Chapter 15

Software Design

CONTENTS

This page intentionally left blank.

Chapter 15

Software Design

"There are very good reasons for everything they do. To the uninitiated some of their little tricks and some of their regulations seem mighty peculiar…" – Eric Frank Russell – "Men, Martians and Machines"

15.1 Introduction

The first digital computer, ENIAC, was constructed in the mid-1940's and became operational in 1945. It weighed over 30 tons, contained 19,000 vacuum tubes and 1500 relays, and used a little under 200 kilowatts of electricity. Its clock cycled at 100 kilohertz and it could multiply 10 digit numbers 384 times a second. [1] In less than the lifetime of a man, ENIAC has evolved into a technology base where computers pervade our existence, being found in appliances and machines affecting virtually every aspect of our lives. This paragraph is being written on a digital computer which can be operated on a human lap, consumes less than 150 watts, and can perform tens of millions of multiplications a second. The physical side of the computing has gone from tubes to transistors to integrated circuits to Very Large Scale Integration (VLSI), and continues unabated in its headlong rush to make everything ever smaller and ever faster.

During this same period the initially experimental craft of programming digital computers has gone through several major revolutions and countless innovations to become a science requiring a lifetime's study to fully comprehend. Any photograph of the current state of computer programming will have become a page in history by the time it is developed. Even the meaning of the sentence you just read will become archaic, if not undecipherable, within a few years because of the convergence of computing and photography. And so it continues.

Computer software began as machine-specific binary code instructions, or ones and zeroes. Thus the first generation of software and programmers were artists as much as anything else, creatively trying to write programs small enough to fit into the limited memory, and fast enough to be useful with the limited processing power. Second generation programming took a step away from ones and zeroes to use mnemonics, special words that represented binary code instructions, to write their programs. Third generation programming incorporated procedural languages in which a single word might be translated into hundreds or thousands of binary instructions. Fourth generation languages attempted to move programming from the realm of telling the computer how to do something, to telling the computer what you want done and letting it write its own procedural program. Fourth generation programming has not been fully successful and software is still generally written in procedural languages with some fourth generation additions and other enhancements. Software development is one of the fastest evolving fields of engineering there are. To cut development costs and time new methods and processes are constantly being created. Software development has become so progressive that its innovations are often applied to other disciplines to help them deal with their own changes.

In spite of all this talk of computer programming, the major work of programming a computer is not programming, or writing software code, but software design. Software is now usually very complex in nature and often consists of multiple programs and resource files. To create good software requires discipline, training, creativity, a lot of work, and good software design processes. This chapter examines the overall software design process and samples some of the design methods used within that process.

15.2 Process Description

Software design is part of the systems development life cycle. Remember that most projects include the development of other system elements in addition to software. Software development rarely has the luxury of proceeding independently of other development, but is constrained to some extent by the other elements and the system as a whole. Software development also has its own life cycle (see Chapter 2) that further constrains the software design effort, determining what methods can be used and how they can be used.

What is presented here as a software design process is a generalized form that will have to be tailored to conform to the various life cycles involved. Additionally, new development methods are evolving to match various types of development with more streamlined and efficient development. While some of these are presented here, the list is incomplete and only representative.

15.2.1 Design Process

The software design phase falls between requirements definition and integration. As shown in Figure 15-1, the inputs to the software design process are the software requirements. The software developed during the design phase is passed on to the integration and test phase. Requirements, integration and test are covered in other areas of this book (Chapters 4, 10, 12, and 14). The design process involves an iterative series of progressively more detailed phases that eventually produce coded software modules ready for integration. Each of these phases is further elaborated below. Remember that there may be more activities or additional iterations of the three shown depending on the development life cycle used. For example, the spiral development model (Section 2.2.1.4) will go through this process two or more times to arrive at the final product.

