Software Design and Development

4

Abstract

This chapter discusses design and development, and software design is the blueprint of the solution to be developed. It is concerned with the high-level architecture of the system, as well as the detailed design that describes the algorithms and functionality of the individual programs. The detailed design is then implemented in a programming language such as C++ or Java. We discuss software development topics such as software reuse, customized-off-the-shelf software (COTS) and open-source software development.

Keywords

Architectural design · Detailed design · Function-oriented design · Object-oriented design · Object-oriented development · User interface design · Open-source development · Customized off-the-shelf software (COTS) · Software reuse · Software maintenance and evolution

4.1 Introduction

The user requirements specify what the customer wants and define *what* the software system is required to do, as distinct from *how* this is to be done. The user requirements are determined from discussions with the stakeholders to determine their actual needs, and they are then refined into the system requirements, which state the functional and non-functional requirements of the system.

The software design of the system is a blueprint of the solution of the system to be developed. It is concerned with the high-level architecture of the system, as well as the detailed design that describes the algorithms and functionality of the

G. O'Regan, *Concise Guide to Software Engineering*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-57750-0_4

individual programs. The detailed design is then implemented in a programming language such as C++ or Java.

Software design is a creative process that is concerned with how the system will be organized and implemented. It consists of the high-level system architecture and the low-level detailed design. The system architecture may include hardware such as personal computers and servers, as well as the definition of the subsystems with the various software modules and their interfaces. The choice of the architecture of the system is a key design decision, as it affects the performance and maintainability of the system.

The architecture is often modelled with block diagrams that give a high-level picture of the system structure, where each diagram represents a subsystem (or component) with arrows indicating the flow of data or control. The architecture facilitates discussion of the system design, as well as recording the design decisions. Architecture in the small is concerned with the architecture of individual programs, whereas architecture in the large is concerned with the architecture of large complex systems that may include other systems.

The system architecture is analogous to the architecture of a building, and it describes how the system is organized as a set of communicating structures (or subsystems). It presents the high-level design of the system, and there may be several views of the architecture (e.g. Kruchten's 4+1 model), which describe the system from different viewpoints (e.g. end-users and managers). The views (e.g. logical, development, process and physical) may be presented using various UML diagrams (e.g. class, activity and state diagrams).

The choice of the architectural design will determine the extent to which key non-functional requirements such as performance, reliability and availability are satisfied. Further, the architecture of the system is costly and difficult to modify, and so it is essential that the right architecture be chosen first time (issues such as scalability may also need to be considered). Detailed (Low-level) design is concerned with the specification of the design of the modules or individual programs.

The software development is concerned with the actual implementation of the design, and it is implemented in some programming language such as C++ or Java. The software may be developed internally or it may be outsourced to another company; existing open-source software may be employed or modified accordingly or a solution may be purchased off-the-shelf. It is essential that the design is valid with respect to the requirements and that the implemented system is valid with respect to the design.

4.2 Architecture Design

The design of the system consists of engineering activities to describe the *architecture model or structure of the system* that will satisfy the functional and non-functional requirements, as well as the *design of the individual programs* to

describe the algorithms and functionality required to implement the system requirements.

The design is concerned with how the system will be organized, and the architecture design is often presented as a set of interacting components. The design activities include architecture design, interface design, component design, algorithm design, and data structure design. There are often several possible design solutions for a particular system, and the designer will need to choose the most appropriate design of the system.

The architectural model of the system is an abstract visual representation of the structure of the system, and it is often presented as a set of boxes or block diagrams. It shows the major components of the system (i.e., the subsystems) and their interactions, and each box represents a component with the architecture showing all of the components and their connections. A box within a box represents a sub-component, and arrows are used to represent the flow of data between the components. This abstract description of the system provides a high-level view of the system and is an effective way to facilitate discussion about the system design with the relevant stakeholders.

There is a need to present multiple views of the system architecture such as how the system is decomposed into modules, how the run-time processes interact and how the hardware is distributed across the processors in the system. These views may include Krutchen's 4+1 model (Table 4.1) [1].

