[Notes] Software Design and Architecture Specialization @University of Alberta

Xu, Bangyao
bangyaoxu0321@gmail.com

December 23, 2022

1 Object-Oriented Design

1.1 Object-Oriented Analysis and Design

- Software design looks at the lower level aspects of a system
- Software architecture looks at the higher level aspects of a system
- *Eliciting requirements* involves not only listening to what client is telling you, but asking questions to clarify what client hasn't told you
- *Conceptual mockups* provide your initial thoughts for how the requirements will be satisfied
- The clearer your conceptual design, the better your technical designs
- *Technical design* is done by splitting components into smaller and smaller components that are specific enough to be designed in detail
- Components turn into collections of functions, classes, or other components → represent a much simpler problem that developers can individually implement
- Certain design decisions involve trade-offs in different quality attributes such as performance, convenience and security
- Context is important to determine what choice of solution is right for the balance of qualities
- *Functional requirements* describe what the system or application is expected to do

- *Non-functional requirements* specify how well the system or application does what it does
- Other qualities to satisfy for the software can include reusability, flexibility and maintainability
- As design gets detailed and implementation is constructed, required quality should be verified through techniques like reviews and tests
- How structure is organized may affect performance as seen by users, as well as the reusability and maintainability as seen by developers
- Components, connections and responsibilities are identified from some requirements when forming the conceptual design
- CRC (Class, Responsibility, Collaborator) cards are used to record, organize and refine the components in the design
- *Collaborators* are other classes that the class interacts with to fulfill its responsibilities → Connections

1.2 Object-Oriented Modeling

- The goal during software design is to construct and refine models of all the objects
- Unified Modeling Language (UML) expresses models in a visual notation → Object-Oriented Modeling
- Apply design principles and guidelines to simplify objects → deal with complexity
- Each new programming language was developed to provide solutions to problems that previous languages were unable to adequately address
- COBOL and Fortran → Algol 68 and Pascal → C and Modula-2
- The goal of object-oriented design is to:
- 1) make an abstract data type easier to write, structure a system around abstract data types called *classes*
- 2) introduce the ability for an abstract data type to extend another by introducing a concept called *inheritance*

- Any object-oriented program is capable of representing real world objects or ideas with more fidelity
- Major design principles of object oriented programs: 1) Abstraction; 2) Encapsulation; 3) Decomposition; 4) Generalization
- *Abstraction* is the idea of simplifying a concept in problem domain to its essentials within some context → understand a concept by breaking it down into a simplified description that ignores unimportant details
- Rule of Least Astonishment → abstraction captures the essential attributes and behavior for a concept with no surprises and no definitions that fall beyond its scope
- Actual values of *attributes* may change, but attributes do not
- An abstraction should also describe a concept's basic *behaviors*
- *Encapsulation* forms a self-contained object by *bundling* the data and functions it requires to work, *exposes* an interface whereby other objects can access and use it, and *restricts* access to certain inside details
- *Methods* manipulate the attribute values or data in the object to achieve the actual behaviors
- Encapsulation helps with data integrity; Encapsulation can secure sensitive information; Encapsulation helps with software changes
- Black Box Thinking → provide inputs and obtain outputs by calling methods and it doesn't matter what happens in the box to achieve the expected behaviors
- Encapsulation achieves the *Abstraction Barrier* → since the internal workings are not relevant to the outside world, this achieves an abstraction that effectively reduces complexity for the users of a class
- *Decomposition* is taking a whole thing and dividing it up into different parts or taking a bunch of separate parts with different functionalities, and combining them together to form a whole
- A general rule for decomposition is to look at the different responsibilities of some whole thing, and evaluate how you can separate them into different parts, each with its own specific responsibility
- A whole might have fixed or *dynamic* number of a certain type of part

- A part can also serve as a whole containing further constituent parts
- One issue in decomposition involves the *lifetimes* of the whole object, and the part objects, and how they could relate $\rightarrow$ *sharing*
- *Generalization* is frequently used when designing algorithms, which are meant to be used to perform same action on different sets of data
- Generalization can be achieved by classes through inheritance $\rightarrow$ take repeated, common, or shared characteristics between two or more classes and factor them out into another class
- Subclasses will inherit attributes and behaviors from the superclass
- UML Class Diagrams are more suited for communicating the technical design of the software's implementation
- *Getter Methods* are methods that retrieve data, and their names typically begin with get and end with the name of the attribute whose value you will be returning
- *Setter Methods* change data, and their names typically begin with set and end with the name of the variable you wish to set
- Data integrity is why having Getter and Setter Methods
- There are three types of relationships found in decomposition: 1) association ('some'); 2) aggregation ('weak has-a'); 3) composition ('strong has-a') $\rightarrow$ define the interaction between the whole and the parts
- *Association* is a loose partnership between two objects that exist completely independently
- *Aggregation* is a weak has-a relationship between classes $\rightarrow$ One object has the other, but the objects are not heavily linked
- *Composition* forms a relationship that only exists as long as each object exists $\rightarrow$ the most dependent of the decomposition relationships
- The superclasses are the *generalized* classes, and the subclasses are the *specialized* classes
- Classes can have implicit constructors or explicit constructors
- A subclass's constructor must call its superclass's constructor, if the superclass has an explicit constructor

