E-notes Software Engineering

By Sagar N. Gharate

Syllabus

References:

1. Software Engineering : A Practitioner's Approach - Roger S. Pressman, McGraw hill(Eighth Edition) ISBN-13: 978-0-07-802212-8, ISBN-10: 0-07-802212-6
2. A Concise Introduction to Software Engineering - Pankaj Jalote, Springer ISBN: 978-1-84800-301-9
3. The Unified Modeling Language Reference Manual - James Rambaugh, Ivar Jacobson, Grady Booch ISBN 0-201-30998-X

Chapter 5

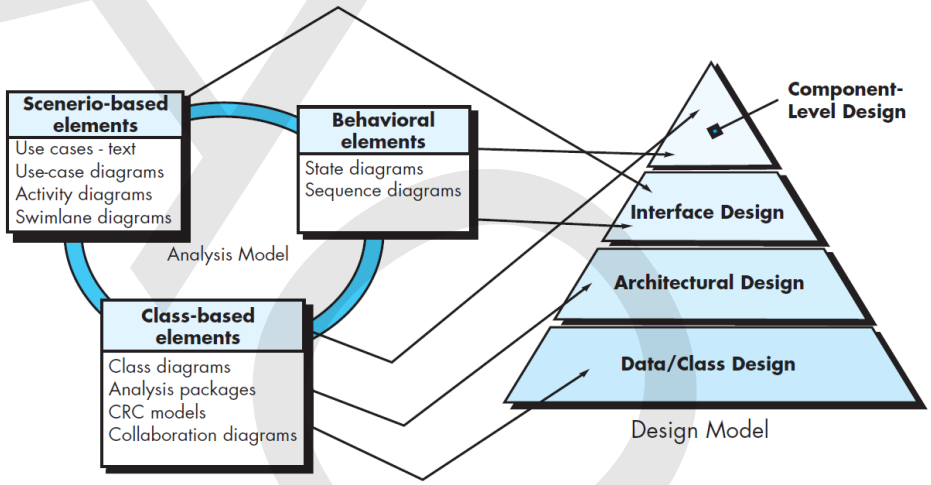
Design Concepts

DESIGN CONCEPTS

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles establish an overriding philosophy that guides the design work you must perform. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design Within The Context Of Software Engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analysed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).



Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in above Figure.

- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design activity. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.
- The architectural design defines the relationship between major structural elements of the software, the architectural styles and patterns.
- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

- The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models and behavioral models serve as the basis for component design.

During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word "Quality". Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction- a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Software Quality Guidelines and Attributes

Throughout the design process, the quality of the evolving design suggested three characteristics that serve as a guide for the evaluation of a good design:

- The design should implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- The design should be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

Quality Guidelines

In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. Consider the following guidelines

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics, and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS- functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

- Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- Usability is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- Performance is measured using processing speed, response time, resource consumption, throughput, and efficiency.
- Supportability combines extensibility, adaptability, and serviceability. These three attributes represent a more common term, maintainability and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration), the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security. Another may demand performance with particular emphasis on processing speed. A third might focus on reliability. Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, not after the design is complete and construction has begun.

The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned more than six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down "structured" manner. Newer design approaches proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions. Growing emphasis on aspect-oriented methods, model-driven development and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

A number of design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow.

DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in these concepts has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you define criteria that can be used to partition software into individual components, separate or data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design.

Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem oriented terminology is coupled with implementation oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

As different levels of abstraction are developed, you work to create both procedural and data abstractions. A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A data abstraction is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

Architecture

Software architecture imply to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

A set of properties that should be specified as part of an architectural design are as follow; *Structural properties* define "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another." *Extra-functional properties* address "how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics. *Families of related systems* "draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

- Structural models represent architecture as an organized collection of program components.
- Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
- Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
- Process models focus on the design of the business or technical process that the system must accommodate.

- Functional models can be used to represent the functional hierarchy of a system.

