Software Design, a 2011 Primer

Sharon Andrews White, Ph.D.
University of Houston – Clear Lake
whites@uhcl.edu

Muhammad "Tuan" Amith, MS
University of Texas- Health Science Center
Muhammad.F.Amith@uth.tmc.edu

Abstract

This paper examines the state of software design in 2011. It covers the benefits of good design, drawbacks of poor design, overviews the major challenges to good design, and categorizes the major strategies in use for design creation. Known design principles that have held the past 20-30 years are presented as well as the compositional makeup of a design method in general. It also provides a categorization of the modeling approaches in use today as well as a categorization of the types of design methods and an overview of four of the more recent promising approaches to software design.

Keywords

Design Principles, Design Strategies, Design Challenges, Design Qualities, Modeling Approaches

1. Introduction

Software design is the activity between the requirements gathering phase and implementation phase, which results in an abstract model of the system. The design is the representation of how the requirements and domain constraints will be implemented. The resulting design models form the design documents from which the code will be written. These models take many forms and level of detail. It is the job of the software engineer operating as software architect and designer to create these models. The software architect/designer is similar to an architect in building construction, commercial or residential (White 1995; Garland 1996). Both create blueprints, or models, of the final product from which buildings, or software systems in the case of software, are constructed. Recently there have been efforts and arguments made to do away with such modeling. The basic argument taking the form: why design, why not just start coding because is not the code the design itself? Is design necessary when requirements are always changing, when time-to-delivery is important, and when the software is continuously evolving? Most design research, and most academic software engineering

programs, in the past 20 years have been based on the fact that design must be rigorously performed, analyzed, and verified before any coding may begin (Sommerville, 2004). Of late, there has been a move to favor methods that take the middle ground in this argument and favor a symbiotic relationship between design modeling and coding, whereby design and code are much closer to each other (Beydeda, 2005). In addition, a more radical departure from the traditional design phase is the notion of going straight to code and viewing code as being one with the design such that no design modeling is performed (Beydeda, 2005). This paper will present the benefits of explicit design, the process, representation, and heuristics of a design method, an overview of major challenges to good design, and the major strategies used for design creation. The last section of the paper presents a few of the current approaches to design modeling.

2. Good design and poor design

Most traditional engineering fields (Electrical, Mechanical, Computer, Software, etc.) are reliant on design methodologies. Developers use design models to understand what they are building and to aid in the communication between developer and customer. Good design is key to such understanding and communication. The following will review what benefits good design models of a system offer:

- *Good design captures documentation and enables knowledge transfer*. A good design will provide documentation of the system specification to the current and future development team. It will lay out exactly what is to be built, it should eliminate guesswork and over building or under building the product.

- *Good design can communicate with customers and clients*. Like blueprints or small-scale design models for architects, many software design models can simplify understanding of the system to customers and clients, as well as to developers. This allows the system (computer, physical building, bridge, etc.) to be built that is actually needed, rather than providing more, or less, than the customer is asking for.

- *Good design can enable understanding of the product and problem domain*. Good design models can provide a method for software architects and developers (and customers) to identify design constraints on the project well before implementation. Constraints will arise that will affect the way a system (software, home or bridge system) can be built and such must be resolved in order to provide a solution that meets all constraints.

- *Good design can be an economic asset*. Good design can avoid costly mistakes and lost time, and therefore save time, effort, and money. This is directly related to constraint resolutions. Working out the constraints within the design before implementation will avoid implementation rework.

On the other hand, poorly designed software, or "code-only" software (no design at all) can be very costly since time can be lost in reworking and re-implementing fractured portions of a prototype system that needs constant reworking. Such continuous reworking without guidance of an overall design can result in a "hacked" system subject to failure of

performance at worst, and failure of future modification at best, due to a very high probability of the resulting system consisting of highly-coupled tangled code. As a result poor design, or no design, can result in many faults. Some of these are listed below.

- *Poor design can result in failed projects*: Many software projects have failed due to being poorly designed (Evans, 2003; Martin).