The process view may be described by data flow diagrams (part of the SSADM method), which show the flow of data through a system. UML is a popular design method that gives several views of the architecture of the system.

The interface design defines the interfaces between the system components, and this allows a component to be used without knowing how it is implemented. Once the interface designs have been specified, the components may be designed and developed concurrently. The component design defines how each component will operate, and the database design defines the data structures that are required. It is essential to validate the design with respect to the system requirements and to ensure that the architecture will satisfy the functional and non-functional requirements.

Table 4.1 Views of system architecture

View	Description
Logical	This view shows the key abstractions in the system as objects or object classes
Process view	This view shows how the system is composed of interacting processes at run-time
Development view	This view shows how the software is decomposed into modules/components for development
Physical view	This view shows the system hardware and how the software components are distributed across the processors in the system

Fig. 4.1 C.A.R Hoare (public domain)

Architectural design patterns are popular and date back to the mid-1990s. A design pattern is an abstract description of best practice that has worked successfully in different systems and environments, and it acts as a reusable solution that may be used in many situations. It is more a description or template on how to solve the problem within a particular context, rather than a finished solution. There are many examples of design patterns (e.g. the client server pattern includes servers and clients with services delivered from the servers).

The views of C.A.R. Hoare (Fig. 4.1) on software design are interesting. He states that there are two ways of constructing a software design.

> One way is to make it so simple that there are obviously no deficiencies.
>
> The other way is to make it so complex that there are no obvious deficiencies.

He argues that the first method is far more difficult to achieve and that it requires skill and insight. The starting point in design is always the problem domain, and it is essential that the problem to be solved be understood from a number of different viewpoints. A number of potential solutions may then be identified, and each potential solution is evaluated. This leads to the chosen solution that may, for example, be the simplest and least costly.

Design is an iterative process and the goal is to describe the system architecture that will satisfy the functional and non-functional requirements. It involves describing the system at a number of different levels of abstraction, with the designer starting off with an informal picture of the design that is then refined by adding more information.

Parnas's ideas on architecture and design have been quite influential, and he recognized that the structure of a software system matters, and getting the structure right is important. His 1972 paper "*On the criteria to be used in decomposing systems into modules*" [2] is a classic in software engineering. He introduced the

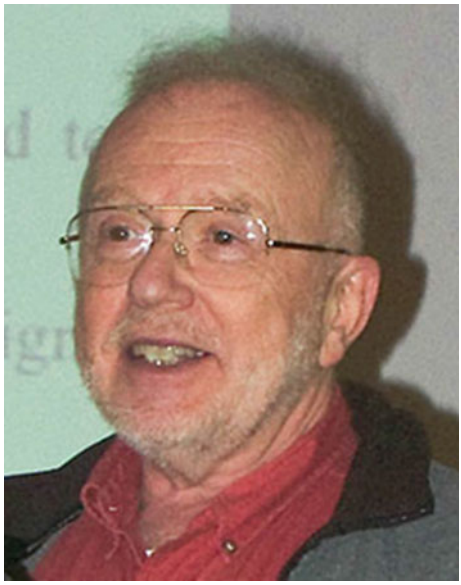

Fig. 4.2 David Parnas

revolutionary *information hiding* principle, which allows software to be *designed in a way to deal with change* (Fig. 4.2).

A module is characterized by its knowledge of a design decision (s*ecret*) that it hides from all other modules. Every information-hiding module has an *interface* that provides the only means to access the services provided by the modules. *The interface hides the module's implementation*. Information hiding is a fundamental principle that is used in object-oriented programming, and Parnas argues in his 1972 paper that:

> It is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others

The design may be specified in various ways such as graphical notations that display the relationships between the various components making up the design. The notation may include block diagrams, flow charts or various UML diagrams such as sequence diagrams and state charts.

The design of programs may employ pseudocode to specify the algorithms, as well as the data structures that are the basis for implementation. Natural language is often used to express information that cannot be expressed formally, but it is essential that the natural language description is precise and unambiguous. The design activities include:

- Architecture Design of system (with all subsystems)
- Abstract specification of each subsystem

- Interface Design (for each subsystem)
- Component Design
- Data Structure Design
- Algorithm Design.

