward. HCI design, in principle, spans levels of scale from aiding in the design of social structures and large-scale architectural designs that are technologically instrumented and enabled, through the design of applications and task support, down to the level of making sure that fonts are visible and trackpoints have the proper damping functions. (See, "The Million-Person Interface).

2. Strong centers. This is probably one of the most overlooked principles of design in HCI. Often, what exists for the user is a "sprawl" of functions, tool bars, and icons with no overall or subsidiary organization. A better design would allow the user to quickly find a "home base" from which, it would be obvious where other subsidiary home bases would be. There is some sense in which hyperbolic trees, fisheye lenses, and home pages begin to address this.

It should be possible for users to "know where the action is" in entering an application or a web page.

In another interpretation, this refers more to underlying architecture and points to the need for a core of functionality that transcends a specific release or even a specific application. A good underlying architecture will communicate this essential center (related to central purpose or style) to the user.

3. Boundaries. Often it is not clear in existing designs what is possible and what is not possible. There are no strong boundaries on function or on data. What words are "in" the dictionary of a word processor spell-checker, e.g.? There is no intuitive way to know; only on a case by case basis can the user explore this. Similarly, it is not typically obvious how large a file can be handled by a typical application program. The only boundaries that are fairly clear are at the very base hardware level; e.g., how much memory the total machine has, how big the screen is.

A related concept is that of "scope." What is the "scope" of action of an invoked command? It should be clear to the user before taking an action over which actions the object should apply. Recently, I meant to apply an action to a single item and my finger slipped (or the cursor drifted) to the next menu item which applied the action to all (over 1000) such items but there was no preview of this action!

4. Alternating repetition. I interpret this in the context of HCI to be nearly identical to its meaning in the context of organizational design; that is, to refer to activity patterns.

Input, process, output. Pick up the phone, answer a person's question, put down the phone. Contact a client, discover their needs, make the sale. Research, develop, deploy. Set the nail, tap, pound, pound, pound. There are many patterns and if one were to see these patterns laid out a symptom of a well-working organization would be that these activity patterns had a rough periodicity to them. If they show so much variability that no pattern can be perceived, the organization is too disorganized.

By the same token, HCI designers must understand the flow and rhythm of activity at a variety of levels and support this. At the intermediate application level, for instance, we can imagine that there is a cycle to creating a document by inputting words into a document and then editing it; then, repeating the cycle. At a higher level, these documents may be part of a larger cycle of activity in which input for a "fall plan" is solicited by numerous people in the organization and then consolidated. Perhaps, another round of input and consolidation is included in the process. At a lower level, the user probably goes through a process of planning a paragraph and then typing it. At a still lower level, there is an alternating repetition of keystrokes.

In the physical world, animals use widespread "rebound" effects to lessen work. For instance, the achilles tendon and the arch in humans store energy when we run that is released in the next step. Most of our input devices do not currently support this kind of rhythmic action.

5. Positive space. I interpret this to mean two things. First, it is better to have functionality over determined than underdetermined. That is, it is better to have several "alternative" ways of accomplishing the same task than to leave "holes" of functionality that cannot be accomplished in any way. Secondly, at a higher level, the overall design of a system should "push" at the edges; allow for users and teams of users to extend functionality into new areas.

6. Good shape. At the more micro levels, this applies making shapes, sounds, textures, etc. that are aesthetically pleasing and clear. Equally, we can apply the idea of "good shape" to input devices. There are gestures of movement that are more natural and complete and those that are less natural and less complete. At a "higher" and more abstract level, this principle can be applied to overall screen design, tool design, and perhaps even to the design of applications. "Good shape" applied at the application level would mean that the parts of the application tools help pull together to define an overall application shape. This overall "shape" helps a user understand "where" particular functionalities would be found and how the various functions fit together. Typically, "editing" a document, "filing it" and "creating" it are all portrayed as single menu items in some fairly arbitrary menu hierarchy. But, how does this contribute to an overall "shape" of the application? It doesn't. We could imagine instead that if there were one or more models of the overall document creation process, that the various functions could fit within and help define this overall model in some abstract, and quite possibly, in some concrete way.

7. Local symmetries. Generally, if there is a method to move "up" there should be a symmetrical method to move "down." If there is a method to paint the foreground, there should be a method to paint the background. If there is a way to make an object larger, there should be a symmetrical way to make it smaller. As I interpret it, the HCI design principle should be applied both to functionality (symmetry of function) and to implementation (the *way* in which the functions are invoked should be symmetrical).

