Contents

1 Software modeling & Use cases

A software model is a shared artifact that describes a system and helps communication. Software design is about improving the simplicity, effectiveness, and maintainability of the software system (this is because software can have a lot of dependencies that interlinks to be very complicated).

Bad software design involves having each package depend on every other package, creating high coupling. In practice, this can happen when developers are unconstrained.

Our goal includes

- Software modeling, creating tangible but abstract representation of a system that can be used to communicate ideas
- Software design is about purposely choosing the behavior and structure of the system. Software behavior is how the system response the event, software structure is about how different parts communicate and work together.

Use Cases are text stories of actors using a system to meet goals. They are used to discover and record requirements. They emphasize user goals and perspective (who is using the system, what scenario is this, what are their goals).

Definitions for Use Cases in general

- SuD, system under discussion
- Actor, something with behavior (person, computer system, etc)
- Scenario (Use Case instance), a specific sequence of actions and interactions between some actors and the SuD
- Use Case, a collection of related success and failure scenarios that describes an actor using the SuD to support a goal

Kinds of actors for Use Cases

- The primary actor has their goals fulfilled through using the SuD. Their goals drive the user case design/implementation.
- The supporting actor provides a service to the SuD that helps the primary actor to fulfill their goal. They can help to clarify external interface/protocols.
- Offstage actors have an interest/stake in the behavior in the user case but is not a primary and supporting actor (tax agency). The use case must ensure that those interests are satisfied. They do not directly interact with the system.

Use cases can be in text (success stories and alternative stories, that is like a flow chart describing all likely scenarios), or they can be modeled as a use case diagram. They come in different levels of details like: brief, casual, and fully dressed.

- Brief, a block of text, no structure
- Casual, still text, structuring is by separating the main success scenarios and alternative scenarios.
- Fully dressed, like casual, but in steps, with control flow. Order alt scenarios by appending a,b,c. A star implies happening at any time.

Include can be used in use cases to refer to a smaller subroutine/function. Underline the included function name in the use case too. Often used to reduce duplication. These subroutine use cases don't have to be use cases themselves (level subfunction) and pass the use case test, but they will need a use case text describing the process.

A Use Case diagram gives context to a system and its environment (use cases it has)

- the primary actors on the left (person with name)
- the SuD as a box in the middle with circle use cases inside
- the supporting actors on the right (person with name, or `<<actor>>` with name for machines)
- lines connect actors to use cases that they belong in
- Include and extend lines between use cases (dotted arrows with `<<include>>` or `<<extend>>`)

Use Cases influence many aspects of a software project like design, implementation, management. They need to be strongly driven by the goals of the project/actors.

Include points from the base case to the special case. Extends points from the special case to the base case. Extends are only used when matching our software to the use case extensions, and should be removed after development. Te

The flow of user cases are

- Use case diagram
- Use case text, for each use case in the diagram
- System sequence diagram, the list of method calls for each use case between actors and system
- Operation contracts, the interface for each method

Guideline for finding useful use cases

- Boss Test, is the use case a valid response to a boss asking what you did today
- Elementary Business Process Test, a task performed by one person in one place at one time, in response to a business event, that adds measurable business value, and leaves everyone in a consistent state (basically a non negligible task that is an entire process)
- Size test, is it a single step or does it take many steps

2 OO Analysis

Object-oriented analysis comes after the use case requirements gather, and it is a process to create a description of the problem domain using objects. This process involves

- Analyzing use cases and identify objects/concepts
- Capture concepts and behaviors in static domain models and dynamic ssds

Note that the classes and objects here are conceptual, in that they are not software classes, they are read by business users.

2.1 Domain Model

The domain model

- Contains concepts, attributes, and associations considered noteworthy in the problem domain
- It is defined as a representation of real-world conceptual classes.
- A part of the Unified Process Business Object Model (UP Business Object Model) (the big picture containing all models), which is about explaining things and products important under a business domain
- Has a UML visual class representation. We only care about the conceptual part, no operations or implementations.
- A visual dictionary of the noteworthy abstraction, vocab, and information in the domain.

The UP Business Object Model contains

- Business modeling, domain model
- Requirements, use case model, system sequence diagram
- Glossary

A conceptual class is an idea, thing, or object. Not a data model that is information, but can be purely behavioral. Formally, it consists of a symbol, intension, and extension.