- In order to access the superclass's attributes, methods and constructors, the subclass uses the keyword called *Super*
- Subclasses can override the methods of its superclass, meaning that a subclass can provide its own implementation for an inherited superclass's method
- An interface only declares method signatures, and no constructors, attributes, or method bodies → It specifies expected behaviors in method signatures, but does not provide any implementation details
- *Polymorphism* is when two classes have the same description of a behavior, but the implementations of the behavior may be different
- A single implementation for multiple interfaces with overlapping contracts is acceptable
- Interfaces are meant to fulfill a specific need, which is to provide a way for related classes to work consistently

1.3 Design Principles

- *Coupling* captures complexity of connecting module to other modules
- Evaluating coupling of a module → degree, ease, and flexibility:
- *Degree* is the number of connections between the module and others
- *Ease* is how obvious are connections between the module and others
- *Flexibility* is how interchangeable other modules are for this module
- *Cohesion* represents the clarity of the responsibilities of a module
- In general, there's a balance to be made between low coupling and high cohesion in the designs
- Separation of concerns is an ongoing process throughout design process
- Increased modularity allows developers to reuse and build up individual classes without affecting others
- *Information hiding* allows models of our system to give others the minimum amount of information needed to use them correctly and hide everything else → allow a developer to work on a module separately

with other developers needing to know the implementation details of this module

- Information hiding through encapsulation allows us to change the implementation without changing the expected outcome
- *Access modifiers* change which classes are able to access attributes and behaviors and they also determine which attribute and behaviors a superclass will share with its subclasses
- *Conceptual integrity* → daily stand-up meetings, sprint retrospectives
- Code reviews are systematic examinations of written code
- Having a small core group that accepts commits to the code base is another approach in achieving conceptual integrity
- *Liskov Substitution Principle* → A subclass can replace a superclass, if and only if, the subclass doesn't change functionality of the superclass
- A sequence diagram describes how objects in the system interact to complete a specific task
- A state diagram can describe a single object and illustrate how that object behaves in response to a series of events in the system:
- 1) *Entry activities* are actions that occur when the state is just entered from another state
- 2) *Exit activities* are actions that occur when the state is exited and moves on to another state
- 3) *Do activities* are actions that occur once, or multiple times while the object is in a certain state
- *Model checking*: 1) modeling phase, 2) running phase, and 3) analysis phase → make sure that not only creating well-designed software, but software that meets desired properties and behavior

2 Design Patterns

2.1 Creational & Structural Patterns

- *Creational Patterns* tackle how to handle creating new objects → Several different patterns are based upon creating and cloning objects

- *Structural patterns* describe how objects are connected to each other → Not only do structural patterns describe how different objects have relationships, but also how subclasses and classes interact through inheritance
- *Behavioral Patterns* focus on how objects distribute work → They lay out the overall goal and the purpose for each of the objects
- *Singleton pattern* refers to having only one object of a class which is globally accessible within the program
- By hiding the regular constructor and forcing other classes to, instead, call the public getInstance method → *gatekeeping*
- *Lazy creation* → the object is not created until it is truly needed
- Singleton pattern is achieve by having a private constructor with a public method that instantiates the class if it's not already instantiated
- A factory object is an instance of a class, which has a method to create product objects → coding to an interface, not an implementation
- The factory method design intent is to define an interface for creating objects, but let the sub-classes decide which class to instantiate
- *Factory method pattern* separates out object creation into a factory method → Each subclass must define its own factory method
- *Facade design pattern* provides a single simplified interface for client classes to interact with the subsystem
- A facade is a wrapper class that encapsulate the subsystem in order to hide the subsystem's complexity
- *Adapter design pattern* will help facilitate communication between two existing systems by providing a compatible interface
- Just because an interface doesn't conform to what the system is expecting, doesn't mean that the system has to change
- *Composite design pattern* achieves 2 goals: 1) compose nested structures of objects, and 2) deal with the classes for these objects uniformly
- *Composite class* is used to aggregate any class that implements the component interface and *leaf class* represents a non-composite type

