

Designing for Change

Peter Grogono
Department of Computer Science
Concordia University
1455 de Maisonneuve Blvd West
Montréal, Québec H3G 1M8
grogono@cs.concordia.ca

Abstract

The ambition of every designer is the software equivalent of a cathedral. But maintenance programmers are more comfortable in a farmhouse than a cathedral. We argue that current design methodologies are oriented towards cathedrals, and we propose object oriented design techniques and tools that are suitable for farmhouses.

During the lifetime of a useful program, its users' requirements change and the code changes to track the requirements. The code drifts away from the original design, becomes increasingly brittle, and eventually can no longer be maintained; each repair introduces new faults. The cure for these ills—design for change—is well-known, but current design methodologies and tools do not facilitate useful changes.

We describe a design tool that supports evolutionary object oriented design. Designers can create and modify designs, view them in textual and graphical form, check their internal consistency, and match them to requirements and code.

To accomplish this, we use text, tables, and diagrams with multiple levels of formality. The tool processes *formal* entities completely (as a

compiler can process source code completely); it stores, retrieves, and displays *informal* entities (whereas a compiler discards comments); and it can perform limited operations on *semi-formal* entities. Our work borrows from formal specification techniques, but is intended for software that evolves.

1 Introduction

Some of the most popular and successful programs and systems are those that have evolved over a period of years. Commercial packages with version numbers such as 3, 4, 5, and more are common. The software grows from simple and modest beginnings, expanding and improving to meet the increasingly complex requirements of an ever increasing number of users. For the programmers working on it, the software resembles an old, rambling farmhouse; a bit cluttered, perhaps, but comfortable and manageable.

Textbook techniques for designing software stand in direct contrast to this view of development. The process of requirements, design, and implementation seems more suited to the construction of cathedrals [3, page 40] than to farmhouses. The requirements of a cathedral do not change over time, and cathedral main-

tenance is directed towards keeping the structure standing rather than adapting it to changing user needs.

Our methods for software design should reflect our successful practice rather than an unachievable ideal. “Design for change” and “code re-use” are not new ideas, but they have been grafted onto traditional development methods rather than being incorporated into the roots of new methods.

Any software development method for non-trivial software is a process that starts with customer requirements and ends with working code. Design, the focus of this paper, is a process that takes requirements as input and yields a specification from which competent programmers can efficiently create robust and reliable code. We make two assumptions that reflect typical situations:

- ▷ the code will be developed incrementally, as a sequence of deliverables that perform a subset of the overall requirements; and
- ▷ the requirements will change as the software increments become available and the customers recognize their mistakes and discover new possibilities.

Figure 1 shows the ideal situation at the time of first release of the complete system. The requirements, design, and code are fully consistent with each other.

Figure 2 shows a typical situation some time later. The client has used the system and modified the requirements. Maintenance programmers have responded by changing the code to meet the new requirements, without changing the design.

As the code drifts away from the design, it becomes increasingly “brittle”—every new change seems to break some other part of the code. One way of avoiding this situation is to keep the requirements, design, and code consistent with one another, as the software evolves.

All of these considerations point in the same direction: designs must be “lightweight”—easy to change and cheap enough to throw away when they are no longer useful. If designers are to use computer-aided design tools, the

tools must contribute to speed and flexibility in the design process.

2 Aspects of Design

A design methodology must respect the range of user requirements on one hand and current implementation practices on the other. Some aspects of design change relatively slowly; others change as new paradigms and practices emerge. In this section, we review some recent developments that affect the design process.

2.1 Object Oriented Design

Software bridges the gap between the “real world” of the problem domain and the “formal world” of the computer that executes the programs. Software that is close to the machine is likely to be efficient but hard to maintain. The modern trend is towards the use of high level languages and other techniques for moving the software closer to the problem domain. The price paid for this move is a loss of efficiency at first, but this loss can be offset to some extent as more sophisticated compilers become available.

Object orientation has contributed to the movement towards the problem domain by introducing the *object* as the primary abstraction and the unit of computation. Objects in the problem domain are mapped to “software objects” in the solution domain. Actions in the problem domain are simulated by messages exchanged by the software objects. (The term “message”, introduced with SmallTalk [8], is used metaphorically. Messages in object oriented programs are procedure calls with a small overhead caused by dynamic dispatching.) Design methodologies based on these ideas have been proposed by a number of researchers [2, 5, 6, 7, 14, 16, 17, 19]. The number of superficially similar notations with common underlying concepts suggests an evolving paradigm that will require several years to reach maturity.