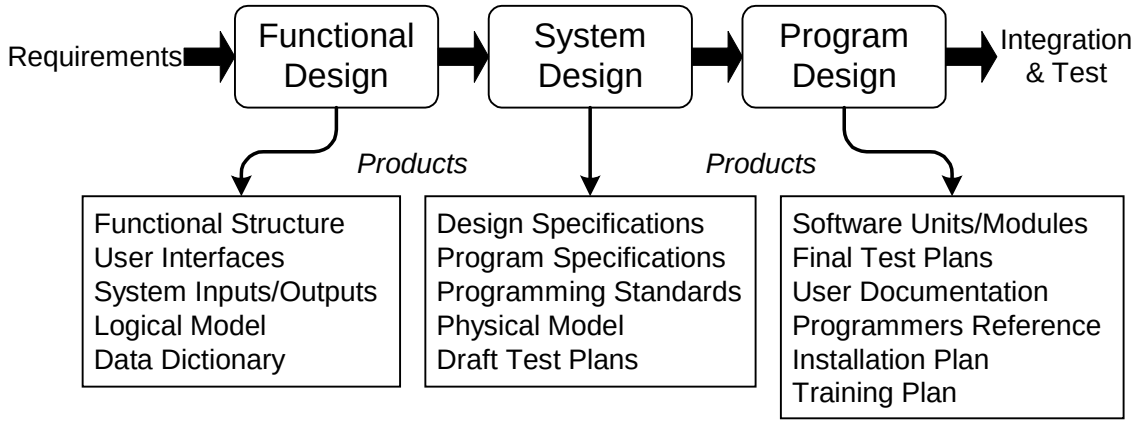

Figure 15-1 Software Design Activities

Note: Functional design and system design may be called by other names, such as preliminary and detailed design.

15.2.1.1 Functional Design [2]

The functional design phase converts the Software Requirements Specification, which tells what the software must do, into a functional design specification, describing how to do it. The functions and structure of the software are defined, including:

- Logical System Flow
- Data Organization
- System Outputs
- System Inputs
- Processing Rules
- Operational Characteristics (User's Viewpoint)

While the focus is on the functional structure of the software, prototyping is often used during this activity to demonstrate the functional design to users and other stakeholders.

The following major activities are performed during functional design:

- Define Software Structure
- Define Content Of System Inputs
- Design User Interfaces
- Define Content Of System Outputs
- Design System Interfaces
- Design System Security Controls
- Build Logical Model
- Build Data Model

- Develop Functional Design
- Conduct Structured Walkthroughs
- Procure Hardware And Software
- Conduct Functional Design Review

The products of the functional design phase include:

- Functional Design Document
- Data Dictionary
- Expanded Requirements Traceability Matrix
- Logical Model
- Design Records
- Hardware And Software Procurement Records

15.2.1.2 System Design [3]

The system design phase converts the user-oriented functional design into technical, computer-oriented, detailed system design specifications. Individual software modules, routines, processes, and data structures are defined, while maintaining the interfaces defined in the functional design phase.

The major activities of this phase include:

- Select System Architecture
- Develop System Design
- Develop Program Specifications
- Develop Conversion Plan
- Develop Integration Test Plan
- Conduct Structured Walkthroughs
- Design Software Module Specifications
- Develop System Test Plan
- Define Programming Standards
- Conduct System Design Review
- Design Physical Model And Database Structure

The products of the system design phase include:

- System Design Document
- Design Specifications
- Physical Model
- Draft Integration Test Plan
- Conversion Plan
- Program Specifications
- Programming Standards
- Expanded Data Dictionary
- Draft System Test Plan
- Expanded Requirements Traceability Matrix

15.2.1.3 Program Design [4]

The function of the program design phase is to produce the actual working software modules specified in the System design phase. The program specifications from the system design phase are used to design and code the individual program modules. Each module must conform to the design documents produced in earlier phases and be thoroughly tested in readiness for integration and testing. Additional activities in this phase include documentation and test planning.

