Software Design and Architecture (Handout)

Topic 1

Overview

In first topic of software architecture and design, we will explore the "Design" meaning in the Cambridge dictionary as verb and noun. We will discuss the application of design in different areas such as design of houses, chairs and helmet. At the end software design and its need will be described in this topic with examples.

Learning Goals

• Learn the design definition as verb and noun
• Learn about the description of design from different areas
• Learn about the software design
• Learn about the need of the design activity

Pre-requisite Courses:

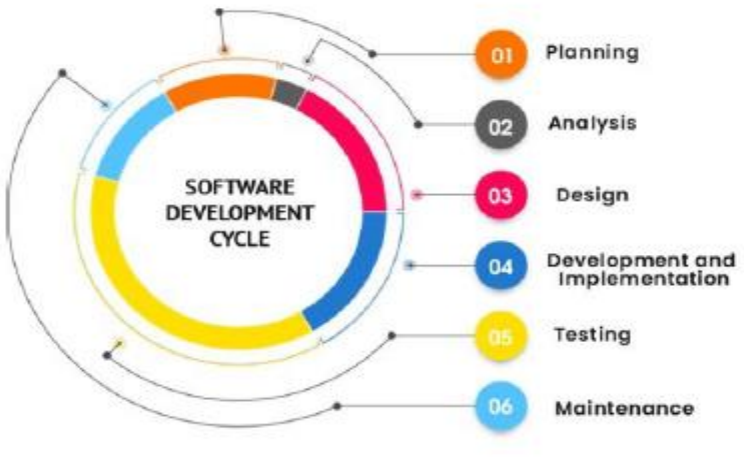
Software engineering, I
Software engineering II
Introduction to programming

Pre-existing Knowledge:

You should know the following concepts prior taking this course
• Basic programming concepts
• Software development life cycle model
• Software requirement specification document (SRS)

Software Application

Software application development process consist on different phases mentioned in the below figure. Planning, analysis, design, deployment and implementation, testing and maintenance are steps involved in software development.



Topic 2

Overview

In this topic, you will get an overview of the design objectives, what we want to achieve from the design, the necessity of the design phase, things involved in design process, what it specifies and how a design should be.

Learning Goals

- Learn about the design objectives
- Learn about the necessity of the design
- Learn about the requirements of the design
- Learn about the problem domain vs design
- Learn about the customer needs and design

Customer Requirements

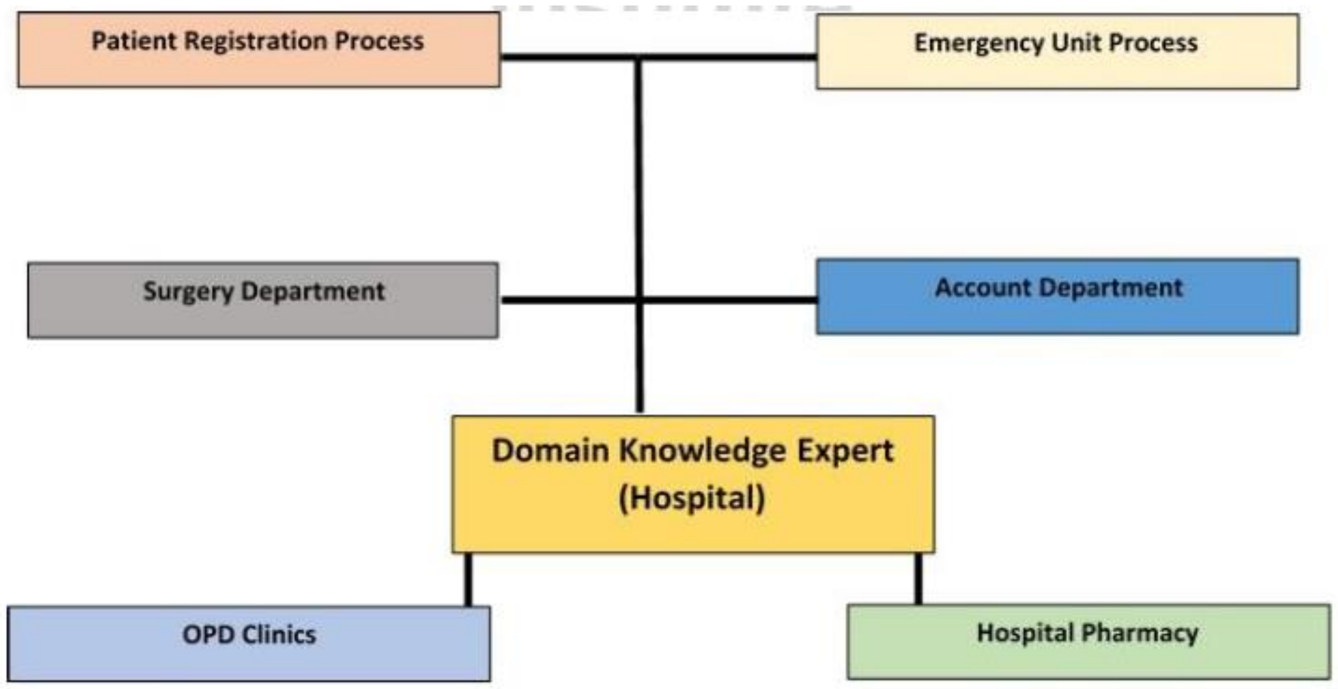
Customer requirements mean what customers or users required from the system. How the system behaves, what are the tasks need to perform by the system, how users will interact with the system, and how the system will present the output. The customer is a significant stakeholder that plays a vital role in the correct development, successful implementation, and system utilization.

