The Layers of Change in Software Architecture

Jorge Luis Ortega Arjona and Graham Roberts
Department of Computer Science, University College London
Gower Street, London WC1E 6BT, U.K., email {J.Ortega-Arjona G.Roberts}@cs.ucl.ac.uk

Abstract

Analogy is a common technique used in many knowledge fields. It allows a system to be considered from the perspective of an equivalent system, making it easier to think about and understand the nature of the original problem, and find good solutions to problems. In this paper, we propose an architectural description using an analogy between building and software architecture, aiming to help software designers and programmers express and understand their ideas more clearly. The concept of the Layers of Change is used as a concept model to the study, design and construction of software programs.

Keywords

Analogy, Software Architecture, Software Patterns, Object–Oriented Programming

1 INTRODUCTION – THE LAYERS OF CHANGE IN BUILDING ARCHITECTURE

Software has evolved into a wide variety of structures, some of which are complex systems that often include more than one program, protocols of communication between processes, many functional elements, and so on. Software construction has become a task as complex as the construction of other artificial systems, such as high performance aircraft or modern skyscrapers. The discouraging fact is not the complexity of software, but that software architects do not possess the experience – as a range of tried and tested techniques and tools – of constructing software equivalent to that of aeronautical engineers or building architects. However, as it is possible to recognise that the construction of artificial systems share analogous features, it can also be considered that practices proposed for solving problems in a more developed field may be represented and used through analogy in another. This could be the case between building and software architecture. The problem is to find a correct analogy in software to good practice in building. In order to illustrate this concept, this paper proposes an architectural description or interpretation in

software of what is know as the Layers of Change in building architecture. The Layers of Change are proposed by Brand (1994) in his book *How buildings learn*. In this book, Brand makes the radical proposal that buildings adapt best when constantly refined and reshaped by their occupants. Architecture can mature from being "an art of space to become an art of time". Based on an original description by the architect Frank Duffy, Brand proposes a general purpose "six S's" for the Layers of Change in building architecture: *Site*, *Structure*, *Skin*, *Services*, *Space Plan*, and *Stuff* (Brand 1994).

2 CONSIDERATIONS AND ASSUMPTIONS FOR THE ANALOGY – THE CONCEPT OF HABITABILITY

The idea of Layers of Change in a building arises from the form in which a building is planned and created, following the actual and future needs of its occupants. Trying to apply these concepts to software architecture, it is necessary to make some basic assumptions, like what is the analogy of a building in software terms, who are its occupants or inhabitants, and how they are able to refine and reshape their software buildings. For this, an interesting analogy can be taken from the book *Patterns of Software: Tales from the Software Community* by Gabriel (1996). This analogy is presented introducing the concept of habitability in software. Originally, the word *habitability* is related to issues about buildings and the quality of living in them. Nevertheless, Gabriel uses this concept to explain an important characteristic of software, which enables programmers and developers to "live" comfortably in and repair or modify code and design. Gabriel applies the concept of habitability as an analogy between software and building, comparing a program to a New England farmhouse which slowly grows and is modified according to the needs and desires of the people who live and work on the farm.

"Programs live and grow, and their inhabitants – the programmers – need to work with that program the way the farmer works with the homestead" (Gabriel 1996).

Therefore, any software program (or simply software) can be considered as an analogy of a building, and programmers represent the analogy of the occupants of software. Refining and reshaping software can be seen as the process of testing, debugging, extending, adapting and maintaining it through time.

3 THE LAYERS OF CHANGE IN SOFTWARE ARCHITECTURE

Continuing the analogy, an architectural description of the Layers of Change in Software can be proposed, using as examples concepts of Computer and Software Architecture, Software Patterns, and Object–Oriented Programming. However, the interpretation of the Layers of Change for Software we give

here is an open concept: it is prone to be changed, adapted or modified to cope with other software development approaches, paradigms or techniques. Taking Brand's Six S's – which are oriented toward building architecture – we try to produce an equivalent version for software architecture. From the programmer's perspective of design and development, the layers are as follows.

3.1 Software Site

The Software Site layer objective is to provide a stable base on which we construct software programs. In our analogy, this can be represented simply by the hardware elements of Computer Architecture and the software environment in which a software program will be developed. In general, a good part of the result of the software development and construction relays on these elements.

A computer is constructed from basic building blocks such as a memory system, processor, and I/O devices. Regardless of the nature of the computer in which they are embedded, the functional behaviour of the components of one computer are similar to that of any other computer, whether it be a personal computer, or a supercomputer; memory performs storage functions, processors execute code, and I/O devices pass data from a processor to the outside world. The major differences between computers lie in the way the modules are connected, the performance characteristics of the modules, and the way the computer system is controlled by programs. Perhaps in the future computers will exhibit other functional behaviour but, up to now, no radically different approaches have achieved any widespread use.