- *Delivery of a poor system.* The application may never do compelling things for the user even though the technical infrastructure may work (Evans, 2003).

- *Poor design results in a system that is very hard to change, or rigid* (Karat et al., 2008). A rigid design is hard to change because every change affects too many other parts of the system.

- *Poor design becomes very brittle.* (This has also been described as fragile (Karat et al., 2008)). When you make a change, unexpected parts of the system break. This is obviously caused by a highly coupled design whose parts are very dependent on each other rather than a design that is independent and modular.

- *Poor design is not portable.* It is hard to reuse in another application because the design cannot be separated from the current application and used in a similar design.

3. The types of design strategies employed

A given design method will call for a particular strategy to be used to actually create the design from the specification. The strategy will define the type of activities that will be required to take place during design. We describe below five basic categories of design strategies: Decomposition, Compositional, Organizational, Template-based (Budgen, 2003), and Domain-based (White, 1996; White, 1995; Evans, 2003).

- *Decomposition strategy (aka, top-down approach).* If a design method is based on a decomposition strategy then the design activity will be driven by a process that requires the designer develop the design through a process of subdivision of the problems specified by the requirements into their sub-problems in a top-down manner creating successively more detail, until the entire solution is designed. A classic top-down problem solving approach.

- *Compositional strateg*y (aka, bottom up approach). A compositional design method will be driven by a design process that requires the basic design model be built up from the from its most simple elements, (from its most detailed requirements first) putting sub-models together as the design progresses to form a set of sub-solutions first, putting these together, and continuing in this way until the final design is derived. A classic bottom up problem solving approach.

- *Organizational strategy*. This design process can be heavily driven by a business organizational structure and its practices. This will mean that the design process will contain models and steps that are required to meet non-functional requirements of the organization in addition to the functional requirements of the product. This type of strategy will typically inject organization specific steps for quality assurance, and compliance, as well as prescribed procedures and methods to be applied throughout the underlying design process. This design strategy used may be a blend of top-down and bottom-up along with the use of templates or patterns that may also be organization specific.

- *Template-based* strategy. A design strategy can be based on class of problems that the strategy, or design process, has been designed to solve. This type of strategy or process is embedded with the knowledge of how to build specific types of systems and provides design templates than can be modified and reused in order to meet requirements that are essentially the same as in previous similar systems. Template strategies based on design patterns is one template-based approach. Of course, availability of good design patterns is essential to this approach. Design patterns that are high level and are truly "design" patterns (not just code patterns that are named design patterns) are the key to true template-based design.

- *Domain*-based strategy. This can be seen in work on Architecture Styles (Garland and Shaw, 1993; White, 1996) where a specific design process can be applied to a defined problem domain. Design methods based on this type of strategy can be very powerful since the style definition contains a set of design rules or heuristics that define how problems are successfully solved (designed) in that domain. This fact moves these types of design methods from general design use into the category of specialized design processes and methods to be used for designing within a specific domain (White, 1996; White, 1997; White, 1998). This idea of domain-based design has also been picked up of late by Eric Evans (Evans, 2003).

4. The Software Design Challenge

Frederick Brooks, in his seminal book, the Mythical Man-Month (1995), and Budgen (Budgen, 2003) and many others have made it clear that design is cornerstone of producing a quality product. They, and many others over the past decades, have demonstrated that some of the main challenges to software design are *complexity, conformity, invisibility, and changeability* (Brooks, 1995; Budgen, 2003). An overview of each of these major design challenges below:

- *The Complexity challenge*. Many problem domains tend to be difficult to model due to the complexity of the domain. This challenge requires practices or techniques for good domain modeling that will allow the system to be modeled in its basic form, modularized well, and removing any extraneous domain elements. In addition, complexity is increased if the system is to operate with components that are dissimilar or distributed. This is true even more so today with the inclusion of web-based software and distributed network computing (Budgen,

2003).

- *The Conformity challenge***.** Another well-known software challenge is the notion of design conformity where a design must meet the many conflicting constraints on a system. These constraints can be categorized into three types: organizational constraints, product constraints and process constraints (Sommerville, 2004). Examples of organizational constraints are standardization constraints.