The following activities are included in this phase:

- Write Programs
- Generate Operational Documents
- Develop Training Program
- Plan Transition To Operational Status
- Conduct Unit Testing
- Conduct Structured Walkthroughs
- Establish Programming Environment

The products of this phase include the following:

- Software Units And Modules
- Final Integration Test Plan
- Project Test File
- Users Manual
- Training Plan
- Installation Plan
- Final System Test Plan
- Transition Plan
- Programmers Reference Manual

15.2.2 Design Methods

Originally, almost anything went as far as programming methodology was concerned. Programmers all had their own styles and idiosyncrasies. The problems associated with this *laissez faire* approach began to surface when programmers had to work together on larger, more complex projects. Without a common design method, they had problems with module boundaries, interfaces, data structures, and the list went on. The problems became insurmountable during the maintenance phase when software had to be updated and the original programmer had moved on. It was often easier to build new software than to update the previous version. To help overcome these obstacles, design methodologies were developed and implemented throughout the industry. While the level of success of these methods varies, it is a fact that implementing design methods is far better than not using them.

Design methods are generally used within the framework of the design process discussed in the previous section. The method and/or the process may be modified to combine the two. There is not enough time or room here to make an in-depth study of all the design methods that are available or in use. The three presented here are considered with the briefest regard. They were chosen because they represent an evolution in software development methods. Learn the essential principles of the methods being used on your project. Know their advantages and disadvantages and why they were chosen.

15.2.2.1 Structured Design

Structured design was a major step toward taming the Hydra of spaghetti code programming. It imposed various rules for software design that made it much easier to document, follow, maintain, and in spite of the complaints of some programmers, even easier to design. Two major rules of this method were that (1) programs were to be broken into functions and subroutines, and (2) there was only a single entry point and a single exit point for any function or routine. This imposed an order heretofore unknown in software development and was a major step in making it into a discipline. While other methods have provided even greater order and capacity, most software code is still structured at its lowest level.

15.2.2.2 Object Oriented Design

While structured design had vastly improved the software industry, there were still problems. A change in one part of the program often required thorough searching of the entire program to determine what effects the change had caused in other routines. A simple design change in a single subroutine could ripple throughout the whole program. Code that affected a single decision might be spread throughout the program. Additionally, variables and program structure were accessible to other developers. It wasn't so much a problem of multiple people being able to affect and modify all the code, as it was that those other people *had* to be mindful of what was going on in subroutines throughout the program so they wouldn't inadvertently interfere with them. And there were other issues.

Object Oriented Design (OOD) was developed to overcome many of the problems left unsolved by structured design. It introduced a whole new way of looking at software and currently forms the basis of most new software development. In OOD, a model of the real world is developed to characterize and test the design before having to build it. The model is populated by *objects*, representations of real world objects that have attributes and functions. OOD also implements *encapsulation*, where data and other information are hidden within objects. Other developers can use an object without having to worry about the data inside or how the object operates. One only needs to know what goes into the object, what comes out, and what operations it performs. Another concept implemented under OOD is *inheritance*, where data and objects can be created from previously defined parent data and objects, and have all the attributes of those parents. A third concept, *polymorphism*, allows multiple objects to be modified

through the modification of a single entity. These three concepts greatly enhance developers' abilities to more easily manage multiple objects, reuse and maintain software, and deal with complex software systems.

15.2.2.3 Extreme Programming [5] [6]

A relatively new method of programming is currently making inroads in many small to medium size software development efforts where requirements are vague or rapidly changing. Known as Extreme Programming (XP), it views risk as the basic problem of software development and implements "common sense" approaches to overcome various risks inherent in design. Table 15-1 lists several common risks and their XP solutions.