The quality of the software architecture directly impacts the robustness, performance and maintainability of the system. The software architecture needs to manage the inherent complexity of the system, and it must ensure a solid performance of the implemented system, with safety, security, availability and maintainability requirements properly addressed.

4.3 Detailed Design and Development

The design of the system consists of engineering activities to describe the components of the system, as well as the algorithms and functions required to implement the system requirements. Design and development are concerned with developing an executable software system.

Function-oriented design involves starting with a high-level view of the system and refining it into a more detailed design. The system state is centralized and shared between the functions operating on that state. Functional design has been overtaken by object-oriented design, and so it is mainly of historic interest today.

Object-oriented design (OOD) is popular, and it is based on the concept of information hiding developed by Parnas. The system is viewed as a collection of *objects* rather than functions, with each object managing its own state information. The system state is decentralized and an object is a member of an object class. The definition of a *class* includes *attributes* and *operations* on class members, and these may be inherited from superclasses. Objects communicate by exchanging *messages*, and messages are the only way to access an object. The internal details of the object are kept private.

Software design and development are closely linked, and often proceed in parallel. Software design is the creative process that identifies the software components and their relationships, whereas software development is concerned with the implementation of the design in some programming language. The choice of language reflects the problem domain, and it may be an object-oriented language such as C++ or Java, or a procedural language such as C or FORTRAN. It is important that the software code is subject to a peer review to ensure that it is of high quality and that it is a valid implementation of the requirements and design. The coding standards for the language need to be followed, as this helps with the maintainability of the code.

Software reuse has become important and organizations recognize the importance of reuse during software development. Its advantages are that it improves software productivity and potentially provides higher quality software. Customized off-the-shelf software (COTS) provides specific functionality that may be purchased

and tailored for use in the software development. It may be possible to buy the entire system off-the-shelf, and so one of the earliest design decisions is whether *to buy* or *build* the application.

Open-source software development has become popular, and the idea is that the source code is not proprietary, but is freely available (under an open-source licence) for software developers to use and modify as they wish. It offers a way to speed up software development, as well as potentially providing a high-quality cost-effective solution.

4.3.1 Function-Oriented Design

Function-oriented design is one of the older design methodologies, and it involves starting with a high-level view of the system and refining it into a more detailed design. The system is considered to be a set of modules with clearly defined behaviour, which interact with each other in a defined manner to produce some system behaviour.

Function-oriented design views the software design as a set of functions that share state, and the functions transform the inputs to the desired outputs. The system state is centralized and shared between the functions operating on the state, and at the end of the phase, all of the major modules (as well as their interactions) and all of the main data structures of the system have been defined.

The system design (top level design) first determines which modules are needed for the system, and the detailed design expands on the system design and is focused on the internal design and specification of the modules. The detailed design is concerned with how the modules are interconnected and implemented.

The functional design is a refinement of the architectural design in that the architectural design has identified the key components, and the functional design then in a sense then determines the module structure for each component (the modules created need to be consistent with the architecture). Functional design is mainly of historic interest, as it has been overtaken by OOD.

4.3.2 Object-Oriented Design

OOD is a design method that models the system as a set of cooperating objects (rather than as a set of functions), and where the individual objects are viewed as instances of a class. OOD is concerned with the object-oriented decomposition of the system, and it involves defining the required objects and their interactions to solve the particular problem. The system state is decentralized with each object managing its own state information. The objects have a collection of attributes that define their state and operations that act on the state. The data in the object is hidden, and the only access to the data is with the operations.

The difference between a class and an object may be seen from the example that walls and windows are classes, whereas individual doors and windows are objects.

A class is a set of objects (rather than an individual object), and all members of the class share the same attributes, operations and relationships. A class may represent a software thing or a hardware thing.

A class may inherit its behaviour from one or more superclasses, with the class definition setting out the differences between the class and its superclasses. The communication between objects is done by exchanging messages (in practice, an object calls a procedure associated with another object).