- Composite design pattern is used to solve the issues of how to build a tree-like structure of objects, and how to treat the individual types of those objects uniformly $\rightarrow$ achieved by enforcing *polymorphism* across each class through implementing an interface or inheriting from a superclass

- *Proxy design pattern* allows a proxy class to represent a real subject class $\rightarrow$ hides a reference to an instance of the real subject class

- The proxy design pattern is useful when you need to defer creating resource intensive objects until needed, control access to specific objects, or when you need something to act as a local representation of a remote system

- *Decorator design pattern* uses aggregation to combine behaviors at runtime $\rightarrow$ Aggregation is used to represent a "has a" or "weak containment" relationship between two objects

- Decorator serves as the abstract superclass of concrete decorator classes that will provide an increment of behavior

- Not only does the decorator design pattern let you dynamically modify objects but it also reduces the variety of classes you would need to write

2.2 Behavioural Design Patterns

- *Template method* defines an algorithm's steps generally, deferring the implementation of some steps to subclasses

- *Chain of Responsibility* is a series of handler objects that are linked together $\rightarrow$ These handlers have methods that are written to handle specific requests

- The intent of Chain of Responsibility design pattern is to avoid coupling the sender to the receiver by giving more than one object the chance to handle the request

- The *state pattern* is primarily used when you need to change the behavior of an object based upon the state that it's in at run-time

- Instead of having these objects directly communicating with each other, the *command pattern* creates a command object in between the sender and receiver

- A command manager can also be used which basically keeps track of the commands, manipulates them and invokes them
- In the *mediator pattern*, an object will talk to all of these other objects and coordinate their activities
- *Observer pattern* makes it easy to distribute and handle notifications of changes across systems, in a manageable and controlled way

2.3 Working with Design Patterns & Anti-patterns

- The *MVC (Model View Controller) pattern* divides the responsibilities of a system that offers a user interface into those three parts
- The defining feature of MVC pattern is separation of concerns between the back end, the front end and the coordination between the two
- The *model* contains the underlying data, state and logic that the user wants to see and manipulate
- The *view* presents the model information to the user in the way they expect it and allows to interact with it
- The *controller* interprets the user's interaction with elements in the view, and modifies the model itself
- *Liskov Substitution Principle*:
- If a class, S, is a subtype of a class, B, then S can be used to replace all instances of B without changing the behaviors of a program
- The *open/closed principle* states that classes should be open for extension, but closed to change
- The open/closed principle is used to keep the stable parts of your system separate from the varying parts
- The *dependency inversion principle* states that high level modules should depend on high level generalizations and not on low level details
- Method calls in an object oriented design should be done using a level of indirection
- The *composing objects principle* states that classes should achieve code reuse through aggregation rather than inheritance

- Aggregation and delegation offer less coupling than inheritance
- The *interface segregation principle* states that 1) a class should not be forced to depend on methods it does not use, and 2) interfaces should be split up in such a way that it can properly describe the separate functionalities of your system
- The underlying idea of the *Law of Demeter* is that classes should know about and interact with as few other classes as possible
- First Rule: A method, M, in an object, O, can call on any other method within O itself
- Second Rule: A method, M, can call the methods of any parameter, P
- Third Rule: A method, M, can call a method, N, of an object, I, if I is instantiated within M
- Fourth Rule: Any method, M, in object, O, can invoke methods of any type of object that is a direct component of O
- According to *principle of least knowledge*, a method, M, of an object should only call other methods if they are:
- 1) encapsulated within the same object
- 2) encapsulated within an object that is in the parameters of M
- 3) encapsulated within an object that is instantiated inside of M
- 4) encapsulated within an object that is reference in an instance variable of the class for M
- *Refactoring* is the process of making changes to your code so that the external behaviors of the code are not change, but the internal structure is improved
- *D.R.Y. principle*: don't repeat yourself
- Having a long method can sometimes indicate that there is more occurring in that method than should be or it's more complex than it needs to be
- Comments can be an indicator of poor design