A number of different architectural description languages (ADLs) have been developed to represent these models. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another

Patterns

Brad Appleton defines a design pattern in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns". Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine

1. Whether the pattern is applicable to the current work,
2. Whether the pattern can be reused (hence, saving design time), and
3. Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller and therefore more manageable pieces, a problem takes less effort and time to solve.

It follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy- it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

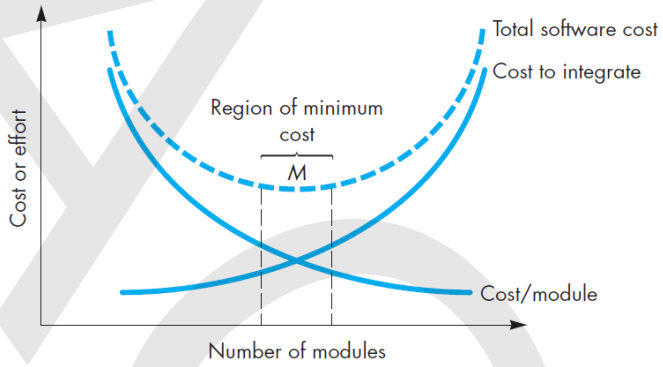
Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Recalling our discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to below Figure, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.



The curves shown in Figure do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. Under modularity or over modularity should be avoided.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Information Hiding

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with "single-minded" function and a "disinclination" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling. Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates throughout a system.

Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. An application is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of *elaboration*. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

Aspects

As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts". Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, A and B. Requirement A crosscuts requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account".

It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components. To accomplish this, the design architecture should support a mechanism for defining an aspect- a module that enables the concern to be implemented across all other concerns that it crosscuts.

Refactoring

An important design activity suggested for many agile methods, refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Mr Fowler defines refactoring in the following manner: "Refactoring is the

process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur. As a consequence, refactoring tools are used to analyse changes automatically and to "generate a test suite suitable for detecting behavioral changes."

Object-Oriented Design Concepts

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Unlike conventional software design methods, OOD results in a design that achieves a number of different levels of modularity. Major system components are organized into subsystems, a system-level "module." Data and the operations that manipulate the data are encapsulated into objects- a modular form that is the building block of an OO system. In addition, OOD must describe the specific data organization of attributes and the procedural detail of each individual operation. These represent data and algorithmic pieces of an OO system and are contributors to overall modularity.

The unique nature of object-oriented design lies in its ability to build upon four important software design concepts: abstraction, information hiding, functional independence, and modularity. All design methods strive for software that exhibits these fundamental characteristics, but only OOD provides a mechanism that enables the designer to achieve all four without complexity or compromise. Object-oriented design, object-oriented programming, and object-oriented testing are construction activities for OO systems.

Design Classes

The analysis model defines a set of analysis classes. Each of these classes describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, you will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed. User interface classes define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor. Business domain classes identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes. Process classes implement lower-level business abstractions required to fully manage the business domain classes. Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software. System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class:

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class Scene defined for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class VideoClip for videoediting software might have attributes and to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, setStartPoint() and setEndPoint(), provide the only means for establishing start and end points for the clip.

High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class VideoClip might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to

implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the Law of Demeter [Lie03], suggests that a method should only send messages to methods in neighboring classes.

Dependency Inversion

The structure of many older software architectures is hierarchical. At the top of the architecture, "control" components rely on lower-level "worker" components to perform various cohesive tasks. Consider a simple program with three components. The intent of the program is to read keyboard strokes and then print the result to a printer. A control module, C, coordinates two other modules- a keystroke reader module, R, and a module that writes to a printer, W.

The design of the program is coupled because C is highly dependent on R and W. To remove the level of dependence that exists, the "worker" modules R and W should be invoked from the control module S using abstractions. In object-oriented software engineering, abstractions are implemented as abstract classes, R* and W*. These abstract classes could then be used to invoke worker classes that perform any read and write function. Therefore a copy class, C, invokes abstract classes, R* and W*, and the abstract class points to the appropriate worker-class (e.g., the R* class might point to a read() operation within a keyboard class in one context and a read() operation within a sensor class in another. This approach reduces coupling and improves the testability of a design.

