605.601 Foundations of Software Engineering
Fall 2020

Module 06: Software Design – Advanced Concepts

Dr. Tushar K. Hazra

tkhazra@gmail.com

(443)540-2230

605.601 Foundations of Software Engineering
Course Module 06: Design II

Topics for Discussion

- Component Level Design
- Design – Advanced topics – Patterns, and Frameworks

Major Steps in Design

- Moving from a high-level view of system to implementation-level details. . .
  1. Representation of system architecture
  2. Modeling of interfaces (with users, other systems, and internal components)
  3. Design of individual components
- The result is an early representation of software that can be assessed for quality

605.601 Foundations of Software Engineering
Course Module 06: Design II

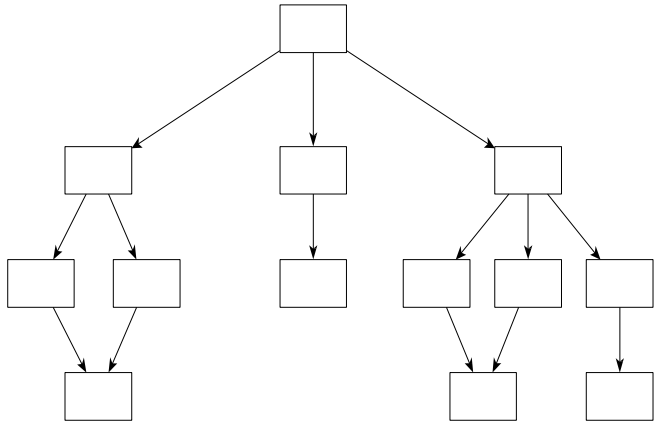
Component-Level Design

- What is a Component?
- Different Views
- Design Guidelines
- Component-level Design
- Frameworks

- What is a Component?
  - A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces [OMG, 2003]
  - Pieces [of software] that are independently purchasable and upgradable [Fowler, 2004]

605.601 Foundations of Software Engineering Course Module 06: Design II

- Different Views
  - Conventional
    - A component is a functional element that encompasses processing logic
  - Control component Coordinates interaction among domain components
  - Domain component Implements required functionality
  - Infrastructure component Supports processing
  - Object Oriented
    - A component comprises a set of collaborating classes



- Design Guidelines
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the open-closed principle)
  - It is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes)

- Cohesion
  - The "single-mindedness" of a module
  - A component encapsulates only attributes and operations that are closely related to one another and to the component itself
- Levels of cohesion
  - Coincidental Arbitrarily grouping (e.g., "utilities" class)
  - Logical Logically categorized as doing the same thing (e.g., mouse and keyword input)
  - Communicational Grouping based on operations on the same data
  - Functional All parts of module contribute to well-defined task

Coupling

- A qualitative measure of the degree to which classes are connected to one another

Levels

- Content Component modifies another's internal data Common Components use shared global variables
- Control flow dictated by flag passed to function
- Stamp Component used as a type for an argument to an operation Data Components exchange many data arguments

Levels (continued)

- Routine Call Operation invokes another operation
- Type use Component uses a data type defined by another component

  Inclusion Component imports a package or module

Note: You are not expected to memorize the coupling levels

Component Level Design I

- Identify all design classes that correspond to the problem domain.
- Identify all design classes that correspond to the infrastructure domain.
- Elaborate all design classes that are not acquired as reusable components.
  1. Specify message details when classes or component collaborate.
  2. Identify appropriate interfaces for each component.
  3. Elaborate attributes and define data types and data structures required to implement them.
  4. Describe processing flow within each operation in detail.

Component Level Design II

- Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Develop and elaborate behavioral representations for a class or component.
- Elaborate deployment diagrams to provide additional implementation detail.
- Factor every component-level design representation and always consider alternatives.

Designing Conventional Components

- The design of processing logic is governed by the basic principles of algorithm design and structured programming
- The design of data structures is defined by the data model developed for the system
- The design of interfaces is governed by the collaborations that a component must effect

Algorithm Design Model

- Represents the algorithm at a level of detail that can be reviewed for quality
- Graphical (e.g., flowchart or box diagram)
- Pseudocode
- Programming language
- Decision table

- Structured Programming
  - Uses a limited set of logical constructs
    - .., Sequence
    - .., Conditional: if-then-else, select-case
    - .., Loops: do-while, repeat until
  - Leads to more readable, testable code
  - Can be used in conjunction with "proof of correctness"
  - Important for achieving high quality, but not enough

Basic Design Principles

- Open-Closed Principle A module [component] should be open for extension but closed for modification.
- Liskov Substitution Principle Subclasses should be substitutable for their base classes.
- Dependency Inversion Principle Depend on abstractions. Do not depend on concretions.
- Interface Segregation Principle Many client-specific interfaces are better than one general purpose interface.

Basic Design Principles (continued)

- Release Reuse Equivalency Principle The granule of reuse is the granule of release.
- Common Closure Principle Classes that change together belong together.
- Common Reuse Principle Classes that aren't reused together should not be grouped together.

Component-Based Development

- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within the architecture of the system to be built?