Domain Knowledge

Domain knowledge means the information and having expertise about the particular environment and industry in which the developed system will operate. For example, knowledge of the pharmaceutical industry and telecom industry.

Domain Knowledge Expert Example

A person have knowledge of hospitalization process and its major departments.



Topic:3

Overview

In this topic, you will get an overview of the design complexity, poorly designed software, why poorly designed software hard to understand and modify. Adding features in poorly and well-designed software effect on cost.

Learning Goals

- Learn about the design complexity
- Learn about the issues with poorly designed software

What is poorly designed software?

Poorly designed means the designer did not spend a good amount of time to analysis software requirements to design. In start, adding small features to show working product to a client is very easy but with the passage of time, software size increased that makes it difficult to understand and make further modifications and adding more features.

What is extendable design?

Change in software requirements and adding more features is very much natural in software development and maintenance process. Well-designed software is easy to understand and welcomes changes in requirements, future modifications and adding new features.

Topic:4

Overview

In this topic, you will get to know the complexity of an object with respect to its size. Application of software engineering in software development.

Learning Goals

- Learn about the complexity vs size
- Learn about the application of software engineering
- Learn about the size does matter
- Learn about the consequence of the poor design

Complexity Vs Size (Example of a Building)

The building in the below image is tall with many floors; the building may contain many offices, commercial shops, parking areas, and apartments. Each entity (offices, shops, etc.) have different kind of facilities, fire exit directions, earthquake resistance, sanitary and sewerage. Due to its large size, any addition or modification will be hard to manage and demand excellent design and architectural skills. In contrast, a single room house is easy to manage and extend.

Complexity VS Size (Example of two different Applications)

A calculator with four functionalities, addition, subtraction, multiplication, and division, is easy to manage, modify, or add any new functionality. While a big application, enterprise resource planning (ERP), contains many different integrated components, it is hard to modify, manage, or add a new feature. Because a calculator has 100 to 200 lines of code, but an ERP solution has millions of lines of code so a single change in ERP may affect other components. You can imagine the complexity relation with the size in the above example; the bigger the size, the bigger the complexity, while the smaller the size, the smaller the complexity.

Topic: 5

Overview

In this topic, you will study the types of complexity, reason of these complexities occurrence in the design and development process.

Learning Goals

• Learn about the types of complexities
• Learn about the examples of types of complexities

Types of Complexities

• Essential Complexities
• Accidental complexities

Essential Complexities

You will understand Essential Complexities after reading below different descriptions of essential complexities.

• Essential Complexity is just the nature of the beast you are trying to tame.
• Essential Complexity represents the difficulty inherent in any problem.
• Essential Complexity is Complexity inherent to the problem. It is Complexity related to the problem and cannot remove.
• Essential Complexity is how hard something is to do, regardless of how experienced you are, what tools you use or what new and flashy architecture pattern you used to solve the problem. Essential Complexity is Complexity inherent to the problem.
• Essential Complexity is the entanglement/combination of components/ideas in software necessary for solving the problem at hand. It cannot avoid.

Essential Complexities Examples

Example 1

If users need a program to do 30 things, then those 30 things are essential; you cannot simply take out a few of them to make the software less complex. Whenever you are solving a problem, there are just some areas of complexity that cannot be whittled down.

Example 2

Coordinating a nation's air traffic is an inherently complex problem. Every plane's exact position (including altitude), speed, direction, and destination must be tracked in real-time to prevent mid-air and runway collisions. The flight schedules of aircraft must be managed to avoid airport congestion in a continuously changing environment—a severe change in weather throws the entire schedule out of whack.

Accidental Complexities

You will understand Essential Complexities after reading below different descriptions of essential complexities.

• Accidental Complexity is Complexity not related to the problem. Ben Mosely and Peter Marks describe it as a "mishap." It is Complexity from the fault of the developer and happens to be there.
• Accidental Complexity is the entanglement of components/ideas that is not necessary for solving the

problem. This Complexity is accidental because someone probably didn’t think hard enough before unnecessarily tying things together. As a result, the software is harder to understand than it should be.
• Accidental Complexity grows from the things we feel we must build to mitigate essential Complexity.

Accidental Complexities Examples

Example 1
Accidental complexity refers to challenges that developers unintentionally make for themselves due to trying to solve a problem. (Fortunately, this kind of complexity can also be fixed or improved by developers.)

Example 2
Developers bring complexity while writing the program. This type of complexity is called accidental complexity.

How to remove accidental complexities?
Accidental complexities can be removed or reduce with the better design.

Topic:6

Overview
In this topic, you will study, why software designs are hard, creativity in software designs, problems in designs and abstraction.