An object is a "black box" that sends and receives *messages*. A black box consists of *code* (computer instructions) and *data* (information which these instructions operate on). The traditional way of programming kept code and data separate. For example, functions and data structures in the C programming language are not connected. However, in the object-oriented world, code and data are merged into a single indivisible thing called an *object*.

The reason that an object is called a black box is that the user of an object never needs to look inside the box, since all communication to it is done via messages. Messages define the *interface* to the object. Everything an object can do is represented by its message interface. Therefore, there is no need to know anything about what is in the black box (or object) in order to use it. The access to an object is only through its messages, while keeping the internal details private. This is called *information hiding* and is due to work by Parnas in the early 1970s.

The main features of the object-oriented paradigm are described in Table 4.2.

There is a need to understand the relationship between the software to be designed and its external environment. This may involve using UML to develop models such as a system context model that shows the other systems in its environment, and an interaction model that shows the interaction between the system and its environment.

This leads to the architectural design where the major components of the system and their interactions are identified. The UML diagrams help in identifying the objects and operations in the system, and the various UML models (e.g. sequence diagrams and state diagrams) show the relationships between the objects. Design patterns (best practice of solutions to common problems that may be reused) are often employed in OOD. The various UML diagrams are described in more detail in Chap. 14.

4.3.3 User Interface Design

User interface design is concerned with the design of the user interface for machines and software. The user interface is the boundary between the user and the system, and the usability of the system (as well as the user experience) will be determined by the quality of the user interface design. The user interface needs to take into account the knowledge and experience of the user, and the user interactions with the system should be as simple and efficient as possible.

Table 4.2 Object-oriented paradigm

Feature	Description
Class	A class defines the abstract characteristics of a thing, including its attributes (or properties) and its behaviours (or methods). The members of a class are termed objects
Object	An object is a particular instance of a class with its own set of attributes. The set of values of the attributes of a particular object is called its state
Method	The methods associated with a class represent the behaviours of the objects in the class
Message passing	Message passing is the process by which an object sends data to another object or asks the other object to invoke a method
Inheritance	A class may have subclasses (or children classes) that are more specialized versions of the class. A subclass inherits the attributes and methods of the parent class. This allows the programmer to create new classes from existing classes. The derived classes inherit the methods and data structures of the parent class
Encapsulation (information hiding)	One fundamental principle of the object-oriented world is encapsulation (or information hiding). The internals of an object are kept private to the object and may not be accessed from outside the object. That is, encapsulation hides the details of how a particular class works, and it requires a clearly specified interface around the services provided
Abstraction	Abstraction simplifies complexity by modelling classes and removing all unnecessary detail. All essential detail is represented, and non-essential information is ignored
Polymorphism	Polymorphism is a behaviour that varies depending on the class in which the behaviour is invoked. Two or more classes may react differently to the same message. The same name is given to methods in different subclasses: i.e. one interface and multiple methods

User interface design requires a good understanding of user needs, as well as how the user will interact with the system. It may involve prototyping of the interface and usability testing of the prototypes to judge its fitness for use. There are usability standards (e.g. ISO 9241 and ISO 16982) that provide guidance on usability.

Today's graphical user interfaces (GUI) have become ubiquitous for applications on personal computers, and a GUI is characterized by:

- Multiple windows on the screen
- Use of icon to represent information
- Command selection via menus
- Use of a mouse.

The advantages of GUIs are that they are easy to learn and use, with users with limited computing experience able to learn the user interface quite quickly.

4.3.4 Open-Source Development

Open-source development is a modern approach to software development in which the source code is published, and thousands of volunteer software developers from around the world participate in developing and improving the software. The idea is that the source code is not proprietary, and that it is freely available for software developers to use and modify as they wish. One useful benefit is that it may potentially speed up development time thereby shortening time to market.

The roots of open-source development are in the Free Software Foundation (FSF). This is a non-profit organization founded by Richard Stallman [3] to promote the free software movement, and it has developed a legal framework for the free software movement.