8. Deep interlock and ambiguity. Ambiguity? Surely, this is something that HCI design should avoid. But is it? At the highest level, if tools are well-designed, the user should not be sure (or even consciously thinking about) whether they are using a tool or interacting directly with the medium. Is a sculptor creating a vision out of the rock, or is the rock revealing to her or him the vision that hides within it? Is the writer creating a fictional world or revealing a deeper truth? Is the writer indeed, writing words or creating images and events? Is the musician playing notes that have been laid down or by the composer or co-creating with the composer by discovering what is there? Is the scientist finding out about nature as it is or building mental constructs that portray a world as built?

The way I interpret "Deep Interlock and Ambiguity" as applied to HCI design is not that (at least typically) it should be unclear what a command does, (although there may be an occasional place for this) but that really good HCI design should support the kinds of ambiguities and interlocks that are characteristic of "creative flow."

9. Contrast. At a fairly micro level, this can refer to such mundane but essential elements as ensuring sufficient contrast between characters to be read on a screen and the background. In general, any two importantly different events from the user's perspective should be sufficiently contrastive to ensure that the user can perceive the difference. This would seem like a fairly obvious principle of user interface design, yet there are many cases where, e.g., it is not clear to the user whether a "message" from the system is a message that essentially means, "Everything is fine and we're just letting you know that" from a message that essentially means, "You are in deep trouble and you'd better do something soon." !! In terms of pragmatic contrast, the words form categories of similarity and difference that are appropriately contrastive for the system designer, but not at all for the end user.

10. Gradients. I see at least two good applications of this principle. First, in terms of finding things and organizing things, it is typically better to have things organized and searchable by gradient than by category. For example, being able to list files alphabetically, by recency, by size, by importance are four ways of graded category and search. This is more amenable to the way human memory and the application of decision criteria typically work than by forcing exact queries. "Give me all the files that were created after April 1, 1999 and have 'Alexander' in the title" may miss exactly the file I'm looking for that was actually created March 31, 1999 and was titled, 'fifteen properties.' The creation of semantic gradients would greatly help in the search and organization of vast quantities of information. In (my personalized) semantic gradient, 'fifteen properties' would be very close to 'Alexander,' whereas in the rather arbitrary alphabetic gradient, they are very far apart.

Again, we can interpret some pieces of the current generations of HCI designs to incorporate some aspects of this in such innovations as fisheye views and hyperbolic trees or dynamic queries.

Secondly, the actions that users take should form natural gradients. When we walk through the physical world, we move continuously through space (except when we fall off a cliff or trip over a branch -- situations we like to avoid in the natural world, though the computer world seems full of them!). As we swing a club harder, the club moves faster, at least up to a point. As we put more effort into throwing a rock, the faster and farther it goes (and the more likely to kill a prey or drive off a predator).

There are aspects of this in current interface design; e.g., the longer we move a cursor, the further it goes; the more times we hit the "bigger" function, the bigger something gets. But there are many areas of action where this breaks down at every level of design.

At the micro-level, most input devices are oblivious to the dynamics of motion. Hitting the "T" key harder doesn't produce a darker "T" on the screen, e.g. shouting at a speech recognition device simply makes it less likely that our words will be correctly transcribed. At the outset, most input devices immediately translate our inherently and naturally (not to mention sophisticated and sensitive!) analog motions *immediately* into digital form (presumably because that is what is easier for the computer to deal with).

11. Roughness. Austin Henderson and Jed Harris (CHI 99) argued eloquently that systems should be designed with this quality (though they did not label it as such). An example they gave was of a paper invoice system a company used. In order to "increase efficiency" the paper invoice system was computerized. What really happened was that the people now have to use both systems. They are required by the company to use the new, expensive, computerized system, but in order to actually get work done, they still have to use the paper system.

Both systems use forms and ask for specific information types to be put into those forms. But in the paper version, people may use "roughness"; e.g., in one situation, where the "address to be delivered" was to be input, the user wrote a notation that before delivery, the dispatcher should call a specific telephone number to determine where the delivery was to be made (because it was a ship visiting different ports). But, in the "intelligent" computerized version, such a notation was impossible because "Call 914-784-7561 and speak with Mr. Thomas to determine the current whereabouts of this ship" is clearly not an address; in fact, it doesn't even fit in the allocated space.

Roughness is an especially good property when one considers architectures that are to support cross-cultural adaptations. To take just one obvious example, the number of "bits" required to specify an English character set is not appropriate for most Asian languages.

12. Echoes. This property offers an excellent potential method for teaching a complex system. If there are echoes of complex advanced function or overall design within the setting of a small, confined first application or small function set, the transition of the user from novice to expert can be facilitated. For instance, if a small introductory word processor includes functions for making characters larger or smaller, might it work for this to foreshadow that graphics functions and more abstract functionalities also allow changes in scale, similarly evoked?