- Symbols are words/images that represent the concept class
- Intension is the definition of the conceptual class
- Extension is the set of examples of the conceptual class

To identify a conceptual class

- Identify based on existing models for common problem domains (use existing conceptual classes in a similar problem domain)
- Use a category list and find conceptual classes belong to them
- Identify noun phrases in use cases. Must be careful if natural language and cannot use a pure noun-to-class mapping

An association is a relationship between conceptual classes indicating something meaningful. We should include associations that are: significant, needed to be preserved for knowledge, and are from the Common Associations List.

An attribute is a logical data value of an object. It should only be included if the requirements suggests/implies the need to remember it. A derived attribute has a `/` infront the name.

On the UML diagram, conceptual classes are rectangles, attributes are inside the conceptual rectangles, and relationships are lines with multiplicities on each end. The labels on relationship have reading direction arrows that indicate direction to read the association.

There might be concepts that are very similar. If we include them as separate concept classes, we will have a lot of duplication in relationships, and we miss associations between these similar concepts. We should model this under a Generalization-Specialization Class Hierarchy

- Generalization is to identify common aspects among concepts, extracting the superclass (general concepts), and creating relationships with subclasses (specialized concepts). The superclass will then have a set of subclasses.
- Benefits of better expression of relationships, reducing repetitions, and improving readability.
- Denoted as a hollow arrow from subclass to superclass
- Not inheritance because that is software
- Have shared attributes in the superclass, and specific ones in the subclass.

Subclasses vs attributes for modeling similar concepts

- We use subclasses if the specialized concepts have different behaviors (different set of attributes) and relationships (with other concepts)
- We use attributes if the different enums have the same behavior except for some changes in constraints or limits

In other words

- subclass has additional attributes
- subclass has additional associations
- subclass are handled differently to other classes
- when modeling, declare superclass abstract, and append the superclass name to the subclass

2.2 System Sequence Diagram

A SSD is a visualization of how the sequence of system events that external actors generates interacts with the system, including inter-system events, and also noting the order of such events. Importantly

- One SSD for one scenario of a use caes
- Helps to identify all external system events
- Treats the system as black boxes when handling events
- Captures of the dynamic aspect of the problem domain or system.
- Keep the events as abstract as possible

The diagram for a single use case scenario contains

- External primary actor is a person top left, System as a box top right
- Going down are back and forth system events
- Primary actor events are in solid arrows, system responses are in dotted arrows
- Events contains a name and parameters via `Name(param1, param2)`
- Responses are optional with return values associated with the previous event like `ret1, ret2`
- Contains control flow boxes (uml frames) like: `loop [condition]`, `ref [ssd]`, `alt [a] [b]`, `optional [condition]`

- If more than one system supporting actors, list the other systems to the right of the system (with the `<<actor>> prefix`). If more than one primary actors, list the actors to the left of the external actor
- If don't want to write the response event, can use `res = event(param)` syntax.

We should make the SSD event reference attributes/relationships in the domain model.

We can make a SSD directly from a use case text. It is easier if it is also fully dressed

3 Object Oriented Design

Object oriented design model is

- A conceptual solution to a problem that meets the requirement
- Emphasizes defining software objects and their collaboration

Object oriented implementation are

- A concrete solution to a problem that meets the requirements
- Implementing the solution in OO languages and technologies.

In the UP model, the OO analysis and use cases will influence OO design

- Use case describes functional requirements that must be realized in design
- Domain model provides inspiration for names of software objects in design
- System Sequence Diagrams indicate interactions between users and system that must be implemented in design

OO Design is a process of creating a conceptual solution by defining software objects and their relationships

- Contains structure and connectedness (levels, architecture)
- Interfaces like fields/datatypes, methods, protocols (external and internal)
- Assignment of responsibilities (design patterns)
- Contains a static element and a dynamic element, Design Class Diagram and Design Sequence diagram

Layers in OO Design

- The layers are: UI layer, application layer, component/system layer
- We will focus on the application layer
- Because OO design of the core logical layer are similar across other applications, with essential OO design skills that are transferable
- Other layers are more technology/platform dependent.

3.1 Static Design Model, Design class diagram

A static design model (Design Class Diagram) is a representation of software objects with: class names, attributes (datatypes), method signatures, and associations. We'll use UML class diagrams to visualize the model.