Learning Goals
• Learn about why design hard
• Learn about design problems
• Learn about the abstraction

Why software design is very important to manage complexity?
One of the most important techniques for managing software complexity is to design systems so that developers only need to face a small fraction of the overall complexity at any given time. Because software designer provides a road map (blueprint) to the developer in which he mentions the system's overall structure.

What is abstraction?
An abstraction is a simplified view of an entity, which omits unimportant details. The software designer ignores the minor details attached to the problem and focused on the high-level details. For example, a software designer designs the system's overall structure instead of working on the minor details of the application, such as the color of user interface forms and font styles of text.

Why software design is hard?
The fundamental problem is that designers must use current information to predict a future state that will not come about unless their predictions are correct. The outcome of designing has to be assumed before the means of achieving it can be explored: the designers have to work backward in time from an assumed effect upon the world to the beginning of a chain of events that will bring the effect about.

Topic: 7

Overview

In this topic, you will study the relation of software design with science and art. Need of creativity in software designing. Science is fundamental principle and art is creativity.

Learning Goals

• Learn about science
• Learn about art
• Learn about relation of science and art in design

What is Art?

Art is a diverse range of human activities involving creative imagination to express technical proficiency, beauty, emotional power, or conceptual ideas.

What is Science?

• Science is made up of fundamental principles, which can be taught as truth.
• A branch of knowledge or study dealing with a body of facts or truths systematically arranged and showing the operation of general laws.

Topic:8

Overview
In this topic, you will study the wicked problem, meaning of guru and his expertise. Moreover, you will study that the software design problem is a ‘wicked’ one and this imposes constraints upon the way in which the process of design can be organized and managed.

Learning Goals
- Learn about the wicked problem
- Learn about guru expertise level
- Learn about that design is a wicked one

Topic: 9

Overview
In this topic, you will understand about the design process, how you will apply it and improve it with the application of methods, techniques, patterns and heuristics.

Learning Goals
• Learn about the design process
• Learn about design process application and improvement

Practice Makes the Man Perfect

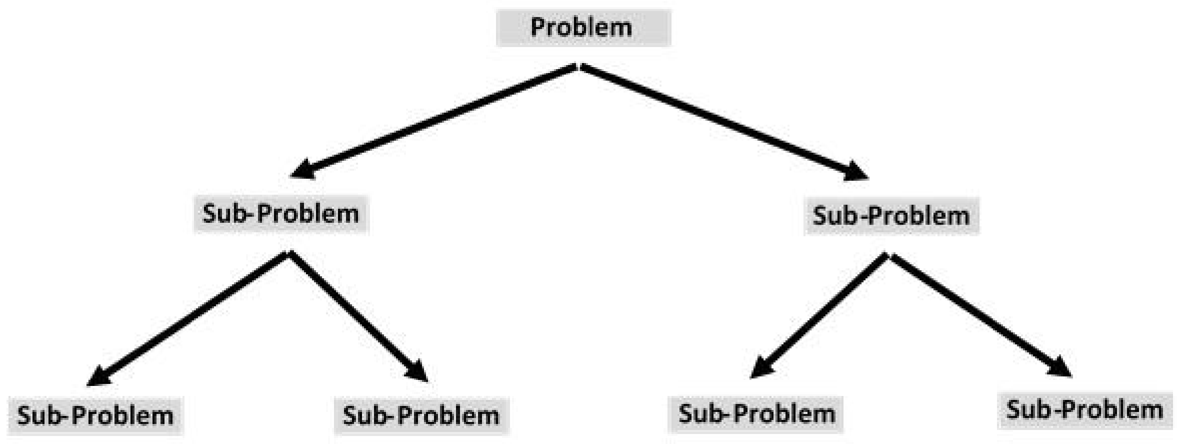
How will we understand the design process? A question arises in our minds. How can we apply a design process to make better software design? Design is a creative activity; every person has different skill levels and techniques, some have more, and some have less. However, through continuous learning processes and practice, we can improve our skills and techniques. The design process would be more systematic and predictable by applying methods, techniques, and patterns according to principles and heuristics. After continues learning and practicing, a designer will be able to get good designs. As a designer grows in this profession, he will develop his principles, guidelines, and heuristics. You may not be able to document and express all expertise.

Topic:10

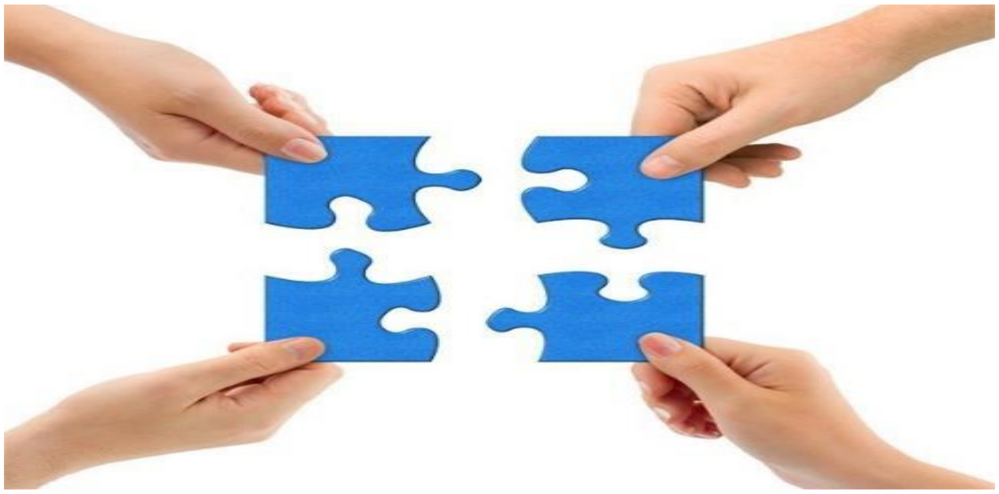
Overview
In this topic, you will understand the different ways to deal with software complexity. Divide and conquer techniques to slice a problem. You will get to know the modularity, hierarchical organization of the sliced problem, information hiding and abstraction.

