Towards a Theory of Software Design:  
Timeless Principles of Software System Design

David R. Wright  
Computer Science Department  
North Carolina State University  
Raleigh, NC 27695-8206  
drwrigh3@ncsu.edu

ABSTRACT
The design and implementation of a software system is the result of many interwoven sequences of decisions. Often, these decisions are made under less than ideal conditions of uncertainty and/or incomplete information. Furthermore, many of these decisions do not have clear "right/wrong" answers — they are value judgments dependent upon the current knowledge of the system and its intended domain. Unfortunately, many of the "design principles" of software development are either specific to a particular aspect of the design or applicable only in a particular implementation paradigm. This paper presents a set of axiomatic design principles, based on established design practices from other disciplines, and illustrates their applicability to software system design. Several well-known specific software design principles are derived from this fundamental set. Preliminary results of an ongoing study using these principles are discussed to demonstrate their value in guiding judgment decisions made with incomplete knowledge.

1 INTRODUCTION
Designing a software system is a process of exploring, learning, and structuring information about a central problem or goal and the environment in which that problem must be solved. Designers make judgements about the value or fitness of particular aspects of a solution, and sequences of these decisions generate the artifacts or models representing the unfolding design of the system. Design problems are ill-structured [1], and are not generally amenable to linear problem-solving processes. Systems of any significant value and/or complexity commonly expand in unexpected directions, require the development of learning strategies about the problem domain, and occur in complex social environments [2]. The designer must generate possible solutions and evaluate them within the target environment, seeking to maximize the fitness of the solution for the initial and derivative problems while minimizing the negative effects of imposing that solution on the target environment. The designer must be able to objectively consider alternatives and maintain an open mind to change.

Under these conditions, the designer must have a set of fundamental principles that serve as guides for his or her decisions. Software engineering texts list many “principles of software engineering” such as Modularity, Information Hiding, Open-Closed, etc., associated with specific topics in the text. While these principles are certainly valuable within the scope of the associated topic, they are specific to that topic. For example, Pressman devotes an entire chapter to principles of software engineering practice, part of another chapter to principles of object-oriented component design, and part of another chapter to design principles for web-based applications [3]. A review of other software engineering texts, including [4]–[8], found similar usage of design principles. Furthermore, these principles are identified descriptively rather than prescriptively, and in this form they offer little guidance towards defining and designing the design problem, a critical aspect of design [2], [9], [10].

The resolution proposed for these limited-scope principles is the development of a set more general principles for software system design. This task is not without significant risk, since the degree of generality of a principle is directly proportional to the set of attributes or behaviors that the principle constrains. Any such principles must strike a balance between generality and specificity, and be structured such that they may be applied to design situations at different levels of detail or abstraction. Existing principles must also be derivable from these general principles. In Section 2, we develop such a set of axiomatic principles, showing how they can be used to justify design decisions. We use these principles to derive several well-known principles of software design and implementation in Section 3 to further illustrate their fundamental nature. The paper concludes with a discussion of some early results from an ongoing study applying these principles to ill-structured and weakly-defined design tasks in a classroom environment.

2 TIMELESS PRINCIPLES FOR SOFTWARE SYSTEM DESIGN
Software systems are complex and dynamic networks of information processing and exchanging components. Their design is subject to change over time as the executable code is modified and extended, as well as through adaptation by their users. They also affect the computational, physical, and social environments in which they exist [11], [12]. Architectural design is also concerned with complex,
dynamic systems: buildings, their environment, the people that live, work, and interact with them, and how the system evolves over time are all important considerations for the architect [13]. Christopher Alexander, an architect whose work with architectural design patterns [14], [15] has had a lasting effect on software design, introduced his most recent work to the computing community in 1996 [16], urging researchers in computing and software to "become responsible for the form and structure of the built environment." So far, this work has attracted only modest attention from researchers and practitioners in this discipline (e.g., [17]–[22]), a result of its abstract and theoretical approach to what Alexander calls “living architecture.”

Alexander’s work deals with complex systems that are under the constant pressure of change. In the four volumes of The Nature of Order he relates the dynamic complexity of urban architecture with living systems, building architectural models that incorporate this dynamic structure and behavior. Alexander identifies fifteen properties of living, sustainable systems, whether natural or human-built [23, pp. 145-229], [24], restating these properties as generative processes, or structure-preserving transformations in the second volume [25, pp. 66-76]. These properties/transformations describe the attributes, behaviors, and relationships between entities in complex, dynamic systems.