The example discussed in the preceding paragraph can be generalized with the dependency inversion principle, which states: *High-level modules (classes) should not depend [directly] upon low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

Design for Test

There is an ongoing chicken-and-egg debate about whether software design or test case design should come first. Rebecca Wirfs-Brock writes:

Advocates of test-driven development (TDD) write tests before implementing any other code. They take to heart Tom Peters' credo, "Test fast, fail fast, adjust fast." Testing guides their design as they implement in short, rapid-fi re "write test code-fail the test-write enough code to pass-then pass the test" cycles.

But if design comes first, then the design (and code) must be developed with seams-locations in the detailed design where you can "insert test code that probes the state of your running software" and/or "isolate code under test from its production environment so that you can exercise it in a controlled testing context".

Sometimes referred to as "test hooks," seams must be consciously designed at the component level. To accomplish this, a designer must give thought to the tests that will be conducted to exercise the component. As Wirfs-Brock states: "In short, you need to provide

appropriate test affordances- factoring your design in a way that lets test code interrogate and control the running system."

The Design Model

Data Design Elements

Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program-component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high- quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role.

Architectural Design Elements

The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural model is derived from three sources:

1. Information about the application domain for the software to be built;
2. Specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and
3. The availability of architectural styles and patterns.

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture (e.g., a graphical user interface might be structured according to a pre-existing architectural style for user interfaces).

Interface Design Elements

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design:

1. The user interface (UI),
2. External interfaces to other systems, devices, networks, or other producers or consumers of information, and
3. Internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

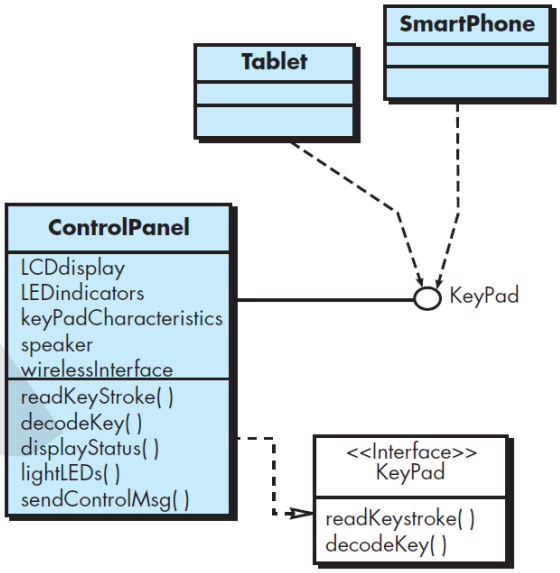
UI design (increasingly called usability design) is a major software engineering action. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering and verified once the interface design commences. The design of external interfaces should incorporate error checking and appropriate security features.

The design of internal interfaces is closely aligned with component-level design. Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes. Each message must be designed to accommodate the requisite information transfer and the specific functional requirements of the operation that has been requested.

In some cases, an interface is modeled in much the same way as a class. In UML, an interface is defined in the following manner: “An interface is a specifier for the externally-visible [public] operations of a class, component, or other classifier (including subsystems) without specification of internal structure.” Stated more simply, an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

For example, the SafeHome security function makes use of a control panel that allows a homeowner to control certain aspects of the security function. In an advanced version of the system, control panel functions may be implemented via a mobile platform (e.g., smartphone or tablet).



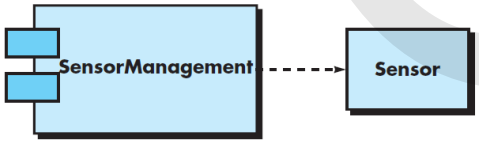
The ControlPanel class (above Figure) provides the behavior associated with a keypad, and therefore, it must implement the operations readKeyStroke () and decodeKey (). If these operations are to be provided to other classes (in this case, Tablet and SmartPhone), it is useful to define an interface as shown in the figure. The interface, named KeyPad, is shown as an <<interface>> stereotype or as a small, labelled circle connected to the class with a line. The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.