- Impediments to Reuse
  - Few companies and organizations have anything that resembles a comprehensive software reusability plan.
  - Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them.
  - Relatively little training is available to help software engineers and managers understand and apply reuse.
  - Many software practitioners continue to believe that reuse is "more trouble than it's worth."
  - Many companies continue to encourage of software development methodologies which do not facilitate reuse.
  - Few companies provide an incentives to produce reusable

- Identifying Reusable Components
  - Is component functionality required on future implementations?
  - How common is the component's function within the domain?
  - Is there duplication of the component's function within the domain?
  - Is the component hardware-dependent?
  - Does the hardware remain unchanged between implementations?
  - Can the hardware specifics be removed to another component?
  - Is the design optimized enough for the next implementation?
  - Can we parameterize a non-reusable component so that it becomes reusable?
  - Is the component reusable in many implementations with only minor changes?

- Design Patterns
  - Each of us have encountered a design problem and silently thought: I wonder if anyone has developed a solution for this?
  - What if there was a standard way of describing a problem and an organized method for representing the solution to the problem?
- Design Patterns – a definition
  - A codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

- Design Patterns
  - Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice. (Christopher Alexander, 1997)
  - "a three-part rule which expresses a relation between a certain context, a problem, and a solution."

- Basic Concepts
  - Context allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.
  - A set of requirements, including limitations and constraints, acts as a system of forces that influences how
    - ... the problem can be interpreted within its context and
    - ... how the solution can be effectively applied.

- Describing a Pattern
  - Pattern name: Describes the essence of the pattern in a short but expressive name Problem Describes the problem that the pattern addresses
  - Motivation: Provides an example of the problem
  - Context: Describes the environment in which the problem resides including application domain
  - Forces: Lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered
  - Solution: Provides a detailed description of the solution proposed for the problem
  - Intent Describes the pattern and what it does

- Describing a Pattern (continued)
  - Collaborations: Describes how other patterns contribute to the solution Consequences Describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern
  - Implementation: Identifies special issues that should be considered when implementing the pattern
  - Known uses: Provides examples of actual uses of the design pattern in real applications

- Kinds of Patterns
  - Architectural patterns describe broad-based design problems that are solved using a structural approach.
  - Data patterns describe recurring data-oriented problems and the data modeling solutions that can be used to solve them.
  - Component patterns (also referred to as design patterns) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture
  - Interface design patterns describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end-users.

- Creational Patterns
  - . . . focus on the "creation, composition, and representation of objects"
  - Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - Factory Method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - Singleton: Ensure a class only has one instance, and provide a global point of access to it.

- Structural Patterns
  - . . . focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure
  - Adapter: Convert an interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.
  - Proxy: Provide a surrogate or placeholder for another object to control access to it.

- Behavioral Patterns
  - . . . address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects
  - Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  - Observer: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- Pattern-Based Design
  - A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system.
  - The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway.

- Thinking in Patterns
  - Shalloway and Trott suggest the following approach that enables a designer to think in patterns (Shalloway and Trott, 2005):

1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
2. Examining the big picture, extract the patterns that are present at that level of abstraction.
3. Begin your design with 'big picture' patterns that establish a context or skeleton for further design work.
4. "Work inward from the context" looking for patterns at lower levels of abstraction that contribute to the design solution.

- Thinking in Patterns (continued)
  - Repeat steps 1 to 4 until the complete design is fleshed out.
  - Refine the design by adapting each pattern to the specifics of the software you're trying to build.

- Design Tasks I
  - Examine the requirements model and develop a problem hierarchy.
  - Determine if a reliable pattern language has been developed for the problem domain.
  - Beginning with a broad problem, determine whether one or more architectural patterns are available for it.
  - Using the collaborations provided for the architectural pattern, examine subsystem or component level problems and search for appropriate patterns to address them.

- Design Tasks II
  - Repeat the preceding steps until all broad problems have been addressed.
  - If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.
  - Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.
  - Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.

- Common Design Mistakes
  - Not enough time has been spent to understand the underlying problem, its context and forces, and as a consequence, you select a pattern that looks right, but is inappropriate for the solution required.
  - Once the wrong pattern is selected, you refuse to see your error and force fit the pattern.
  - In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or erroneous fit.
  - Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.

- Design Granularity
  - When a problem involves "big picture" issues, attempt to develop solutions (and use relevant patterns) that focus on the big picture.
  - Conversely, when the focus is very narrow (e.g., uniquely selecting one item from a small set of five or fewer items), the solution (and the corresponding pattern) is targeted quite narrowly.

- Frameworks I
  - Patterns themselves may not be sufficient to develop a complete design
  - In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a framework, for design work.
  - That is, you can select a "reusable mini-architecture that provides the generic structure and behavior for a family of software abstractions, along with a context . . . which specifies their collaboration and use within a given domain." (Ambler, 1998)

- Frameworks II
  - A framework is not an architectural pattern, but rather a skeleton with a collection of "plug points" (also called hooks and slots) that enable it to be adapted to a specific problem domain.
    - .., The plug points enable you to integrate problem specific classes or functionality within the skeleton.