A principle is a rule or standard, a basic or essential element that determines the intrinsic nature or behavior of something [26]. When the dual nature of Alexander’s fifteen properties/transformations are considered together, they are fundamental principles of sustainable complex systems. Coplien [17] and Gabriel [19], [20] have attempted to visually apply these properties to source code, following Alexander’s visual characterizations, but these attempts do not capture the fundamental nature of Alexander’s work, nor are they easily applicable throughout the software development process. This work considers these properties/transformations as principles guiding the software system designer in the exploration and organization of the information content of the system being designed. In this section, we define these principles in terms of information content, using Alexander’s original names and ordering, and include examples illustrating their application.

**Principle 1. LEVELS OF SCALE: The hierarchical information content of design elements in sustainable systems should be consistently scaled to optimize coherence and stability.**

Large or disproportionate changes in information content are difficult to understand. Salingaros showed that the base of the natural logarithm is a common scaling factor in naturally-occurring systems and is also an optimal factor for architectural forms [27]. To the software designer, this means that there must be a consistent scaling of the levels of abstraction and information content in design models to optimize the communicability of those models. This applies to hierarchies of classes and modules, architectural components of the system, and information transfer, as well as to the visual and cognitive aspects of user interfaces [28].

**Principle 2. STRONG CENTERS: Important and/or critical elements orient and focus the understanding and knowledge of a system.**

While this might seem obvious, identifying the key elements and decisions in a complex software system is often a difficult task. The critically important aspects of a system are often identified later rather than earlier in the design process [9]. Alexander notes that strong centers evolve as the design of a system develops, and elements that were weak or even latent are defined and strengthened by supporting elements [23]. Agile software development proponents recommend delaying important decisions as long as possible so that they can be made with as much information as possible [29]. Strong centers are also the “key abstractions” captured by a design component.

**Principle 3. BOUNDARIES: Individual concepts and design decisions should be distinguishable from others while also being unified within the system.**

The various parts of a software systems should be clearly distinguishable from each other, separated by clearly-defined interfaces that are themselves rich with information about what they enclose and how information is allowed to cross the boundaries they establish. Boundaries exist at all levels of software development, from programming languages and their primitive constructs that separate the programmer from the hardware, to function/method signatures, type systems, APIs, network protocols, and user interfaces.

**Principle 4. ALTERNATING REPEITION: Complimentary alternation and repetition between system elements unifies, intensifies, and stabilizes the dynamic structure and behavior of a system.**

Alternating repetition is distinctly different than simple iteration. It is the rhythmic interaction between two or more elements in a system. This principle is fundamental to the dynamics within a software system, as it prescribes complimentary and cooperative interaction between parts of the system. Scheduling algorithms used to control processes and threads, I/O operations, and other concurrent operations are an example of this principle when they are designed to avoid deadlock, starvation, and race conditions.

**Principle 5. POSITIVE SPACE: Each element in a system must be whole, well-defined, and substantial in its own right, and must fit within the system as a whole.**

Well-designed and implemented software systems do not contain unnecessary components or code; every part of the system has a definite purpose, and all of these parts work together to achieve the goal of the system. A rule of eXtreme Programming advocates ignoring potential future requirements and added functionality in favor of concentrating development efforts on what is actually needed at the present time [30]. Refactoring is a process used, in part, to enforce this principle in a developing software system.

**Principle 6. GOOD SHAPE: Sustainable components are made up of smaller, coherently related components that are also sus-
Robust systems must be well designed and implemented from the most primitive elements through the large-scale system architecture. Every part of the system is important — a chain is only as strong as its weakest link. Elementary good shapes in software are those basic design and implementation elements that are well-defined and well-understood: programming idioms and language constructs, documented algorithms, domain-specific business rules, etc. These basic elements are combined into larger, more complex components while maintaining a vision of how each part contributes to the whole of the system, including the potential for unintended interactions with other parts of the system.

**Principle 7. Local Symmetries:** Groups of related design elements are stronger and more coherent when when the relationships are symmetrical and balance opposing forces or interactions.