- *The Invisibility challenge.* Invisibility is a long known challenge of software design well documented by Ian Sommerville in the early 1980's. Since the software itself takes no physical form, in order to model it and to manage the process of creating it, testing it, etc, we must create abstract representations of it. Obviously, the better the abstraction provided by the design representation techniques and methods, the easier and higher quality, the modeling, analysis, testing, and other manipulations required will be.

- *The Changeability challenge.* This challenge is also known as modifiability. The design may have to be changed due to incomplete or inconsistent requirements that, left undetected, make their way to a system that is also incomplete or inconsistent and thus must be changed to correct these inadequacies.

- *The Consistency and Completeness challenge*. This challenge refers to consistency and completeness between the requirements, design, and code. In "Software Design in a Postmodern Era" by Philippe Kruchten (Kruchten, 2005), Kruchten refers to similar challenges as *upstream* and *downstream* gaps. These challenges have been well documented in software engineering literature since the 1980's under the realm of requirements analysis. They are actually a sub-problem of the complexity challenge. With upstream gaps, or what we choose to call requirements gaps, the captured requirements and specification do not accurately reflect the user needs and results in incomplete and inconsistent requirements, which will filter down to the code if not corrected. Downstream gaps, or what we choose to call design gaps, are places where the design and code are out of sync. The programmer may add functionality not called for or leave out functionality that has been specified, or the gaps in requirements have progressed unchecked to the coding phase.

6. Current Approaches to Design Modeling

6.1 Categories of Modeling Approaches

A software model is an abstraction of an event, problem, interaction, system, etc. Models are used to capture a problem or solution domain. The information captured by a model can be used to reason about a problem domain or to design a solution in this domain. The newest modeling approaches used in 2000-2010 range from what has been called a "code-only" model category, which essentially means coding without any preliminary modeling, to model-based categories of approaches where a lot of the code is not written but rather is generated from a good model. These approaches are briefly described as:

- *No Modeling* (aka, Code-only (Beydeda et al., 2005)). No modeling is used at all. The solution is coded directly in a 3GL (third generation language) such as Java, or C++ usually within a development environment such as Microsoft Visual Studio. This approach can only be useful to apply to the production of small, basic, well-understood systems that will not be changed or maintained over time.

- *Code embedded models*. Models are direct visualizations of the code module. For example, a diagram of a C++ procedure showing the interface definition and constants, and procedure code highlighted in color to help with debugging. The diagram is on an abstraction level equal to the code level (Beydeda et al., 2005). An example of such is IBM Rational Rose (IBM.com) and open-sourced Eclipse tools dedicated to this type of modeling method.

- *Iterative prototype models*. Architecture and high-level design models are created from successive prototype models, then coded, and this process repeats with changes made to either the design or code. The resulting models typically end up out of step with the implementation (Beydeda, 2005).

- *Transformation-based models*. (Model-Centric (Beydeda, 2005)). Models of the system are developed in sufficient detail that the system can be implemented by applying a set of transformations to the models. The transformation will vary depending on the Model-Centric method. MDA (Beydeda, 2005), Model Driven Architecture is an example of one such method.

6.2 More recent design Methods.

6.2.1 *Service Oriented Architecture*

A service-oriented architecture is an architecture model that describes interactions between service providers, service consumers, and the service registry or service depository. A service is a function that is provided to any consumer of the service (a service is a like a procedure in a procedural based system). A service publisher constructs and maintains a description of the service and provides access to the service implementation. A service consumer can use this service by calling the services identifier (like a procedure call, or remote procedure call), also called the uniform resource identifier (URI), directly or via a third party service registry. The service broker or provider provides and maintains the service registry. This allows a business to extract services from its company and house them in a registry and make them available to future systems. This allows for reuse on a scale not seen recently. The architecture of SOA is based on providing loosely coupled services that are described by a description, an implementation and binding attributes. It has proven to be a very successful model for companies like IBM and TIBCO since it provides an enterprise-scale solution to linking needed services as needed across multiple applications.