The dashed line with an open triangle at its end (aboveFigure) indicates that the ControlPanel class provides KeyPad operations as part of its behavior. In UML, this is characterized as a realization. That is, part of the behavior of ControlPanel will be implemented by realizing KeyPad operations. These operations will be provided to other classes that access the interface.

Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets, and every other detail associated with a room.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).



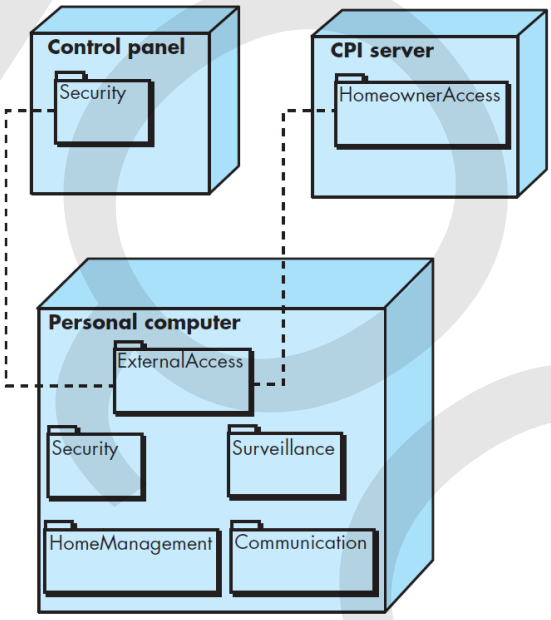
Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in above Figure. In this figure, a component named

SensorManagement (part of the SafeHome security function) is represented. A dashed arrow connects the component to a class named Sensor that is assigned to it. The SensorManagement component performs all functions associated with SafeHome sensors including monitoring and configuring them.

The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. For example, the elements of the SafeHome product are configured to operate within three primary computing environments- a homebased PC, the SafeHome control panel, and a server housed at CPI Corp. (providing Internet-based access to the system). In addition, limited functionality may be provided with mobile platforms.



During design, a UML deployment diagram is developed and then refined as shown in above Figure. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and functionality delivered by mobile platforms). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features. In addition, an external access subsystem has been designed to manage all attempts to access the SafeHome system from an external source. Each subsystem would be elaborated to indicate the components that it implements.

The diagram shown in above Figure is in descriptor form. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the “personal computer” is not further identified. It could be a Mac, a Windows-based PC, a Linux-box or a mobile platform with its associated operating system. These details are provided when the deployment diagram is revisited in instance form during the latter stages of design or as construction begins. Each instance of the deployment (a specific, named hardware configuration) is identified.

Chapter 5 Questions:

A. MCQ's:

1. A design should be_____; that is, the software should be logically partitioned into elements or subsystems.

   a. complex b. critical

   c. modular d. aligned

2. Cohesion is a natural extension of the _____ concept.

   amodularity b. information-hiding

   c. patterns d. aspects

Ans:

1. c 2. b

B. State True or False

1. Every software quality attribute is weighted equally as the software design is developed.
2. Independence is assessed using two qualitative criteria: cohesion and coupling.

Ans:

1. False 2. True

C. 1 mark questions

1. Why is design for testing so important?
2. Which are the four elements of the design model
3. Which are the software quality attributes?
4. What is mean by Pattern?
5. Define four characteristics of a design class
6. Give any two quality guidelines

D. Brief Questions

1. Describe software architecture in your own words?
2. Write short note on Interface Design Elements
3. How do we assess the quality of a software design?
4. What is Deployment-Level Design Elements?
5. Discuss what the dependency inversion principle?
6. Write short note on design for test
7. What are the goal of design process?