Localized symmetries arise in a complex system from interactions between related parts of the system. In the design and development process, parallel and complementary attention must be paid to all of the parts involved in an interaction to maintain the balance of forces in the interaction. For example, the computation invoked by a function call should be proportional to the data provided as parameters to the call, as well as to the context in which the call is made. Data structures reflect the kind of data they hold and how it will be accessed. Specifications reflect the requirements they satisfy, and design elements reflect the specifications they implement.

**Principle 8. Deep Interlock and Ambiguity:** Strongly related components share well-defined boundaries that create coherence without generating unnecessary interdependence between the components.

Large software systems are modularized to simplify their development and maintenance, and the modules must work together as a coherent whole to accomplish the goals of the system while avoiding tight coupling that make changing the system difficult. This principle is closely related to Boundaries, and helps define how boundaries between parts of a system should work. These interfaces between system components provide the mechanism to coherently connect the components together, while preserving the ambiguity of the processing that occurs within each component (from the perspective of the other components).

**Principle 9. Contrast:** Opposing forces generate and define key balance points within a system.

Input-output, producer-consumer, and transmitter-receiver are examples of opposites common in software systems. Without these opposites, it is difficult, if not impossible, to determine if the system is doing anything useful. Maintaining contrasting elements and components helps distinguish functionality between them, making their development and maintenance easier. Understanding the opposing forces and processes within a system can also help to define how concurrent operations interact and relate to each other, offering insight into how they should best be scheduled.

**Principle 10. Gradients:** Gradual transitions are easier to understand and control than abrupt discontinuities as internal and external system conditions change.

Normal computation in a software system proceeds smoothly along predictable paths defined by the implementation. Smooth transitions between programmatic components are easier to trace, debug, maintain, and modify when necessary. Data should flow smoothly through a software system, adding, modifying, and removing information as needed, but always in a logical and orderly manner. Similarly, processing control should also move through the system without abrupt changes. The initial function calls that start the system are very general, and initiate sequences of calls that move smoothly and gradually towards the specific actions the system is designed to accomplish. User (and other external agent) interactions with the system should also proceed in an orderly fashion to support the user’s understanding of the system and their productivity as they use it.

**Principle 11. Roughness:** Small variations provide slack that allow robust systems to bend without breaking, absorb the shock of unpredictable events without failure, and adapt to minor changes in the computational environment.

Although computer software is ultimately predictable, in real-world situations the complexity of software system is such that this predictability is not realizable, or is dependent upon unpredictable external events. Scheduling algorithms often incorporate randomness into programmatic delays to avoid cyclic deadlock situations that could occur when the entities waiting for the scheduled resource have synchronized waiting times. Local properties allow users to customize their interfaces to their particular needs and tastes, as well as allowing system administrators to tailor client functionality and performance to local conditions (e.g., operating system, network connectivity, hardware performance, etc.).

**Principle 12. Echoes:** A complex system is strengthened by common themes that echo throughout its depth and breadth.

Echoes represent tradition and the acceptance of what works, as well as the consistency of interactions across a system. For example, there are many common keystroke combinations users have grown accustomed to over time, i.e., the use of CTRL-S or ALT-S to save the current workspace. This functionality extends across many different software applications and operating systems. User interface designers must account for the habits and metaphors users employ when using software systems. In another respect, echoes can represent coding standards and conventions used in the implementation of the system, which should be consistent regardless of who writes the code and when it is written. A third example, particularly in systems implemented with object-oriented languages, is that the system echoes the real-world entities and processes it automated. Retaining these metaphors into the
structure and implementation of the system can help those who must modify the system later understand it better.

**Principle 13.** **THE VOID:** Empty space is essential for growth and new development within a complex system.

Software systems must function within hardware and operating system constraints such as memory allocation limits, CPU time, network and other I/O connections, etc. These resources must be used efficiently not only with respect to the system’s internal processes, but also in competition and cooperation with other software systems. Generally, these resources should only be requested when they are needed, and released back to the operating system when that need has been satisfied. Software systems must also provide space to accept input data, both in terms of programmatic memory and in terms of user input space.