6.2.2 *Aspect Oriented Development*.

Aspect Oriented Modeling is viewed as an enhancement to traditional Object Oriented Design and Analysis (OODA), that tries to alleviate some of the non-cohesiveness or "Tangling" (Suzuki, 1999) and loosely coupled design problems "Scattering" (Suzuki, 1999) that OODA fails to address (Whittle, 2008). Aspect Oriented Development deals specifically with *crosscutting concerns*. A crosscutting concern is one, which spans the entire system. Some examples are security, and fault tolerance, which are non-functional requirements, and user-interface and memory management concerns, or non-functional requirements. A crosscutting concern is a concern (or requirement) that spans many components of an architecture, design, or system implementation. An "Aspect" is an abstraction which is viewed as a horizontal slice through a vertical decomposition of the architectural components and the properties relating to the functional and non-functional characteristics of this slice that define the crosscutting concern. It is intended that the separation of concerns afforded by Aspect Oriented Design will result in a reduction in complexity and improved reusability (Filman, 2004). Three Aspect Oriented Modeling approaches are: the Symmetrical "Hyperslice" approach, the Aysmmetrical "AspectJ" approach (Whittle, 2008; Filman, 2004), and the Theme approach (Clarke, 2005).

6.2.3 Component Based Design

The component based development model is similar to the spiral model of development. It is evolutionary and iterative. Software is construction from preexisting commercial components (called COTS, Component Off The Shelf) where possible. The process calls first for identification of available commercial components, integration of components is then considered, an architecture is designed utilizing the components, the system is then development and tested. The availability of proper components to reuse is the limiting factor as well as the ability to modify components for reuse (how well the components have been designed for reuse) (White, 1998). The larger the component to be reused is (the larger the reuse scale) the harder the reusable component (COTS) is to develop for reuse in the first place, thus there are typically not be many large-scale components that can be reused. Mostly it will be small-scale components that will be available for reuse. However, the idea of using existing components to solve common similar problems is very powerful. If a reusable library of useful large-scale parameterized components could be built it offers more promise to producing quality systems faster than most all other design and development methods (White, 1995).

6.2.4 Agile Design Process

Agility, in the context of Software Engineering, has become a buzzword to describe a process that strives to accommodate change: Change to the software itself and change to the product. It emphasizes incremental development that results in a series of rapid delivery prototype-like products that are evaluated and added to over time to develop the final product. Intermediate work products are de-emphasized or ignored depending on the degree of "agility". The perception of many agile advocates is that the code is the design (Amber, 2009; Fowler, 2004). A fundamental principle of an agile process is the early and continuous delivery of working software, where requirements change is welcomed, even late in development. Working software is the primary measure of progress. Documentation and design take a back seat. They are seen to "emerge" from the code and to reside implicitly within the product. Extreme Programming, XP, is an example of an agile process. With XP design is kept simple as possible. The design is simply a set of class-responsibility-collaborator cards. If a difficult problem arises that such cards cannot resolve the problem is prototyped rather than design. No design for future modification takes places. Other agile methods have modeling included, most notably Agile Model Driven Development/MDA (Mellor, et al., 2004).

7. Conclusion

The software landscape changed gradually over the last 30 years with respect to design tools and methods. During the 70's software development was dominated by procedural and functional development techniques. Then the 80's brought Object Oriented development, and OO design methods, tools, and languages. The 90's saw the development of service oriented architecture, and associated distributed design methods and tools for distributed systems serving multiple clients. The 2000 decade saw the rise and acceptance of Service Oriented Architectures, Aspect Oriented Development, Domain Driven Design, and Agile Process. During the most recent years design has evolved to more procedural based designs that are highly reusable (e.g., in service oriented architectures), to designs which try and capture complex non-functional slices of cross cutting concerns (aspects), and to the extreme of throwing design away in lieu of rapid code-delivery via prototyping and incremental development to accommodate change (as with an Agile Development Process). All of these approaches have shown their usefulness in the right environment.