The characteristics of object oriented programming that affect design include the fol-



Figure 1: The original situation

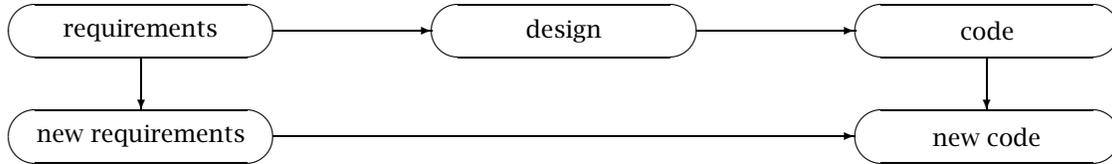


Figure 2: After some changes

lowing.

- ▷ Focusing attention on objects rather than on functions has a profound effect on the design process.
- ▷ Inheritance, appropriately used, facilitates incremental design and development.
- ▷ Dynamic binding encourages a flexible development style in which code can be written and executed without final commitment to representations.

Object oriented design and functional design are different paradigms. Combining them into a single methodology, as has been suggested [18], is not feasible because they work in opposite directions. Object designs derive their robustness by being matched to the application domain rather than to any particular problem [16]. Any workable object oriented design methodology must respect this difference.

Current approaches to object oriented design have been criticized on several grounds. The close correspondence between the problem and implementation domains, one of the alleged advantages of the method, is seen by some as a weakness. Høydalsvik and Sindre point out that matching real-world objects to computational objects brings concerns that belong to implementation back into the analysis and design phases [13]. Haythorn argues that conventional object oriented methods may lead to software that is no easier to maintain than software design by older, structured techniques [12]. Although these criti-

cisms cannot simply be dismissed, they are symptomatic of a paradigm shift [15]. In moving to object orientation, we must shed some of the wisdom of earlier paradigms.

Aksit and Bergmans describe a number of problems that they encountered while applying published design techniques [1]. Several of the problems that they report can be attributed to inadequate design methodologies: difficulties arise in problem decomposition, identifying subsystems, conflicts between inheritance hierarchies and subsystems, and so on. Although their study shows that better techniques are needed, they do not suggest abandoning the object oriented approach.

2.2 Formal Methods

The variety of superficially similar diagramming techniques reveals the immaturity of object oriented design methodology. Similarly, the plethora of logics, advertised by tedious proofs of simple programs, suggests the immaturity of formal approaches to software design.

A formal system has three components. The *syntax* describes the structure of well-formed formulas; the *semantics* ascribes a meaning to each well-formed formula; and the *inference rules* (of which axioms are a limiting case) are syntactic transformations that maintain meaning. Formal system design involves trade-offs: logicians and software designers do not have the same requirements.

For software design and programming, formal systems are interesting because they provide opportunities for automation. The first two components of a formal system, syntax and semantics, are useful and often sufficient for the early stages of software development. Inference rules are required in the final phases for verifying specifications.

The process of formalization is as useful—sometimes even more useful—than the formal system that it produces. Formalizing forces us to consider every component and every interaction of the system in complete detail. Before we can formalize, however, we must have something to formalize. “Good design” means making creative decisions, based on skill and experience, that can later be formalized. It is this, earlier, aspect of design that is the focus of this paper.

2.3 Diagrams

Many of the popular design techniques are based on a pictorial notation. Pictures are useful because they efficiently communicate information about the overall structure of a system. Pictures are an efficient form of informal description, especially when they are associated with a more formal representation in another medium. This is why offices and classrooms have whiteboards and speakers at conferences use overhead projectors. But pictures lose some of the effectiveness when they become constrained—all structure charts tend to look the same—or cluttered with a plethora of symbols. Or when they are hard to draw.

Diagrams are not yet an efficient way of communicating with a computer. Diagrams help us to think because they can be sketched quickly and without concern for the *process* of drawing. Modern graphical user interfaces are powerful, but they cannot yet provide the required transparency. As Yourdon points out, CASE tools move the emphasis from the user’s application to “the artistic elegance of the diagrams drawn by the CASE tool” [20]. Although computers can generate a visual representation of data that may help a person, they are not yet capable of extracting useful informa-

tion from a rough sketch—although we may not have long to wait for this capability. Current state of the art favours designs based on textual input. Software tools can present the text in various ways: as text, hypertext, tables, and pictures.