**Principle 14.** **SIMPPLICITY AND INNER CALM:** Always do the simplest thing that could possibly work.

Software systems commonly compete with other applications for access to (relatively) scarce hardware and operating system resources. Extraneous and unnecessary functionality can wastefully consume these resources, and should be avoided in robust systems. Superfluous elements should not be added into the design or implementation of a system, and when unnecessary components are encountered, they should be removed. This strategy applies to all levels of the system model from the architecture down to the source code implementation of the system. It should also be applied to the communicative properties of the system, i.e., naming conventions for modules, classes, etc. should be simple metaphors for the real-world entities they represent in the system.

**Principle 15.** **NOT-SEPARATENESS:** A sustainable system is one with its environment, not separate from it.

Not only should software systems accomplish their intended functional goals, they should also exist cooperatively with the other software applications running on the same platforms [31]. In the larger human context, a software system is often one of many resources at a user’s disposal, and one of many entities competing for the user’s attention. Designers must be aware of the impact their systems have on their users, and should design systems that integrate smoothly and naturally with the human environment in which they are used [32].

It should be noted that these principles are inductive in nature. A complex system is itself made up of smaller systems, down to an arbitrary atomic level dependent upon the level of detail necessary to understand and manipulate the system. For software systems, the atomic level will usually be that of the source code and primitive constructs of the programming language(s) used. When third-party software components are used, the atomic level for these elements will be the APIs of the components, since the internal implementation of the components is not open to examination or modification.

### 3 DERIVED PRINCIPLES

Fundamental principles should be applicable at any level of abstraction in the design process. In this section, three principles of software design and development are derived from combinations of the fifteen principles defined in the previous section. The domain-specific principles derived represent a diverse range of software design issues and illustrate the fundamental nature of the principles presented in this paper.

We begin with an important principle of object-oriented design: the *Open-Closed Principle*, which Martin [33] (from [34, p.23]) states as “A module should be open for extension but closed for modification.” The **GOOD SHAPE** and **STRONG CENTERS** principles form the basis for modularity within a complex system. Strong centers (modules) are strengthened and intensified by **BOUNDARIES** that define their responsibilities and interactions with adjacent modules. The strength and size of these boundaries should be proportionate to the importance and size of the module they enclose, based on the **LOCAL SYMMETRIES** principle. By the **POSITIVE SPACE** principle the module, including its bounding interface, must be an essential part of the overall system, and a larger strong center within that system. **DEEP INTERLOCK** and **AMBIGUITY** asserts that boundaries contain and hide the internal structure of a module while allowing other elements of the system to use and extend the module’s functionality. Reversing out initial perspective on the **GOOD SHAPE** principle, in conjunction with the **ECHOES** and **ROUGHNESS** principles, this well-defined, strongly-bounded, and essential module can be extended to provide new functionality for the system without altering the existing behavior of the module.

Next, we consider an implementation-level principle, McConnell’s *Principle of Proximity* that requires related actions to be kept close together in source code [35, p.181]. **STRONG CENTERS** are not limited to module-level constructs. Any important decision or information is a strong center with respect to other information at the same level of abstraction or detail. Thus, in source code, individual statements, expressions, function calls, variables, etc. are centers of information. By the **STRONG CENTERS** and **LOCAL SYMMETRIES** principles, important information does not exist in isolation — each element has parallel elements that define and strengthen it. **POSITIVE SPACE** requires an information space to be filled with intimately related elements, and by extension, that closely related elements should be grouped into intense centers rather than scattered over a larger area along with unrelated elements.

Finally, one of the basic principles supporting usability of software is that of *Familiarity and Affordance*, which states that designers should leverage a user’s past experience with similar or analogous tasks to create user interfaces that are easily usable by people who do not have experience with a new system [36, p.167-8]. The **NOT-SEPARATENESS** principle above requires systems to smoothly integrate with their environment, including users’ work patterns. One aspect of that integration, with respect to users, is that the system should be understand-able to them in the context of the tasks the software helps
them perform. If the new system ECHOES their previous experience with the task, they will be able to establish a cognitive connection more quickly, lowering the risk for lost productivity as users learn the new system. Furthermore, the key tasks the system is intended to perform should be clearly evident and recognizable (i.e., STRONG CENTERS) as such to the user.