Table 15-1 How XP Deals With Common Risks

Risk	XP Recommended Mitigation
Schedule slips	Implement short release cycles so that the scope of any slip is limited. Implement higher priority features first so features that slip past the release date are lower value.
User mission misunderstood	The user is an integral part of the team. The specification is continuously refined during development with both developer and user learning.
Mission changes	Release cycles are shortened so there is less change during the development of a single release.
Software rich with unneeded features	Features are implemented in order of their priority (value).
Staff turnover	Programmers are given responsibility to estimate and complete their own work. They receive feedback to improve their estimates. Communications between team members is encouraged to reduced feelings of isolation.
High defect rate	Software is tested from both the programmer's and user's perspectives, function by function, and program feature by program feature.
Project is cancelled	The user chooses the smallest release that makes the most sense mission-wise so there is less to go wrong and software value is the greatest.
Software system goes bad	A comprehensive set of tests is created and maintained, and then run multiple times after every change to ensure a quality baseline.

Extreme programming gets its name by doing good things to extreme levels. Because code reviews are good, code is reviewed all the time. Because testing is good, everyone tests all the time, even the users. Because architecture is important, everyone defines and refines the architecture all the time. If short iterations are good, make them very short, even hours or minutes instead of weeks, months, or years.

Obviously, XP is not applicable to all projects, but it can be successful in the proper setting. And if XP may not be right for a specific project, some of its principles and strategies may be. It also shows us one of the directions software development is heading.

15.2.3 Other Design Considerations

15.2.3.1 Programming Guidelines

The software development guidelines for your project should be documented in a Software Standards and Conventions Document (SSCD). This document defines the following:

1. Overall software development process, including each of the design phases.
2. Programming language quality, style, and standards guidelines.
3. Documentation Standards.
4. Guidelines for use of design tools.
5. Reuse strategies.

6. Software configuration control.
7. Test standards.
8. Review and inspection processes.
9. Metrics to be used.

15.2.3.2 Reuse

Because of the high cost of software development, considerable consideration should be given to designing software that is reusable wherever possible. Designing for reuse begins in the planning phase and must be considered throughout the project to be effective. Proper implementation of reuse techniques and modern design methods, such as OOP, will make maintenance easier and less costly, and can even provide software modules for other projects.

15.2.3.3 Computer Aided Software Engineering (CASE)

CASE tools are software programs that assist the software engineer in designing and documenting software. They are usually dependent on the design methodology employed by the development team, and may even be dependent on the language chosen for implementing the design. Their chief contributions consist of the following advantages:

- Encourage or enforce a formal design process or methodology.
- Document the design in a consistent, formal manner.
- Track the details to help ensure things aren't forgotten.
- Unify the development team in their efforts.

In addition to the above advantages, some CASE tools help in discovering errors, developing tests, tracking requirements, and simulating the software design. Every effort should be made to select an easy-to-use, functional, helpful, and proven tool that not only integrates with your development process, but actually facilitates, assists, and documents that process in as many areas as possible.

15.2.4 Example Design Process

The following example is a process currently being used in the development of an upgrade to an Operational Flight Program (avionics) for an aircraft. The original software already exists so this upgrade will provide additional functionality. The project uses object oriented programming, and is implemented in Ada. There are three design phases, identified as Preliminary, Detailed, and Implementation.

15.2.4.1 PRELIMINARY DESIGN

a. **Review and Allocate Requirements** - Review all new requirements and allocate them to objects, creating new objects as needed.

b. **Create Description of Objects –** Describe each new object at a high level.

c. **Identify Object Associations** – Identify the associations between objects. This is often done by identifying the messages that flow between the objects and using the flows to identify associations.

d. **Allocate Objects to Subsystem –** Allocate all objects to subsystems in the existing architecture.

e. **Identify Object Attributes –** Define the visible attributes (data portion) of each object.

f. **Identify Object Operations –** Define the visible operations of the object.

g. **Develop Static Structure –** Static structure is developed using Class/Object diagrams in a CASE tool.

h. **Develop Scenarios –** High-level scenarios (sequences of events and responses) are developed for significant events.