- *Data classes* are classes that contain only data and no real functionality → have getter and setter methods, but not much else
- *Data clumps* are groups of data appearing together in the instance variables of a class, or parameters to methods
- The best solution for *long parameter lists* is to introduce parameter objects → common occurrence in graphics libraries
- The *divergent change* code smells occur when you have to change a class in many different ways for many different reason
- *Shotgun surgery* → a change in one place requires you to fix many other areas of the code as a result
- *Feature envy* occurs when you've got a method that is more interested in the details of a class other than the one that it's in
- *Inappropriate intimacy*→ two classes talk really closely to each other → a method in one class calls methods of the other and vice versa
- *Primitive obsession* → overuse of the primitive types occur when you are not identifying obstructions and defining suitable classes
- *Speculative generality* occurs when you make a superclass, interface or code that is not needed at the time, but you think you may use it someday
- *Refused request* occurs when a subclass inherit something and doesn't need it

3 Software Architecture

3.1 UML Architecture Diagrams

- Software architecture defines 1) what elements are included in the system, 2) what function each element has, and 3) how each element relates to one another
- Software architecture provides many advantages that help *developers* to create and evolve software → It makes development easier by providing a strong direction and organization on what needs to be done

- A well-defined architecture will help *project managers* to understand tasks dependencies, impacts of change and coordinate work assignments
- A clear software architecture will help to communicate coherently and confidently to *clients* that you know what you're doing
- *End users* may not directly care about how the software actually works, but they will care that it works well for them
- *Kruchten's 4+1 view model*: Logical, Process, Development, Physical
- The *logical view* focuses mostly on achieving the functional requirements of a system
- The *process view* focuses on achieving nonfunctional requirements which specify the desired qualities for the system → These include quality attributes such as performance and availability
- The *development view* is concerned with details of software development and what is involved to support that → programming languages, libraries and tool sets & scheduling, budgets and work assignments
- The *physical view* handles how elements in the logical process, and development views must be mapped to different nodes or hardware for running the system
- *Scenarios* align with the use cases or user tasks of a system and show how the four other views work together
- *Components* are defined as independent, encapsulated units within a system → Each component provides an interface for other components to interact with it
- *Component diagrams* are a static view of the software system and depict the system's design at a specific point in its development and evolution → focus on the component in a system, not their methods and specific implementations
- *Ball* - Provided interface vs. *Socket* - Required interface
- Steps to build a component diagram: 1) identify the main objects used in the system, 2) identify all of the relevant libraries needed for system, and 3) come up with the relationships found between these components

- *Package diagrams* show packages and the dependencies between them
- A package diagram can be useful in order to get a larger view of the system, particularly of namespaces and names and who knows whom
- A software *release* involves separate libraries, an executable, an installer, configuration files, and many other different pieces
- A *specification level diagram* gives an overview of artifacts and deployment targets, without referencing specific details like machine names
- An *instance level diagram* can map a specific artifact to a specific deployment target → identify specific machines and hardware devices → highlight the differences in deployments among development, staging, and release builds
- The purpose of the *activity diagram* is to capture the dynamic behavior of the system → map out the branching into alternative flows

3.2 Architectural Styles

- An *abstract data type* can be represented as a class that you define to organize data attributes in a meaningful way, so that related attributes are grouped together, along with their associated methods
- Object-oriented paradigm allows for inheritance among abstract data types → one abstract type can be declared an extension of another
- Main program and subroutine is a style that is fundamentally focused on functions → suitable for computation-focused systems
- A principle that arise from procedural programming paradigm is *one entry, one exit per subroutine*, which makes the control flow of the subroutine easier to follow
- A subroutine may be affected by data changes made by another subroutine during execution
- At the core of a data centric architecture are two types of components:
- 1) *Central data* is the component used to store and serve data across all components that connect to it

- 2) *Data accessors* are the components that connect to the central data component $\rightarrow$ make queries and transactions against the information stored in the database
- The data centric software architecture allows to:
- 1) store and manage large amounts of data into a central data component
- 2) separate the functionality of the data accessors
- 3) facilitate data sharing between data accessors through database queries and transactions
- The key characteristic of a *layered architecture* is that the components in a layer only interact with components in their own layer or adjacent layers
- Layering allows for separation of concerns into each of the layers
- The layered architecture can be relaxed by allowing for pass-through
- *Tiers* often refer to components that are on different physical machines
- *Request-Response* $\rightarrow$ The client requests information or actions, and the server responds
- A simple Client-Server relationship with one server and one set of clients is also often called a 2-Tier Architecture
- One of the principle advantages of the 2-Tier client/server architecture is it is very scalable
- A middle layer can take the role of managing application logic in accessing the database directly
- Systems based on *interpreters* can allow an end user to write scripts, macros or rules that access, compose, and run the basic features of those systems in new and dynamic ways
- Interpreter-based architecture is used in a variety of commercial systems because it provides users with flexible and portable functionality
- *Portability* is becoming more important with the rise of virtual machines and virtual environments