These examples demonstrate the utility of these principles as elements for defining higher-level principles of software system design and development. Defining these principles in terms of a common core vocabulary offers the possibility of discovering and explaining relationships that would otherwise remain hidden. Plans for future work in this and other areas where these principles can be applied are discussed in the next section.

4 CURRENT AND FUTURE WORK

The fifteen principles identified in this paper are currently being used in an ongoing classroom-based exploratory study with advanced undergraduate Computer Science students. There are two goals for this study: first, to refine the language used to define the principles; and second, to explore how students with limited design experience might use the principles to help them solve software design problems outside their existing experience. The final results of this study will be used to guide and condition further research in this area, as discussed later in this section.

The class in which these principles were presented and are being used is a senior-level special topics class entitled “Perspectives on Software System Design.” The overarching goal of the course is to introduce students to some of the challenges faced in the design and development of large, complex software systems. The fifteen principles were presented early in the semester, in a series of three lectures which were supplemented with a handout. For the rest of the semester, students are required to use them to support their design decisions, both in homework assignments and class discussions.

The initial language used to present the principles was a close paraphrase and adaptation of Alexander’s original definitions [23], [25]. Class discussions and student comments led this author to refine the language in the notes handout to simplify some of Alexander’s more abstract descriptions, as well as incorporating terminology more specific to software system design. As the semester has progressed, the principles have been reinterpreted from many different perspectives relating to lecture, discussion, and homework topics, and the presentation in this paper represents a second revision to incorporate the new understandings that have been gained from these different viewpoints.

A detailed analysis of the students’ work products in the class is currently in progress, but two important insights have become evident from the graded evaluation of this work. First, the five to seven students who consistently applied these principles in their justifications for solutions to design problems performed significantly better than their peers who did not make extensive use of the principles in their solutions. Specifically, they correctly identified more necessary design elements in the given problems, which were deliberately constructed to be outside their previous experience. These students were also the only ones to discover latent dilemmas built into the problems that required more information than was given. These assessments seem to indicate that these principles may stimulate and help structure critical thinking skills in a software design context.

This insight suggests that techniques for stimulating and teaching critical thinking skills could provide a more effective platform for introducing these principles. Additionally, techniques for constructing problems for assessing critical thinking skills should also be applicable to developing both classroom and research problem sets that would provide better qualitative and/or quantitative data on how the principles are used in practice.

The second insight gained from teaching this course suggests that these principles may also be applicable to the structure of a software development organization. The relationship between an organization’s structure and work products was first noted by Conway [37], and has been considered in more detail and specificity towards software system development [38]-[41]. This insight has developed from peer evaluations students completed as part of the project part of the course. Students worked on project tasks in teams of three or four, changing teammates and tasks each week for the last half of the semester. As part of their grade for the projects, students had to complete a peer evaluation for each of their teammates. Use of the principles in this evaluation was not suggested in any form, but every student in the class used one or more principles to describe strengths or weaknesses in a particular team configuration. This was an unexpected result, particularly from the two or three students who seemed intent on resisting the use of the principles in their work. This insight opens a new application domain for these fundamental principles, and a wealth of related new research problems.

The current analyses focus on confirming a correspondence between the use of these principles and the quality of design artifacts, and identifying common patterns of application of the principles to different types of design problems by novice software designers. Future work will extend this exploratory study using better instruments for assessing how the principles are used, as noted earlier in this section. A second application that is under development will use these fifteen principles as a basis for analyzing existing software systems, including the design of qualitative scales for each of the properties, and a tool suite for examining source code. The second insight mentioned above may offer another area of application for these metrics, and their design will take into account social and organizational structures that may be analyzable with these principles. We are also interested in the conjunction of these two areas: software development processes and process improvement.

In conclusion, the fifteen principles presented in this paper, drawn from centuries of experience in architecture, show promise and potential as a basic tool for evaluating software system designs and guiding their development,
based on preliminary results of an exploratory study. These principles provide a vocabulary for describing artifacts of the software design process, as well as software development processes and the dynamic structures of organizations that develop software systems. The development of metrics for measuring the absence, presence, and quality of an entity exhibits based on these principles will position them as a fundamental notation system for theorizing about the complex, dynamic nature of the people and processes involved in developing large software systems.

REFERENCES