15.2.4.2 DETAILED DESIGN

a. **Translate Object Diagrams into Ada** – Translate each Class/Object diagram into an Ada package specification.

b. **Refine Ada packages using Program Design Language (PDL)/Ada –** PDL/Ada is a design methodology that combines Ada language control structures with commentary, allowing evaluation of a design before it is coded. PDL/Ada consists of block comments and only those Ada control statements sufficient to describe the program structure. The following components are included:

- **Block Comments** – These describe processing to be performed and are written in English.
- **Program Structure** – When necessary, show program structure with Ada control statements.
- **Data Definitions** – Define data to be used. Include name, description, units, and range.
- **Object Interface Definitions** – Details of object interfaces are defined and documented.

15.2.4.3 CODING

a. **Coding –** Ada packages are coded according to the specifications of the Detailed Design.

b. **Unit Testing –** Program units are tested to ensure they perform correctly against scenarios before being turned over to integration.

15.3 Software Design Checklist

This checklist is provided to assist you in understanding the software design issues of your project. If you cannot answer a question affirmatively, you should carefully examine the situation and take appropriate action.

15.3.1 Before Starting

- ❑ 1. Do you have a well-documented software development process?
- ❑ 2. Do you understand what is to be performed and produced in each phase of the design process?
- ❑ 3. Do you have a Software Standards and Conventions Document (SSCD)?
- ❑ 4. Does the SSCD contain direction in those areas listed in Section 15.2.3.1?
- ❑ 5. Are you familiar with the methods, tools, standards, and guidelines in the SSCD?
- ❑ 6. Are applicable and efficient design methods (OOD, etc.) being implemented on your project?
- ❑ 7. Are the developers experienced in the chosen development process and methods?
- ❑ 8. Is software reuse being considered throughout the development effort?
- ❑ 9. Has an analysis of alternatives been completed?
- ❑ 10. Is the selection of architecture and design methods based on system operational characteristics?

15.3.2 During Design

- ❑ 11. Are CASE tools being used to assist and document the design effort?
- ❑ 12. Does your design process include a robust configuration control process?
- ❑ 13. Is the design effort being properly documented? Adequate but not burdensome?
- ❑ 14. Is your team committed to following the design process?
- ❑ 15. Are all design elements traceable to specific requirements?
- ❑ 16. Are all requirements traceable to design elements?
- ❑ 17. Have all software units been identified?
- ❑ 18. Are the characteristics of all data elements identified (type, format, size, units, etc.)?

15.4 References

[1] Weik, Martin H., The ENIAC Story, 1961: http://ftp.arl.mil/~mike/comphist/eniac-story.html

[2] Department of Energy (DOE) *Software Engineering Methodology*, Chapter 5: http://cio.doe.gov/sqse/sem_toc.htm

[3] ibid, Chapter 6.

[4] ibid, Chapter 7.

[5] Beck, Kent, Extreme Programming Explained, Addison-Wesley, 2000.

[6] Mayford Technologies web site: www.mayford.ca

15.5 Resources

CSIRO-Macquarie University, Design tool resources: www.jrcase.mq.edu.au/seweb/designtool/dt.html

Champeaux, Dennis de, et al, *Object-Oriented System Development*, 1993, readable online at: http://gee.cs.oswego.edu/dl/oosdw3/

Department of Energy (DOE) *Software Engineering Methodology*: http://cio.doe.gov/sqse/sem_toc.htm

Guide to Software Engineering Body of Knowledge: www.swebok.org

Little Book of Software Design, Software Program Managers Network, November 1998. Download at: www.spmn.com/products_guidebooks.html

Object Agency , "Comparison of Object-Oriented Development Methodologies": http://www.toa.com/smnn?mcr.html

Program Manager's Guide for Managing Software, 0.6, 29 June 2001, Chapters 8 & 10: www.geia.org/sstc/G47/SWMgmtGuide%20Rev%200.4.doc

Software Engineering Body of Knowledge: www.sei.cmu.edu/publications/documents/99.reports/99tr004/99tr004abstract.html