The Linux operating system is a well-known open-source product, and other products include mySQL, Firefox and Apache HTTP server. The quality of software produced by the open-source movement is good, and defects are generally identified and fixed faster than with proprietary software development.

A company needs to decide whether the product to be developed should use an open-source approach, as well as determining the risks and benefits associated with this approach. The type of open-source licence required needs to be identified and obtained.

4.3.5 Customized Off-the-Shelf Software

Customized off-the-shelf software (COTS) is a software (or a system) that is ready made and may be purchased off-the-shelf and adapted to the user's requirements. A COTS product typically needs to be configured for the specific use required, and the tailoring is within the parameters of the commercial software, and so custom development is usually not required.

The use of COTS components may shorten the time to market and help to reduce software development costs, as the components may be purchased from a third-party vendor rather than developed internally. Further, there is greater confidence in the quality and reliability of the COTS software (compared to custom built software), as its reliability has already been shown through its use with other organizations.

The disadvantages of COTS are that it could lead to dependency on a particular vendor, or the risk that the COTS product could become obsolete with the vendor no longer supporting it. Further, there may also be security risks if the COTS software contains security vulnerabilities (this is even more serious if the COTS software is integrated with other software products to create larger systems). For

this reason, the product development strategy needs to be clearly thought through, with all risks carefully considered.

4.3.6 Software Reuse

Software reuse is the systematic reuse of existing software technology to build software. It involves the reuse of software deliverables produced during the software development lifecycle (e.g. designs, code and test suites), and its successful implementation may shorten the time to market, as well as reducing software costs and improving software quality and productivity.

The successful introduction of reuse in an organization requires an infrastructure to support reuse. It is a lot more than creating a repository of software assets, where software engineers add software items to the depository, with the hope that other software engineers will use the contents of the repository.[1]

The reuse process involves activities to manage the reuse infrastructure, and establishing the reuse goals and the roles involved. It includes activities to create reusable assets which involve understanding the domain in which the software will be used, and designing the software for use in multiple products, as well as identifying, collecting and representing the required software assets.

Finally, it involves activities to classify and retrieve the assets in the reuse library, and activities to search and retrieve the required software assets from the library.

4.3.7 Object-Oriented Programming

Object-oriented programming has become popular in large-scale software development, and it became the dominant paradigm in programming from the early 1990s. Its proponents argue that it is easier to learn, and simpler to develop and maintain such programs, and its growth in popularity was helped by the rise in popularity of GUI, which are well suited to object-oriented programming. The C++ programming language has become popular, and it is an object-oriented extension of the C programming language.

The traditional view of programming is that a program is a collection of functions, or a list of instructions to be performed on the computer. *Object-oriented programming* is a paradigm shift in programming, where a computer program is considered to be a collection of objects that act on each other. Each object is capable of sending and receiving messages and processing data. That is, each object may be viewed as an independent entity or actor with a distinct role or responsibility.

[1] I recall Parnas making a joke many years ago that we have developed all of this reusable software that nobody reuses.

The origins of object-oriented programming go back to the invention of Simula 67 at the Norwegian Computing Research Centre[2] in the late 1960s. It introduced the notion of a class and instances of a class.[3] Simula 67 influenced later languages such as the Smalltalk object-oriented language developed at Xerox PARC in the mid-1970s.

Xerox introduced the term "*Object-oriented programming*" for the use of objects and messages as the basis for computation. Most modern programming languages support object-oriented programming, and object-oriented features have been added to many existing languages such as BASIC, FORTRAN and Ada.

C++ and Java Bjarne Stroustrup developed the C++ programming language in 1983 as an object-oriented extension of the C programming language. It was designed to use the power of object-oriented programming and to maintain the speed and portability of C. It provides a significant extension of C's capabilities, but it does not force the programmer to use the object-oriented features of the language.

A key difference between C++ and C is the concept of a class. A *class* is an extension to the C concept of a structure. The main difference is that while a C data structure can hold only data, a C++ class may hold both data and functions. An *object* is an instantiation of a class: i.e. the class is essentially the type, whereas the object is essentially a variable of that type. Classes are defined in C++ by using the keyword class.