Learning Goals
• Learn about the different ways to deal with complexity
• Learn about modularity
• Learn about the hierarchical organization
• Learn about the information hiding
• Learn about the abstraction

1, Divide and Conquer
This topic is related to a strategy for solving a problem. If a problem is P and it is a large problem. Suppose its size is n. Then we will divide P into smaller sub-problems P1, P2, P3……. Pk. So a big problem has been divided into smaller sub-problems. Obtaining a solution to each smaller problem is easy rather than solving a big problem at once. All sub-problems are solved individually. After getting the solution to each sub-problem, combine all solutions into one big solution of the one big problem. If the sub-problem is also big then apply the same divide and conquer strategy to divide this sub-problem further.



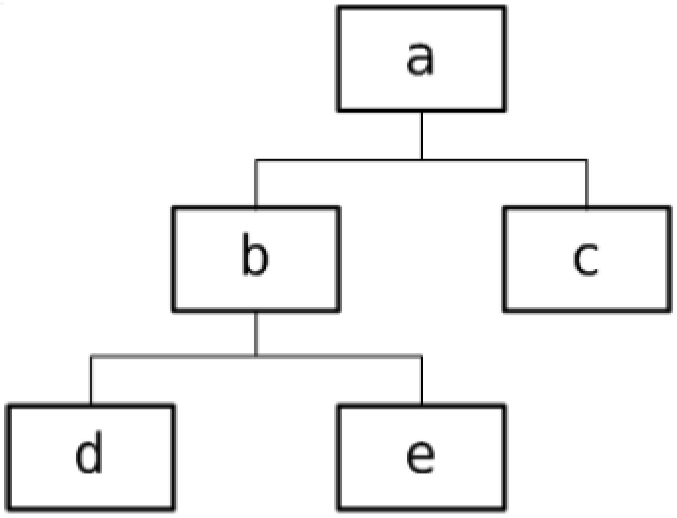
2, Modularity
The dictionary defines module as “each of a set of standardized parts or independent units that can be used to construct a more complex structure.”
For example: ‘ships are now built in modules rather than built in a whole from the base up’.



In above figure, you can imagine that four individual modules have been solved individually then combined all modules into one combined solution.

3, Hierarchical Organization / Structure

A big problem is broken into smaller meaningful parts (Sub-systems) that can be solved individually then we need to combine all these parts that depict the solution of a big problem. But how will we know which part will be integrated with which part? How we will know, which part will be top-level and other parts will come as the sub-part of the top-level part. So Hierarchical Organization provides the solution to organize each part in proper Hierarchy/structure.