3 The Design Format

A design incorporates three models of the desired system [16].

- ▷ The *functional model* describes the computations performed. A computation may be carried out by a single object or by an ensemble of objects working in harmony.
- ▷ The data or *object model* describes the way in which data are represented and managed within the system. In an object oriented system, each object encapsulates a component of the data; objects may be parts of other objects.
- ▷ The *dynamic model* describes the flow of control within an object and between objects.

Our design notation does not explicitly differentiate between these models. It is organized around the object model; a design is essentially a collection of object (actually class) interfaces. The functional model can be extracted from the object model by examining descriptions of methods. In most cases, there will be a single method that is responsible for a computation, even if the actual computation is distributed over several objects.

The dynamic model cannot, in general, be extracted from the design and must therefore be described separately. Developing methods for describing the dynamic model is one of our current activities. Although state-charts [11] are popular for dynamic modelling, other methods may be more suitable for object oriented systems. We are currently considering time-threads [4] and use cases [14].

A design, then, is a collection of classes. Classes may be independent, but are usually components of subsystems, frameworks, and inheritance graphs.

A *class* may be described at any level of detail. The minimal description of a class provides just its name. A full description includes the relation of the class to other classes, the instance variables of the class, and a description of each method of the class. A method has a name and a signature and may also have a specification and a list of the methods that it may invoke. Obviously, the amount of useful feedback that the tool can provide to the designer increases with the amount of information that the designer provides. Nevertheless, the ability of the tool to process partial designs is crucial to its flexibility.

The text provided by the designer has three levels of formality [9].

- ▷ The first level is *informal*. Each feature of the design may be decorated with a comment. Comments are not discarded: the tool maintains the connection between feature and comment and displays the comments on request.
- ▷ The second level is *formal and syntactic*. Declarations of classes, relationships between classes, instance variables, and methods are at this level. Formality enables the design tool to identify inconsistency and incompleteness in the design.
- ▷ The third level is *formal and semantic*. The text consists of formulas of a suitable logic and is used to specify methods and classes.

The extent to which the tool can process text depends on the level of the text within this hierarchy. Informal text can be re-arranged—for instance, comments can be copied from a base class to derived classes—but not processed in any useful way. The tool can apply syntactic transformations and simple semantic checks to text at the second level. Processing text at the third level requires a theorem prover. We do not intend to incorporate a prover into the design tool, although the tool might generate proof obligations for another tool.

4 A Design Tool

“Lightweight” designs of the kind proposed in Section 1 cannot be printed documents. We do not maintain source code in printed and bound books, and it is just as absurd to maintain designs in this way. A *design tool* is a software system, or component of such a system, that provides facilities for creating, examining, checking, and modifying designs. All relevant information about the designs is stored in a database that is usually called a *repository*. What would we expect of an ideal design tool? The tool should:

- ▷ read, write, and display designs;
- ▷ provide multiple views of a design and provide facilities to change the design from any view;
- ▷ check the consistency and completeness of the design upon request;
- ▷ respond to queries about the design, especially “what if?” queries;
- ▷ generate high-quality printed reports for archiving;
- ▷ provide traceability to both requirements and code.

We are currently building a design tool with features that we believe will meet the requirements outlined above. The tool is a continuation of previous work in the development of object oriented software [10]. A browser that provides abstract views of source code is essential for the development and maintenance of object programs. The design tool is a browser in reverse: it maintains the abstract views before the code exists.

The design tool transforms the text into a directed graph that can be stored in memory or on disk. The principal purpose of the tool is to display the design in ways that permit the designer to explore and modify it as easily as possible. To this end, it provides several kinds of display.

The first form of display is *text*, essentially as entered by the designer, but pretty-printed. The tool adds some information to the original text, mostly by performing closure operations on relations. The designer can select the

level of detail so that the display is suited to the current task. A key goal is to minimize clutter: the tool suppresses redundant information and noise.

The second form of display consists of *tables*. Tables have several advantages over bubble-and-arc diagrams in a human-computer interface.