During a software development, the elements that compose the Software Site should be analysed determined as the first elements that influence the design and implementation of software programs. "Site is eternal" (Brand 1994) during the lifetime of a software program.

3.2 Software Structure

The Software Structure layer objective is to provide stability and support to the other subsequent layers. It is represented by the basic organisation schema of a software program.

Software Structure is the description of a software program as a set of defined subsystems, specifying their responsibilities, and including rules and guidelines for organising the relationships between them. Software Structure is concerned with the issues about partitioning a complex software system. The partition of software is necessary to cognitively deal with complexity. A big problem is divided into smaller subproblems that are possible to reason about, and perhaps perform some work on separately, at more comfortable

cognitive level. Software Structure can be described in terms of architectural patterns (Buschmann *et al.* 1996):

"An architectural pattern expresses a fundamental structural organisation schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organising the relationships between them" (Buschmann *et al.* 1996).

Software Structure changes (or at least should change) very little or not at all during the lifetime of a software program. An important factor in determining the success or failure of a software program relies deeply on choosing an appropriate Software Structure that properly matches with the requirements and resources. A careful selection of the Software Structure has to be made; if the Structure has benefits and liabilities, these will be reflected by the software program during its entire lifetime. At this level, architectural patterns can help to define or choose an initial overall Software Structure, as they express and specify structural properties, and can be used as the starting point of coarse–grained design. Software Structure is properly the first design stage of a software program. In practice, software development consider the Software Site as provided or given, and initiates from the Software Structure design. Due to it supports and stabilizes all other layers, any considerations and decisions taken during its design affect the other layers design and construction. "Structure persists and dominates" (Brand 1994).

3.3 Software Skin

The Software Skin objective is to give an appearance to a software program, exhibiting its functionalities. In general, it is represented by all the elements that allow user interaction, mainly expressed by the use of graphical user interfaces, in which end users conceptualize how the software program works.

Software Skin ranges from complete graphical environments to simple text screens. User friendly software programs try to enhace their Software Skin to make them attractive and understandable. The aim of Software Skin then is to support the usability of software, allowing users to learn about the software program and use it effectively. During software construction, Software Skin has to be considered as integral part of the software program. For its definition and design, the conceptual model embodied by the system should be considered. The conceptual model is a description of how the system works from the end user perspective, presented as a plan of interaction between the user and the system, and therefore, has an impact on the design of the Software Skin and other layers. An important characteristic of interactive software is to keep functionality independent of the user interface. Software functionality usually remains stable, whereas Software Skin is often changed and adapted to support different standards, customer requirements, or aesthetic considerations. Software Patterns (Buschmann *et al.* 1996), like Model–View–Controller or

Presentation–Abstraction–Control allow adaptation of a user interfaces without affecting its functionality or data model. Design patterns, like Composite, Decorator, Abstract Factory, Bridge, etc. (Gamma *et al.* 1994), are intended to help with the development of the Software Skin layer of software programs.

The most visible changes that can be performed on a software program are at the Software Skin layer. Just observe the graphic or programming environment of different users and programmers. Although they preserve the same elements, most of them represent a modified version of an original. Changes can be performed quickly and easily, according to necessity or taste. Also, it is noticeable that from one software release to another, developers commonly introduce changes in the user interface of a software program. The change may follow different causes, from newly added capabilities to aesthetics. These changes represent, in a more visible form, the improvement of the software as a product but occasionally, as with buildings, this is the only improvement of the software. "Skin is mutable" (Brand 1994).

3.4 Software Services

The Software Services layer objective is to provide support for common activities during the use of a software program. Software Services are defined as those elements which are part of the "working guts" of a software program. From a programming point of view, Software Services can be found in the form of all those prebuilt standard components that provide common functionality like mathematical, input/output, and disk access. Software Services are available as libraries of standard components from reliable suppliers or experienced programmers. Often, Software Services found in a library should be customized by the designer or programmer for a particular software design. For example, libraries of classes can be used effectively for customization in Object–Oriented programming (Stroustrup 1991).

Design patterns and idioms can help with the use and development of Software Services. Idioms provide information about how to use Software Services and the rules and expectations when using them. Design patterns capture and organize reusable pieces of software as new Software Services and can be used to clearly express dependencies between Software Services. Memory allocation techniques, exception handling strategies, and input/output mechanisms are Software Services that can be easier to comprehend and apply when expressed as Patterns.