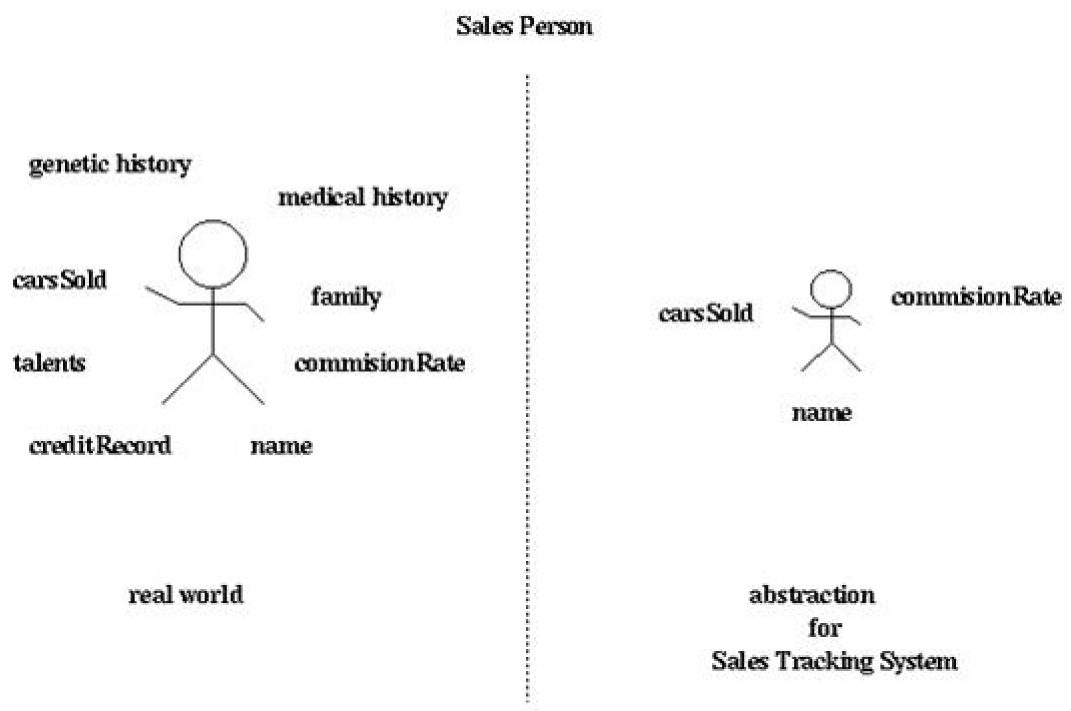
4, Information Hiding

• A component encapsulates its behaviors and data, hiding the implementation details from other components. Modules should be designed so that information (i.e., algorithms and data) contained within one module is inaccessible to other modules that have no need for such information.

• Information hiding, the purpose of information hiding is to separate interface from implementation. By hiding the implementation details that are likely to change, encapsulation allows for flexibility in design. For example, a stack interface can define two public operations, push and pop, that are available to other parts of the program. However, its internal data organization, e.g., whether to use a linked list or an array for storing stack elements, can be hidden.

5, Abstraction
Transitioning your thought process from program design to software design involves the use of abstraction. When designing software, you think of ways to abstract away certain details. This is typically done by grouping certain details together based on shared characteristics or purposes. These shared characteristics or purposes are then a way to generalize your design. For example, a programmer may think about a soccer ball, tennis ball, baseball, and softball as distinct types of items. A software designer would generalize these items by classifying all of them as a type of ball that share certain characteristics (e.g., they are all spheres with a center point and radius) and purposes (e.g., they are all struck by an object). (Reference Book: Guide to Efficient Software Design by David P. Voorhees)

6, Abstraction Hand-on-Exercise
You are required to explain and discuss below image moderate discussion board.



Topic: 11

Overview

In this topic, you will learn about the design characteristics, deterministic process, design is non-deterministic, design is heuristic base, design is evolved with the passage of time and design is an emergent process.

Learning Goals

- Learn about the design characteristics

Topic: 12

Overview

In this topic, benefits of good design is discussed. A good design benefits includes easy to understand, easy to implement and easy to maintain. Furthermore, benefits of code reuse is discussed. How quality will increase with the code reuse and how complexity is the root cause.

Learning Goals

• Learn about the benefits of good design
• Learn about the code reuse and quality
• Learn about the complexity is the root cause

1, Why code reuse is important?

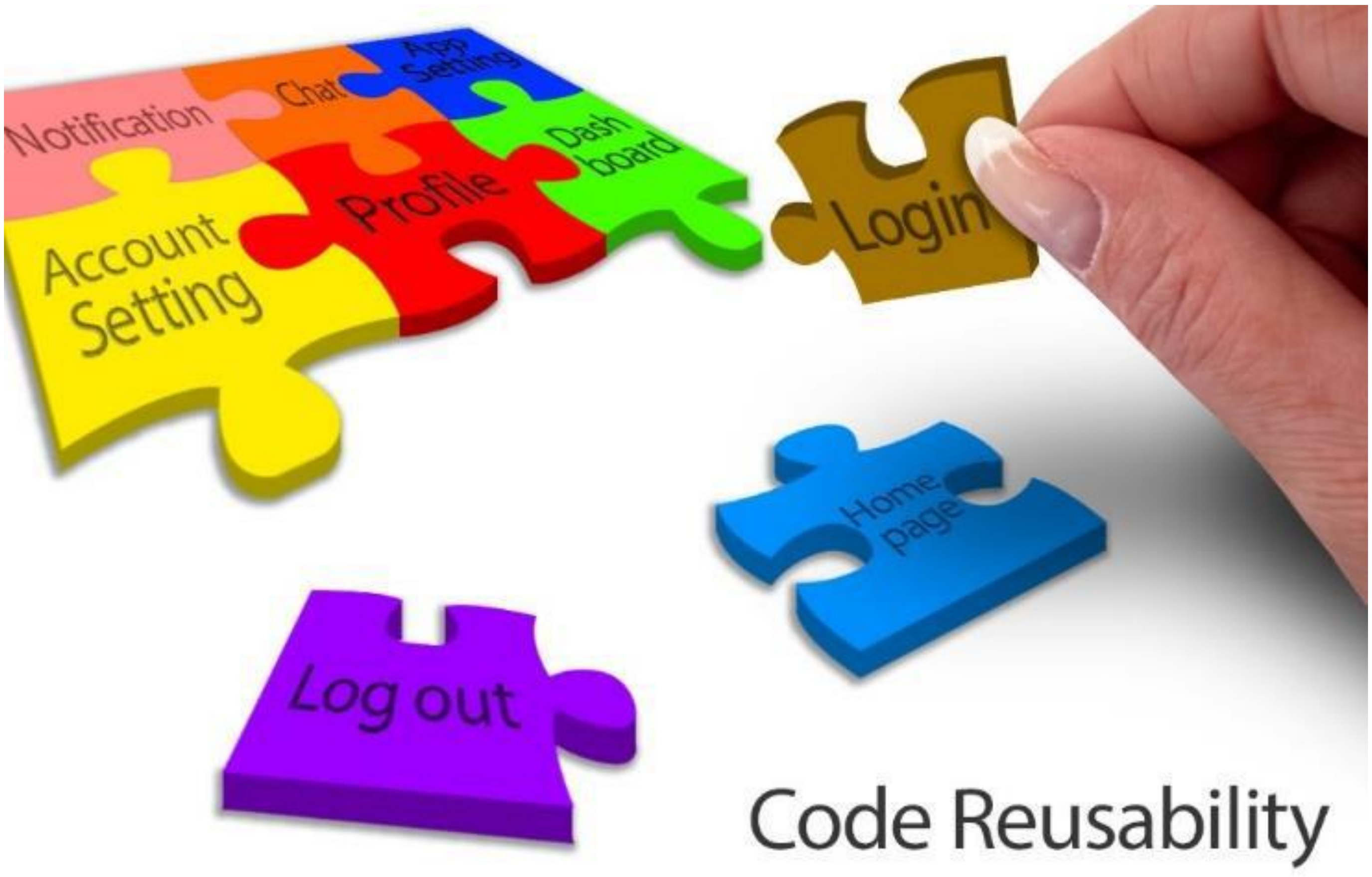
If you have developed a software, library, class, written an algorithm or written a function. If you have written it well and test properly. Moreover, this functionality is used repeatedly in your software. Its mean that you can call/copy/paste this functionality in your code. In return, your development effort will be less, time will be saved and you can utilize your time efficiently. One more benefit of code reuse is an enhancement in the quality. For example, a code written that tested multiple times and used in different locations in your code will increase the quality of the software and boost the confidence.