- ▷ Tables are easy to create.
- ▷ Large tables, like spreadsheets, are easy to navigate by simple scrolling operations.
- ▷ Tables use make efficient use of precious screen space. Microscopic fonts are not required.
- ▷ Small changes to the design tend to cause small changes to tables. In contrast, diagrams that alter in shape and layout after a small change waste time and cause confusion.

The third form of display consists of *diagrams*. The disadvantages of diagrams that we discussed in Section 2.3 are not sufficient to eliminate diagrams altogether: our point is that diagrams should not be the only, or dominant, form of presentation. Diagrams are derived from text and are used to provide quick confirmation that the expected relationships are present.

In addition to screen displays, the tool can output designs in a form suitable for printing. We have found T_EX to be very suitable for this purpose: it is straightforward to generate T_EX code, and the printed output is pleasing to the eye. Although the development of a design is best done interactively, leisurely study of hard-copy often reveals areas for improvement.

Our principal goal in building the tool is to minimize the input provided by the designer while maximizing the output generated by the tool. The means by which we accomplish this goal, which is essential to our concept of “lightweight” design, include the following.

- ▷ Minimize redundancy in the input. The designer should never have to enter the same information in different places or different ways.
- ▷ Maximize the inferences performed by the tool. For example, the tool can flatten in-

heritance graphs, enabling the designer to see the consequences of inheritance without exploring the graph.

- ▷ Perform transitive closures. These enable the designer to determine the consequences of a change, for instance, with a minimum of effort.

5 Conclusion

Early design decisions are often made by an individual on the back of an envelope or by a team clustered around a whiteboard. Automating the design phase will not succeed unless the design tools respect the essential spontaneity of the early phases of design.

Late phases of the design process require careful scrutiny because the design must be checked for consistency and completeness. In these phases, the design tools must do as much checking as is feasible and generate the information that designers require to complete the checking.

During all phases, the design tools must support the concept that a design is a continuously evolving entity, in which any part may change at any time.

We do not expect our tool to have industrial strength, at least in its early versions. As a university research team, our job is to experiment with prototypes, hopefully pointing the way to solutions of industrial-scale problems.

Acknowledgements

I would like to thank Patrice Chalin and T. Radhakrishnan for many hours of stimulating discussion—and, at the same time, to absolve them from responsibility for any of the mistakes here. Hanwei Ding and Tom Klemola are currently building the first version of the design tool described in this paper.

About the Author

Peter Grogono worked for fifteen years in the software industry before joining the Department of Computer Science at Concordia University in 1984. His interests include the de-

sign and implementation of programming languages as well as software engineering.

This paper was presented at a workshop on software maintenance at CASCON, Toronto, Ontario, November 1994.

References

- [1] Mehmet Aksit and Lodewijk Bergmans. Obstacles in object-oriented software development. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 341–358, 1992.
- [2] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.
- [3] Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1978. Anniversary Edition, 1995.
- [4] Raymond J. A. Buhr and Ronald S. Caselman. Architectures with pictures. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 466–483, 1992.
- [5] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, 1990.
- [6] Derek Coleman et al. *Object-Oriented Development: the Fusion Method*. Prentice Hall, 1994.
- [7] Dennis de Champeaux, Douglas Lea, and Penelope Faure. *Object-Oriented System Development*. Addison-Wesley, 1993.
- [8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [9] Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.
- [10] Peter Grogono and Benjamin Cheung. A semantic browser for object oriented program development. In *Proceedings 25th Hawaii International Conference on System Sciences, Volume II*, pages 38–45, Kauai, Hawaii, January 1992. IEEE.
- [11] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:213–274, 1987.
- [12] W. Haythorn. What is object-oriented design? *Journal of Object-Oriented Programming*, 7(1):67–78, 1994.
- [13] Geir Høydalsvik and Guttorm Sindre. On the purpose of object-oriented analysis. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 240–255, 1993.
- [14] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.
- [15] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago, 1962.
- [16] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [17] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, 1988.
- [18] A. Wasserman, P. Pircher, and R. Muller. The object-oriented structured design notation for software design representation. *IEEE Computer*, 23(3):50–63, March 1990.
- [19] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.
- [20] Edward Yourdon. *Decline and Fall of the American Programmer*. Yourdon Press, 1992.