This paper reviewed the qualities and challenges of design, overviewed the basic categories of design modeling strategies and methods including presented a brief discussion of the recent trends in software design in an effort to provide the reader with a primer on the state of software design as of 2011.

References

Ambler, Scott. "Agile Design." 2009. Ambysoft. 13 March 2010.

Amber, Scott. “Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development.” 2009. Ambysoft. 13 March 2010.

Beydeda, Sami; Book, Matthias; and Gruhn, Volker. “Introduction: Models, Modeling, and Model-Driven Architecture (MDA).” Model Driven Software Development. New York: Springer Berlin Heidelberg, 2005.

Brooks, Frederick. Mythical Man-Month, 2nd Edition. Upper Saddle River, NJ: Addison-Wesley Professional, 1995.

Budgen, David. Software Design, 2nd Edition. Pearson Education Ltd., 2003.

Clarke, Siobhán and Baniassad, E. Aspect-Oriented Analysis and Design: The Theme Approach. Upper Saddle River, NJ: Addison Wesley Professional, 2005.

Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston, MA: Addison-Wesley Professional, 2003.

Filman, Robert E., Elrad, Tzilla, Clarke, Siobhán, Aksit, Mehmet. Aspect-Oriented Software Development. Boston, MA: Addison Wesley Professional, 2004.

Fowler, Martin. “Is Design Dead?” May 2004. martinfowler.com. April 2010. http://martinfowler.com/articles/designDead.html

Garland, David and Shaw, Mary. “An Introduction to Software Architecture”, Technical Paper CMU/SEI-93-TR-33, Carnegie Mellon University, Software Engineering Institute. 1993.

Garland, David and Shaw, Mary. “Software Architecture” Perspectives on an Emerging Discipline, Prentice Hall, 1996.

Suzuki, Junichi and Yamamoto, Yoshikazu. “Extending UML with Aspects: Aspect Support in the Design Phase.” 3rd Aspect-Oriented Programming Workshop at ECOPP’99, 1999.

Karat, John and Vanderdonckt, Jean. Web Engineering- Modeling and Implementing Web Applications. Springer-Verlag, London, 2008.

Kruchten, Philippe. “Software Design in the Postmodern Era.” IEEE Software. March/April 2005.

Martin, Robert. “Dependency Inversion Principle.” http://www.objectmentor.com/resources/articles/dip.pdf.

Mellor, Stephen J., Scott, Kendall, Uhl, Axel, and Weise, Dirk. MDA Distilled:

Principles of Model-Driven Architecture. Boston, MA: Addison-Wesley, 2004.

Rational Software. IBM. 6 June 2010. http://www-01.ibm.com/software/rational/.

Sommerville, Ian. Software Engineering, 7th Edition. Pearson Education, Ltd., 2004.

White, S. A. "Software Architecture Design Domain." Proceedings of the second Integrated Design and Process Technology Conference, Vol 1. pp. 283-290, 1996.

White S. A. and C. Lemus, "Architectural Reuse in Software Development", Proceedings of 20th International Computers in Engineering Symposium (ASME-ETCE98) Jan. 1998, pp. 1-8.

White, S. A. and Lemus C., "Architecture Reuse through a Domain Specific Language Generator" position paper in Proceedings of WISR8: The eight annual workshop on Software Reuse, pp. White-S-A-1 : White-S-A-6, March 23-26, 1997, Ohio State University.

White, S. A. "Software Architecture Design Domain", S. A. White, Proceedings of the second Integrated Design and Process Technology Conference (IDPT'96), Vol 1. pp. 283-290, Austin, TX., Dec. 1-4, 1996.

White, S. A. "A Design Metalanguage for Design Language Creation" , S. A. White, Proceedings of ASME Symposium on Computers in Engineering, (ASME-ETCE'96 conference) Houston TX, Jan. 29 - Feb 2, 1996, pp.135-144.

White, S. A., "A Framework for the development of Domain Specific Design Support Systems", Proceedings of the First World Conference on Integrated Design & Process Technology (IDPT), Dec. 6-9, 1995, Austin, TX., IDPT-Vol.1. pp. 37-42.