2, Software Quality in regard to Design

The concept of quality is a familiar one, although we tend to associate it with properties that arise from the tasks of construction rather than with those of design. Asked for examples of 'good' quality, many people are likely to identify such examples as well-made joints on an item of furniture, the quality of the paintwork on an expensive car, or the texture of a woven fabric. All these are examples of quality of construction rather than of design, yet quality in construction depends upon good design: to achieve high-quality product high standards are needed in both, and in order to achieve high standards of quality in design products, one needs to seek high standards of quality in the design process. Unfortunately, quality is a concept that can rarely be related to any absolutes, and even for manufactured items ideas about quality usually cannot be usefully measured on any absolute scale. The best that we can usually do is to rank items on an ordinal scale, as when we say that we believe the construction of this coffee table is better than that of another, in terms of materials used and the workmanship it exhibits. However, we lack any means of defining how much better the one is than the other. When it comes to assessing design, rather than construction, the problem is, if anything, worse. Quality in design is often associated with visual properties, as when we appraise the quality typefaces, furniture and clothes. We associate elegance with good design, yet it offers no help in quantifying our thinking and, even worse, is likely to have ephemeral aspects: it may be strongly influenced by our ideas about fashion – so creating a context for quality. (As an example, a 1930s radio set might look quite in place in a room that is furnished in 1930s style, but would be most unsuited to a room that was furnished in the style of the 1980s.)

No matter how good the design is, in terms of proportions, efficient use of materials and ease of construction and modification, our ideas about its quality will still be very largely influenced by the actual construction. If it is badly assembled, then, regardless of how good its design may be, we will not consider it to be of good quality. Equally, if the design is poor, no amount of good craftsmanship applied to its construction will be able to disguise its fundamental failings. (The door may be well made and may close well, but if it is positioned so that we graze our knuckles each time it is opened, then we will still find it inconvenient to use.) [Book: Software Design by David Budgen]

Exercise: Analyze the below image regarding code reusability and describe that what comes to your Mind



Topic: 13

Overview

In this topic, following areas will be discussed such as, design process, the final design evolves from experience and feedback. Design is an iterative and incremental process, where a complex system arises out of relatively simple interactions. Software design steps, for example, understanding the problem (requirements), construct the black box model of solution (system specification), look for existing solutions (e.g. architecture and design patterns) that cover some or all of the software design problems identified. Discover missing requirements, consider building prototypes, document and review design, iterate over solution (refactor) (evolve the design until it meets functional requirements and maximize non-functional requirements).

Learning Goals

- Learn about the design process
- Learn about the design is iterative and incremental process
- Learn about the steps involved in design process