13. The Void. Christopher Alexander writes (The Nature of Order, Part I, p. 80) “In the most profound centers which have perfect wholeness there is at the heart a void, which is like water, in infinite depth -- surrounded by, and contrasted with the clutter of the stuff and fabric all around it....Is there a way that the presence of the void arises mathematically, as part of a stable unified structure, or is it merely a psychological requirement? It is the latter. A living structure can’t be all detail. The buzz finally diffuses itself, and destroys its own structure. The calm is needed to alleviate the buzz.”

One obvious interpretation of how this might apply to HCI design is to point to the danger of over-optimizing and re-engineering. This is another way to conceptualize the point that Harris and Henderson made.

Another interpretation is that there should be a place for the individual team, the individual user, and the individual role of the user to "fill in" with specific function. A fairly trivial example of this is simply that most "spell-check" dictionaries and Automatic Speech Recognition facilities allow the user to add their own vocabulary elements. Many systems also allow the user to store their own objects, functions, macros, etc.

14. Simplicity and Inner Calm. Christopher Alexander (Ibid., p. 85) writes “Everything essential has been left; nothing extraneous is left. But the result is simple in a profound sense, but not in the superficial geometric sense. So it is not true that outward simplicity creates inner calm; it is only inner simplicity, true simplicity of heart, which creates it.”

Amen. This seems to be crying out against "feature-creep" --- the adding of the union of whatever features any competitor actually has or has pre-announced and whatever features any programmer on the team has time to program regardless of whether such features add to the actual utility of a system, application, or widget. The lack of simplicity and inner calm is even more apparent when needed aspects of a system are not allowed for in the basic architecture and must be "hacked in" later. So, it would seem that the architecting for breadth and then being austere in implementation might be one heuristic for achieving this.

15. Not-separateness. In HCI design, this seems to reflect the following interconnected set of notions.

Users do not come alive at the instant they begin using your system and die when they exit. They come with preconceptions and the system affects their lives outside and after interacting with your system. This is most dramatic, perhaps, in the case of Repetitive Stress Injury, but applies more subtly in various other cognitive and perceptual domains as well. What is the impact of continually placing the sensitive analog body movement capabilities against the crude, digital, discrete world?

Information, artifacts, and results do not live only in the space of the computer system we are designing. I visited a lab with Lewis Branscomb (former Chief Scientist of IBM) once where we were being shown a new printer. The printer was cheap and produced a curly, silver piece of paper about 5 inches by 4 inches. Lew asked the inventor, "what will the person do with this piece of paper after he gets it off the printer?" It was clear that the inventor had never considered this question. There was no existing infrastructure (folders, binders, etc.) to support the collection and use of curly, silvery, 5" by 4" pieces of paper.

Further examples of considering the "ecological validity" of design can be found in Thomas and Kellogg.

A note from J. Thomas:

I think that the fifteen properties may serve as guiding principles to be used in conjunction with the development of a more specific "Pattern Language" for HCI. The current draft is merely a first attempt and could benefit greatly from Dialogue and critique.

* The ideas expressed here are those of J Thomas.

END NOTES

i Information from Charles Young, a pre-Architecture program advisor at Calvin College

ii Many contemporary architects rejected Alexander's pattern approach because they considered it highly abstract and impractical. See Nikos A. Salingaros's The Structure of Pattern Languages. http://www.math.utsa.edu/sphere/salingar/StructurePattern.html

iii The idea is that adaptation and generativity occur naturally in traditional societies where the local people build their own houses, etc. Thus, by identifying with the common relationships between a set of patterns (i.e. sharing a pattern language), that exists in their society, the local people were able to generate complete living structures. See [3] for a detailed explanation of this concept.

iv My definition of software is that used in Terry Winograd's Bringing Design to Software: software is a "medium for the creation of virtualities – a world in which the user of the software perceives, acts and responds to experiences." It is helpful, in this case, to think of the user as an inhabitant, rather than a user of the virtual space.

v See Alexander's The Nature of Order for a complete explanation of this concept of centers: this concept of centers and the 15 properties form the foundation of that book.

vi Alexander states that his original concept for pattern languages was based on certain generative schemes that existed in traditional cultures. However, in trying to imitate these schemes in his pattern language, he ended up spending more time on the individual patterns and less emphasis was placed on the fact that these patterns had to be used sequentially to produce a coherent result.

vii Enoch Y. Wang and Betty H.C. Cheng in their paper, A Rigorous Object-Oriented Design Process, describes a design process which achieves mathematical rigor in design specifications and applies formal methods during design, which enables the developers to detect and avoid design flaws.