- Interpreters can be slow → Basic implementation spend little time analyzing the source code and use a line by line translate and execute strategy
- A pipe and filter architecture has independent entities called *filters* which perform transformations on data input they receive, and *pipes*, which serve as connectors for the stream of data being transformed
- The order in which the filters transform data may change end result
- Breaking system down into filters and pipes can provide greater reusability, decrease coupling, and allow for greater flexibility in the system
- *Events* can be signals, user inputs, messages or a data from other functions or programs → act as both indicators of change in the system and as triggers to functions
- *Event generators* send events and *event consumers* receive and process these events
- The defining feature of *implicit invocation* is that a functions are not in direct communication with each other → All communication between them is mediated by an *event bus*
- One way to implement the event bus is to structure the system to have a main loop that continually listens for events
- We say a system has *race conditions* when the behavior of the functions depend on the order in which they are called
- A simple binary *semaphore* may consist of a variable that toggles between two values, available and unavailable → Available indicates that the shared data is not in use, and unavailable indicates that the shared data is in use by a function → control access to share data
- Event based architectural style is well suited to interactive applications
- The most basic form of *process control* is called a *feedback loop*
- A feedback loop has four basic components: a sensor, a controller, an actuator and the process itself

3.3 Architecture in Practice

- Software architecture aims to combine software design patterns and principles in order to define the software's elements, their properties, and how the elements interact with each other
- A system needs to be able to operate and perform within a given *context*
- The quality of a system is determined by *quality attributes*:
- 1) Developer perspective:
- *Maintainability* determines how easy systems can undergo change
- *Reusability* allows to take functionality or parts from a system and use it in another system
- *Flexibility* is the ability to adapt to future requirements changes in a timely and cost efficient manner
- *Modifiability* determines the ease at which the system is able to handle changes to functions, incorporating new functionality, or remove existing ones
- *Testability* measures how easy it is to demonstrate errors through executable tests
- *Conceptual Integrity* includes consistency across the entire system, such as following naming conventions
- 2) User perspective:
- *Availability* is the amount of time the system is operational over a set period of time
- *Interoperability* is the ability to understand interfaces and use them to exchange information under specific conditions with external systems
- *Security* is how well the system is protected from unauthorized access
- *Performance* determines how well the system is able to respond to a user command or system event
- *Usability* determines how well your system is able to address the requirements of the end users

- Architecture is not good or bad → It is a matter of selecting the appropriate architectural solution for the problem
- All quality attributes use *quality attribute scenarios* to determine if a system is able to meet the requirements that are set for the quality attribute → *general* and *concrete*
- Each scenario consists of a stimulus source, a stimulus, an artifact, an environment, a response, and a response measure
- Scenarios involving incorrect input, heavy system loads, or potential security breaches should be prioritized highly
- Architecture Tradeoff Analysis Method (ATAM) involves 3 different groups of participants:
- 1) Evaluation team, which has 3 different subgroups: designers, peers, or outsiders → unbiased
- 2) Project decision makers, includes project managers, clients, products owners, software architects, and technical leads
- 3) Architecture stakeholders, includes end users, developers, and support staff
- ATAM process: 1) Present the ATAM; 2) Present the business drivers; 3) Present the architecture; 4) Identify the architectural approaches; 5) Create a quality attribute tree; 6) Analyze the architectural approaches; 7) Brainstorm and prioritize scenarios; 8) Re-analyze the architectural approaches; 9) Present the results
- ATAM doesn't require to have intimate knowledge of the system → It involves all important stakeholders, and it puts emphasis on the system's quality attributes → Knowing the risks, sensitivity points, and tradeoffs of the system is important
- *Conway's law*: A system will tend to take a form that is congruous to the organization that produced it
- Extra work may be needed to provide unified and scalable architectures
- One way to harness code reuse for efficient development is to treat a group of products as a product line or product family

- If several products share characteristics, this means customers and developers can move from one to the other with less of a learning curve and fewer surprises
- *Commonalities* → features of product line stay same in every product
- *Variations* → features of product line that vary between products
- *Product-specifics* → features are specific to one and only one product
- *Domain engineering* is development of commonalities and variations
- *Application engineering* includes using commonalities, deciding which variations are necessary and integrating them into the product, and developing product-specific features
- The *reference architecture* is designed with respect to the needs of the software, except it now must take into account all the current products in the product line
- Reference architecture must also include the capacity for variation → 3 general techniques to realize variations in a system: 1) adaptation, 2) replacement, and 3) extension
- The extra resources and the common code in product lines can be used for all kinds of good → reliability, user experience, security, time to market, and maintainability

■