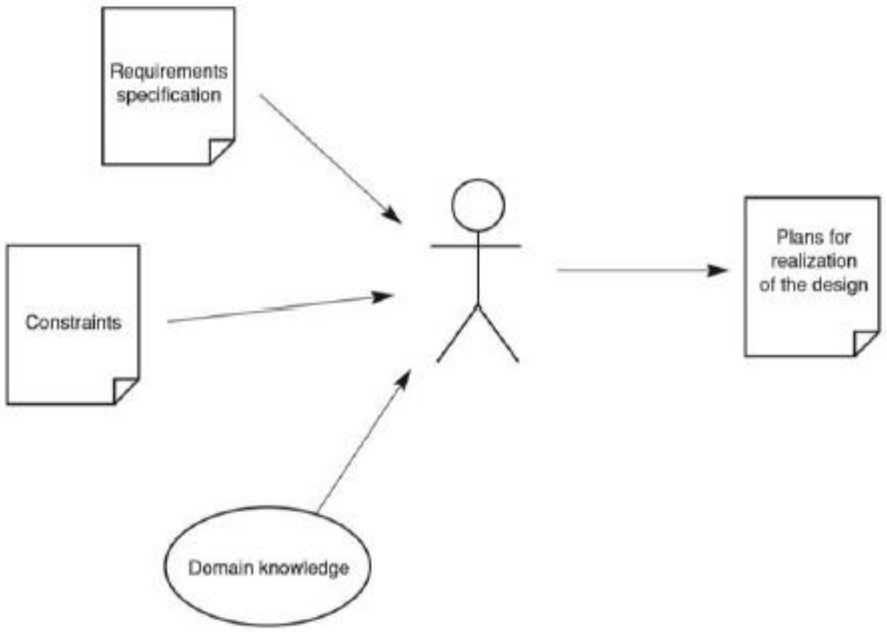
Topic: 14

Overview
In this topic, you will study the different inputs of the design process, for example, user requirements and system specification including constraints on design and implementation. You will study the Importance of domain knowledge.

Learning Goals
• Learn about the inputs of the design process
• Learn about the importance of domain knowledge

Constraint
A constraint is a decision over which you have little or no control as a designer/ architect. Your job is just to satisfy to design the best system that you can, despite the constraints you face. Sometimes you might be able to argue for loosening a constraint, but in most cases, you have no choice but to design around the constraints.

Designer's Communication Channels
A software designer may need to acquire some degree of 'domain knowledge' as a routine part of the input needed for undertaking any particular design task. Below figure present, the different communication channels (Inputs to the design process) through which the designer acquire the information needed in the design process.


Requirements specification
Constraints
Domain knowledge
Plans for realization of the design

Topic: 15

Overview

In this topic, you will study; desirable design internal characteristics, poorly designed programs difficult to understand and hard to modify, ease of maintenance one of the prime objective, code will be read more often than it is written, minimal complexity, keep it simple, system should be extendable, engineering is all about balancing conflicting objectives, loose coupling, reusability and portability.

Learning Goals

• Learn about the desirable internal design characteristics
• Learn about ease of maintenance, minimal complexity and portability

1, Maintainability

As systems get larger and more costly, the need to offset this by ensuring a long lifetime in service increases in parallel. In order to achieve this, designs must allow for future modification.

2, Simplicity

A characteristic of almost all good designs, in whatever sphere of activity they are produced, is a basic simplicity. A good design meets its objectives and has no additional embellishments that detract from its main purpose.

3, Reusability Phenomenon

The concept of reuse makes an explicit appearance in the definition provided by Brown and Short, and implicit appearances in the others. Strictly, of course, reuse is not an essential characteristic for a component, only a desirable one. If a large system contains one or two uniquely crafted components among a set of reused ones, then this can be considered as a quite pragmatic design decision that reduces the detailed design problem to that of designing only a few components, rather than a whole system

Jones says,

"*Most experienced programmers have private libraries which allow them to develop software with about 30% reused code by volume. Reusability at the corporate level aims for 75% reused code by volume, and requires special library and administrative support. Corporate reusable code also implies changes in project accounting and measurement practices to give credit for reusability*"

Van Snyder points out,

"*We conjecture that barriers to reuse are not on the producer side, but on the consumer side. If a software engineer, a potential consumer of standardized software components, perceives it to be more expensive to find a component that meets his need, and so verify, than to write one anew, a new, duplicative component will be written. Notice we said perceives above. It doesn't matter what the true cost of reconstruction is. Reuse has been successful for mathematical software for two reasons: (I) It is arcane, requiring an enormous intellectual input per line of code; and (2) there is a rich and standard nomenclature, namely mathematics, to describe the functionality of each component. Thus, the cost to reconstruct a component of mathematical software is high, and the cost to discover the functionality of an existing component is low. The long tradition of*
professional journals publishing and collecting algorithms, and offering them at modest cost, and commercial concerns offering very high-quality algorithms at somewhat higher but still modest cost, makes discovering a component that meets one's need simpler than in many other disciplines, where it is sometimes not possible to

specify one's need precisely and tersely. These factors collaborate to make it more attractive to reuse rather than to reinvent mathematical software."

Parnas writes,

"Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation." [Book: The Mythical Man-Month]

4, Portability

The system's level of independence on software and hardware platforms. Systems developed using high-level programming languages usually have good portability. One typical example is Java — most Java programs need only be compiled once and can run everywhere. [Book: Software Architecture and Design Illuminated]

Topic 16

Overview

In this topic, you will study; design must fulfill user's requirements, comparison of different designs, assessment features of designs, an important feature of assessment design for example, maintainable design, and understandability of designs.

Learning Goals

- Learn about the features of a good design

What are Characteristics of a Good Software Design?

Six characteristics of good software design:

- simplicity,
- coupling,
- cohesion,
- information hiding,
- performance,
- and security.