The correct use of Software Services can lead to more manageable, extensible and maintainable implementations. However, they are not universal standard programming tools or programming libraries. They depend closely on features and resources of the computer system where they are suppose to be used. Due to this, it would be unreasonable to expect them to be fully standard. Most Software Services can be considered standard on only a specific

type of computer system (Stroustrup 1991). When computer systems change, Software Services have to evolve with them. "Services obsolesce and wear out" (Brand 1994).

3.5 Software Space Plan

The Software Space Plan layer objective is to organize the different partial tasks or activities performed by a software program. It represents the way in which the layout of a software program is organised. Following a particular paradigm or technology, data structures and functions are organised as abstractions in the form of software components and interfaces among them. Object–Oriented Programming, for example, presents organisations of classes that can be used as a layout for cooperating objects. This is the base for the concept of design patterns (Buschmann *et al.* 1996, Gamma *et al.* 1994):

"A design pattern provides a scheme for refining the subsystems or components of a software system, or the relations between them. It describes a commonly - recurring structure of communicating components that solves a general design problem within a particular context" (Buschmann *et al.* 1996).

Design patterns are intended to be independent of any particular language. Their application does not impact the fundamental structure of a software system, but influences deeply the development of the subsystem in which it is applied (Buschmann *et al.* 1996). Design patterns can be applied in the design of the Software Space Plan by specifying detailed design aspects and the implementation requirements of a component. They can also be applied in refining and deciding on the basic module interfaces, following the reuse principle "Program to an interface, not an implementation" (Gamma *et al.* 1994). The aim of decoupling interface from implementation is to simplify the reuse and reorganisation of the Software Space Plan.

As with the case of Space Plan in building architecture, Software Space Plan is the layer that often changes to cope with the needs and desires of occupants. This level is where most systems are designed to evolve, in response to changes of existing requirements or the needs of new requirements. "The space plan is the stage of the human comedy. New scene, new set" (Brand 1994).

3.6 Software Stuff

The Software Stuff layer objective is to represent the actual programming elements that perform or support process, or contain information: functions, procedures, data representations, data structures, etc. In an object–oriented program, classes are defined with an interface controlling access to the data and functions, and an implementation that represents the coding of such data

and functions. Idioms are the Software Pattern approach for describing Software Stuff.

"An idiom is a low–level pattern specific to a program language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language" (Buschmann *et al.* 1996).

Idioms represent patterns for the design and implementation of code that provides specific functionality in a particular language. Idioms are used in the implementation phase to transform a software architecture into a software program written in a specific language. They address problems that arise during the implementation stages and capture the programming experience of previous implementations (Buschmann *et al.* 1996). Software Stuff is precisely the working material of the programmer. It is the layer of software that is continuously evolving and changing. "Stuff just keeps moving" (Brand 1994).

4 SOME COMMENTS AND OBSERVATIONS ON THE ANALOGY

Why is it often simpler to understand architectural properties in building architecture than in software architecture? It is probably because a building is the most commonly human constructed system that we know. A building is a stage for human life. We usually spend all our lives in buildings, and know about their advantages and liabilities, whereas only recently the idea of "inhabiting" software has been considered by the Pattern Community. Analogy is the basis for the discovery of patterns.

Surely, this is not the only description analogy that can be obtained between building and software architecture. Our interpretation of the Layers of Change is not unique. Interpretations depending on other paradigms, techniques and applications can also be proposed. However, as our work is closely related to the area of Software Architecture, Software Patterns, and Object–Oriented Programming, this analogy seems to fulfil our requirements and expectations. Our aim is that the way we use analogy to obtain this interpretation for software design and construction would be useful for others to propose their own interpretations.

The concept of the Layers of Change can be used as an approach to the study, design and construction of systems in general. In the case of software, recognition of the Layers of Change provides a firm basis for understanding how software systems can be changed and modified, while respecting the different ways that different systems evolve. However, our problem now is that only the programmer or designer who wrote a software program can precisely recognize the layers in his/her design and estimate their rates of change. A closer study of the evolution of software is required to understand more about rates of change and how they can be measured and communicated.

REFERENCES

Steward Brand (1994) *How Buildings Learn. What happens after they're built.* Phoenix Illustrated, Orion Books Ltd.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal (1996) *Pattern–Oriented Software Architecture. A System of Patterns.* John Wiley & Sons, Ltd.

Richard Gabriel (1996) *Patterns of Software: Tales from the Software Community.* Oxford University Press.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994) *Design Patterns: Elements of Reusable Object–Oriented Systems.* Addison–Wesley, Reading, MA.

Bjarne Stroustrup (1991) *The C++ Programming Language.* Second edition. Addison–Wesley Publishing Co.