For the Design Class Diagram

- Reference to OOSD for drawing and understanding the Design Class Diagram since they are identical
- Note that the directionality of relationships are now relevant

Compared to the domain diagram

- Domain model focuses on a conceptual perspective of the problem domain, what are the important concepts in the problem domain
- Design models focuses on an implementation perspective of the solution, what are the roles and collaborations of software objects
- Domain model should inspire the design model. Reducing the representational gap means reusing the same concepts in the domain model in the design model. It helps other stakeholders who understand the domain to understand the software representation

To code

- OO Design provides information necessary to generate the code, by mapping it to OO programs
- Attributes are fields; method signatures are java methods; responsibilities are handled in the constructors

3.2 Responsibility Driven Design

RDD is a way to design software objects

- Focuses on assigning responsibilities to software objects: the obligation or behaviors of an object
- Two types of responsibilities: knowing, doing
- Knowing responsibilities is about: knowing private encapsulated data, knowing about related objects, knowing about things it can derive
- Doing responsibilities is about: doing something itself like creating an object or calculation, initating actions in other objects, controlling and coordinating activities in other objects

Process

- Look at attributes in conceptual classes in domain model
- Declare knowing and doing responsibilities
- Create design model with such responsibilities

Note that

- Responsibilities are not methods, they are abstractions of architecture and methods (but methods implement the responsibilities)

3.3 Dynamic Design model, Design Sequence diagram

It is a representation of how software classes interact through messages. Uses the UML sequence and communication (not) diagrams to visualize. More concretely,

- It illustrates the sequence (time ordering) of messages sent between software objects
- A found message is a message whose sender is not specified
- An activation/execution bar denotes the duration where a message handler is running
- A message can have a parameter

Against the system sequence diagram

- SSD treats the system as a black box, and focuses on the interaction between actors and the system
- DSD illustrates the behavior inside the system, focusing on the interaction between software objects (a detailed part of an event in SSD towards the system)

RDD and design sequence diagram

- Helps in realizing and visualizing responsibilities of software objects
- A message implies a responsibility for the receiver class, and a responsibility of the sender class

Can directly translate DSD to code implementation of methods.

4 GRASP 1

GRASP, general responsibility assignment software patterns. It is defined as: a set of patterns/principles of assigning responsibilities to objects.

GRASP aids in creating designs in a methodical, rational, and explainable way. Given a specific category of problems, GRASP guides how you assign responsibilities to objects. Helps to support RDD.

Pattern is defined as a: named and well-known problem solution pair that can be applied in new contexts. It is a recurring successful application of design in a particular domain.

Advantages of patterns are

- Capture expertise and make it accessible to non-experts in a standardized form
- Make application of expertise easily to re-use
- Improve understand-ability
- Help communication among practitioners by providing a common language
- Make new solutions from a modified version of an existing pattern

4.1 Creator

Problem: who is responsible for creating a new instance of a class (doing responsibility)? This is important because creating objects is an important OO activity, and it is useful for maintenance and creates more opportunity for reuse (low coupling).

Solution: assign class B the responsibility for creating A if any of the following holds (the more true the better)

- B contains or aggregates A
- B records A (composition of A)
- B closely uses A
- B has the initializing data of A

If more than one class satisfies the above, prefer the aggregate class.

To use the creator pattern in the design model, we assign the creator an composite aggregation relationship with the createe. In the DSD, we add an event from the creator to the createe to create the createe.

Contraindications are examples or scenarios where we shouldn't apply the pattern. The contraindication for creators are

- If creating objects with significant complexity: using recycled instances for performance, or conditionally creating an instance from a family of similar classes depending on some external property (decision making).
- In such cases, delegate creation to a helper class like Concrete Factory or Abstract Factory rather than use the creator suggested class.

4.2 Information Expert

Problem: a general principle of assigning responsibilities to objects. Basically, given an object, which basic responsibilities should we assign to it.

Solution: assign the responsibility to the object if it has the information to fulfill that responsibility. This pattern is useful for understandability, maintainability, and extendibility.

To assign the class given the responsibility, find it in the domain model and replicate it. Otherwise, make it in the design model.

Contraindications,

- The solution by expert can be undesirable due to the resulting dependencies
- For instance, letting Sale to do the DB storing logic will create an undesirable dependency of the database logic in Sale, which reduces it coherence, and also creates some representational gap due to incoherence
- Doing this to all DB storing classes will create a lot of dependency on the specific DB tech, which is poor maintainability
- Should use a DB expert class that abstracts the DB logic for all classes instead.