Topic:17

Overview

In this topic, you will study; coupling that is the measure of dependency between modules and components. It is the degree of interdependence between two modules and components. A dependency exists between two modules/components if a change in one could require a change in the other. Coupling is inter-component relationship. How to measure coupling and complexity of each interface?
Why high degree coupling is bad, because highly coupled systems have strong interconnection. These systems are difficult to test, reuse and understand separately. Loosely coupled modules are less dependent upon each other and hence are easier to change you can say, low coupling can be achieved by eliminating unnecessary relationship and reducing the number of necessary relationships.

Learning Goals

• Learn about the coupling
• Learn about measuring the coupling
• Learn about difficulties in coupled systems
• Learn about reducing the coupling

Key to Understand the Coupling

The key to understanding coupling is the word module found in the software design usage description. Unfortunately, the word module has been used to describe a wide range of physical objects (e.g., Apollo Lunar Module, modular home, power supply module) and software constructs (e.g., component, package, class, function).

Module For an object-oriented software design:

• A module is typically a class (i.e., a group of logically related attributes and methods). For example, a class that contains methods that validate data entered by a user. For very large software systems, a module may be a package of classes.
• A less commonly used interpretation is that a module is a single method found in a class definition.

Module For a structured software design:

• A module is typically a group of logically related functions. For example, a set of functions that perform validation of data entered by a user could be considered a distinct module within a structured design. This type of module is also known as a component (or in a very large software system a subcomponent). For example, a Python source code file is called a module. Thus, each Python module should contain logically related functions.
• A less commonly used interpretation is that a module is a single function found in a source code file.

Coupling

The coupling definitions indicate that coupling describes the amount of connectedness between two or more modules. Thus, a design that has only one module cannot exhibit coupling. That is, coupling can only be described between distinct modules. Our intuition suggests having fewer connections between modules mean there is less to test, maintain, and fix. Having fewer connections between modules is called low coupling, loose coupling, or weak coupling. A good design is one that exhibits low coupling between its modules. In contrast, more connections between modules means there is more to test, maintain, and

fix. Having more connections between modules is called high coupling, tight coupling, or strong coupling. A bad design is one that exhibits high coupling between its modules.
[Book: Guide to Efficient Software Design by David P. Voorhees, Publisher: Springer; 1st ed. 2020 edition]

Topic:18 – 21

Overview

In this topic, you will study; coupling relevancy with maintainability or modification. Reflection of coupling in code by web application example.

Learning Goals

- Learn about coupling by code example
- Learn about basics of cohesion

Topic:22

Overview

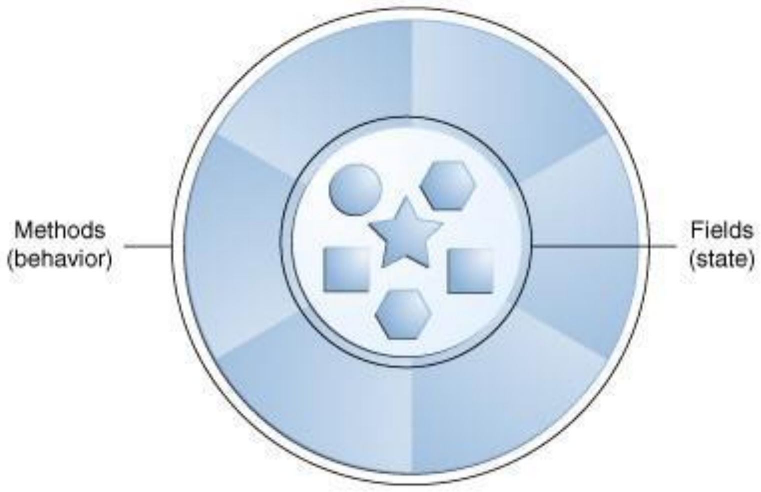
You will study about the object, its state, behavior and identity.

Learning Goals

• Learn about the object

Object

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle. Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, and breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, and current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming. Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



Topic:23

Overview

In this topic, you will study about the class. Class represent similar objects in object orientation. Class is a template of an object that includes properties and behavior.

Learning Goals

- Learn about the class in object orientation

The Class

In the real world, you will often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

Topic:24

Overview

You will study about the relationship in object orientation Concepts

Learning Goals

• Learn about the relationship in object orientation Concepts

Relationships in Object Oriented Concepts

If you are building a house, things like walls, doors, windows, cabinets, and lights will form part of your vocabulary. None of these things stands alone, however. Walls connect to other walls. Doors and windows are placed in walls to form openings for people and for light. Cabinets and lights are physically attached to walls and ceilings. You group walls, doors, windows, cabinets, and lights together to form higher-level things, such as rooms. Not only will you find structural relationships among these things, you'll find other kinds of relationships, as well. For example, your house certainly has windows, but there are probably many kinds of windows. You might have large bay windows that don't open, as well as small kitchen windows that do. Some of your windows might open up and down; others, like patio windows, will slide left and right. Some windows have a single pane of glass; others have double. No matter their differences, there is some essential "window-ness" about each of them: Each is an opening in a wall, and each is designed to let in light, air, and sometimes, people. In the UML, the ways that things can connect to one another, either logically or physically, are modeled as relationships.
In object-oriented modeling, there are two kinds of relationships that are most important: inheritance and associations.

[Book: Unified Modeling Language (UML) User Guide By Grady Botch James Rumbaugh Ivar Jacobson]