Java is an object-oriented programming language developed by James Gosling and others at Sun Microsystems in the early 1990s. C and C++ influenced the syntax of the language, and the language was designed with portability in mind. The objective is for a program to be written once and executed anywhere. Platform independence is achieved by compiling the Java code into Java bytecode, which are simplified machine instructions specific to the Java platform.

This code is then run on a Java virtual machine (JVM) that interprets and executes the Java bytecode. The JVM is specific to the native code on the host hardware. The problem with interpreting bytecode is that it is slow compared to traditional compilation. However, Java has a number of techniques to address this including just in time compilation and dynamic recompilation. Java also provides automatic garbage collection. This is a very useful feature as it protects programmers who forget to deallocate memory (thereby causing memory leaks).

The reader is referred to [4] for a more detailed explanation of the design and development activities.

[2]The inventors of Simula-67 were Ole-Johan Dahl and Kristen Nygaard.

[3]Dahl and Nygaard were working on ship simulations and were attempting to address the huge number of combinations of different attributes from different types of ships. Their insight was to group the different types of ships into different classes of objects, with each class of objects being responsible for defining its own data and behaviour.

4.4 Software Maintenance and Evolution

Software maintenance is the process of changing a system after it has been delivered to the customer, and it involves correcting any defects that are present in the software and enhancing the system to meet the evolving needs of the customer. The defects may be due to coding, design or requirements errors, with coding defects the cheapest to fix and requirements defects the most expensive to correct. The resolution to the defects involves identifying the affected software components and modifying them, and verifying that the solution is correct and that no new problems have been introduced.

Software systems often have a long lifetime (e.g. some systems have a lifetime of 20–30 years), and so the software needs to be continuously enhanced over its lifetime to meet the evolving needs of the customer. Software evolution is concerned with the continued development and maintenance of the software after its initial release, with new releases of the software prepared each year. Each new release includes new functionality and corrections to the known defects.

4.5 Review Questions

1. What is the difference between requirements and design?
2. Explain the difference between architectural design and detailed design.
3. Explain the difference between functional-oriented design and OOD.
4. What are the advantages and disadvantages of COTS software.
5. What is object-oriented programming?
6. What is software reuse and how is it accomplished?
7. Explain the differences between COTS, software reuse and open-source software.
8. Explain the difference between software maintenance and evolution.

4.6 Summary

The success of business is highly influenced by software, and companies may develop their own software internally, or they may acquire software solutions off-the-shelf or from bespoke software development.

The user requirements specify what the customer wants and define *what* the software system is required to do, as distinct from *how* this is to be done. The requirements are the foundation for the system, and it is essential that they are correct and reflect the needs of the customer.

The software design of the system is a blueprint of the system to be developed. It is concerned with the high-level architecture of the system, as well as the detailed design that describes the algorithms and functionality of the individual programs. Software design is a creative process that is concerned with how the system will be organized and implemented.

The system architecture may include hardware such as computers and servers, as well as the definition of the subsystems with the various software modules and their interfaces. The choice of the architecture of the system is a key design decision, as it affects the performance and maintainability of the system.

The detailed software design of the system is concerned with activities to describe the algorithms and functions required to implement the system requirements. It may include hardware as well as the various software modules and their interfaces. Design and development are concerned with developing an executable software system.

The software development is concerned with the actual implementation of the design in some programming language such as C++ or Java. The software may be developed internally or it may be outsourced to another company or a solution may be purchased off-the-shelf. It is essential that the design is valid with respect to the requirements and that the implemented system is valid with respect to the design.

References

1. P. Kruchten, Architectural blueprints—the "4+1" view model of software architecture. IEEE Softw. **12**(6), 42–50 (1995)
2. D. Parnas, On the criteria to be used in decomposing systems into modules. Commun. ACM **15** (12) (1972)
3. G. O'Regan, *Giants of Computing* (Springer, London, 2013)
4. I. Sommerville, *Software Engineering*, 9th edn. (Pearson, Boston, 2011)