4.3 Coupling

Problem: how to support low dependency, low change impact, and increased reused of objects?

Coupling is how strongly one object is connected to (knowing or depend on) other objects. Low coupling is when elements don't depend on too many other elements. High coupling creates objects that are hard to understand in isolation, hard to reuse, and contains high impact change.

Solution: assign responsibilities in a way to avoid unnecessary coupling. Use low coupling to evaluate alternatives. Low coupling will minimize dependencies, increasing maintainability, and increasing code reuse.

If an extra dependency exists between two objects, changing any one of the object will also change the dependency object.

In general, low coupling

- An evaluation principle that should be kept in mind always during design
- Allows classes in designs to be more independent, reducing change impact
- Avoid unnecessary dependencies

Contraindication. High coupling with stable and well understood concepts (using standard library everywhere) is usually not a problem, since there are few propagating changes, and the library concept is usually understandable and overall it is maintainable.

4.4 Cohesion

Problem: how to keep objects focused, understandable, and manageable (and also, support low coupling).

Define functional cohesion as how strongly related and focused the responsibilities of an object are. Low cohesion implies that a class has many unrelated things and does too much work, making the code hard to understand, reuse, and maintain.

Solution: Assign responsibilities to avoid low cohesion. We should use high cohesion as a principle to evaluate alternatives, and achieving it will ease understanding the design and simplify maintainability.

Similar to low coupling, cohesion is an evaluative principle that should be used for all design decisions.

Contraindications

- Low cohesion is sometimes needed for the design to meet non-functional requirements like performance requirements

4.5 Controller

Problem: how to interface the UI with the domain model, without creating coupling.

Solution, create a controller class that acts as the interface between the UI layer and the domain layer. This improves cohesion and coupling. We either create a controller for each use case, or have a main controller facade that references other use-case controllers. Often, we name the use class controller as Use-case Handler.

5 GRASP 2

5.1 Indirection

Problem, assign responsibility while avoid direct coupling: how to de-couple objects so that low coupling is achieved and reuse potential is higher.

Solution, assign the responsibility to the intermediate object to mediate between other components so that they are not directly coupled. This intermediary creates an indirection between the surrounding components (the dependent depends on the indirection element, while the indirection element hides the questionable element).

For example

- Sale need to use an external tax calculator actor to calculate tax
- The responsibility of computing tax assigned to sale implies a lot of external non-sale related logic in the sale class, low cohesion.
- Using indirection, make a tax adapter intermediate class and assign it the responsibility of calculating tax using the external actor. The sale can then use the adapter as an indirection to compute tax.

Discussion

- Motivation is usually to lower coupling
- Old adage, most problems in cs can be solved by another level of indirection
- Counter adage, most problems in performance can be solved by removing another level of indirection

5.2 Pure Fabrication

Problem, which object should have the responsibility, when solutions from other patterns are not appropriate without violating high cohesion and low coupling.

While domain models inspire the design model, in many cases, assigning responsibilities based only on the domain model leads to poor cohesion or coupling.

Solution, assign a set of highly cohesive responsibilities to an artificial or convenience class that has no domain counterpart. (make up a class)

For example

- No class is responsible for rolling a dice without violating cohesion (game directly rolling dice will bloat the class)
- Fabricate a DiceRoller that rolls the dice. This doesn't exist in the domain but has the responsibility of rolling the dice

Contradindications

- Pure fabrication are often used for indirection: indirection is assigning any class as intermediary, pure fab is assigning an artificial class as intermediary
- Pure fabrication should not be used as an excuse to add new objects
- If overused, the design can have each class having only one responsibility, creating too many objects that create high dependencies

5.3 Polymorphism

Problem, handling alternatives based on type (behavior depends on type), creating swappable/pluggable software components (client-server relationship where we want to replace one without replacing the other)

One solution for handling type alternatives is conditional variation, where we make new alternatives and use if-then-else everywhere. This is bad.

Solution, when related behaviors depend on type, assign responsibility for the behavior in the type where the behavior varies using polymorphic operations. Polymorphic operations are operations with the same interface. Essentially make parent class have abstract method, and implement different behavior in child classes overriding the abstract method.

The idea is that we shouldn't use conditional logic to implement different behaviors of similar types.

Polymorphism often stems from generalization-specialization in conceptual domain models. We represent the general class as an abstract class, and the specialize class as derived classes

We can represent polymorphism in DSD using the ALT frame.

Discussion, Contraindication

- Polymorphism is a fundamental pattern in designing how systems deal with similar variations. Allows the design to easily be extended to handle new variations
- Designing systems with polymorphism for speculative future-proofing against unknown variations could be an unnecessary effort, if those variations are improbable and never implemented

5.4 Protected Variations

Problem, how to guard objects and systems against variations and instabilities in some elements, so that they don't have undesirable impacts on other elements. Example points of variation/change are

- Variation point, variations in existing systems or requirements (different squares in monopoly)
- Evolution point, speculative variations that can arise in the future

Solution, identify such variation points with predicted variations, assign responsibilities in class that creates a stable interface around it. Interface here means providing a consistent means to access the method

Example

- External tax system is an evolution point since their API may change.
- Assign responsibility to an interface tax adapter allowing creating multiple adapters in response to instability in external systems. Uses polymorphism here

Discussions

- A very important fundamental principle
- Root principle motivating most patterns

- Equivalent to the open closed principle: modules should be open to extension and closed to modification. This is because it promotes adding flexibility through interfaces at variation points, while encapsulating the instabilities in a few classes that are fixed
- Contraindictation, similar to polymorphism, can have speculative protected variations at evolution points that may never be used, thus creating high coupling. Sometimes it is better to just rework a brittle design

Overview of GRASP. An arrow from A to B implies that we can use A to achieve B. Low coupling is a way to achieve protected variation at a variation point. Polymorphism is a way to achieve low coupling through polymorphic operations.

6 State Machine

If an object have different behaviors based on its state/status, it is difficult to capture its behaviors using a static or dynamic model due to it needing long if-else statements.

State machine

- A behavioral model that captures the dynamic behavior of an object in terms of states, events, and state transitions
- State is the condition/status of the object at a moment in time
- An event is a significant occurrence that affects the object and changes it state
- A transition is a directed relationship between two states that an event can cause change from one state to another

UML state machine diagram

- The black dot implies the initial pseudo-state, it has a transition to the initial state without any events
- States are circular nodes
- Events are on top of state transitions
- Transitions are directed edges between states representing state change, they are arrows from the from state to the to state.

Types of object given their behaviors

- State-dependent objects reacts differently to events depending on the object's state
- State-independent object reacts to events the same in all states
- State-independent object with respect to an event reacts to that specific event the same in all states, but otherwise is state-dependent

Guidelines to when to apply state machine model

- If the object is state-dependent with complex behavior. The state machine here will model the behaviors of a complex reactive object. (Device controllers, transaction objects, role mutators)

- Complex reactive objects are less common for business information systems (systems that record information), and more common in communications and user control applications (communication protocol, ui page, sessions)

UML diagram extras

- Guards are pre-conditions to a transition: a transition can only take place if the guard is fulfilled
- Transaction actions is an extra action that the object does when a transition happens. In software, this represents invoking a method of the class during the state transition
- Summarized, if trigger event happened, check if guard passes, calls the transaction action, state transition

Nested states

- A state can contain nested substates. A substate inherit all the transitions of its superstate (enclosing state)
- Transitions into the superstate will transition into the initial state of the nested substates
- Transitions out of the superstate will apply to all substates (ie, an outwards transition will transition from any substate)
- In UML, superstate is a box, substates makes another uml state diagram inside the box

Choice pseudostate

- Representation of a dynamic conditional branch on events. It evaluates the guards of the outgoing transitions to decide which single transition to take. Represented as a diamond with no text
- Can have multiple outgoing transitions with guards
- Use an `[else]` guard to act as the default transition if no other guards are selected

Event types

- External events, event caused by something outside the system
- Internal events, event caused by something inside the system
- Temporal events, event caused by a specific date or time, often controlled by a timer

7 Facade, Decorator, Builder

The GOF book about design patterns were published in 1995.

Patterns exists in public architecture as a reusuable structure that fulfills a set of purposes with similar feature sets.

Patterns allows people to reuse solutions for common problems with solution from experienced experts.

Categories of GoF patterns

- Creational patterns, abstraction of object instantiation

- Structural patterns, how classes and objects are structured to achieve a bigger purpose, also how objects relate to each other
- Behaviroal patterns, how objects communicate and interact with each other

GRASP principles are generalizations of ideas captured in design patterns.

7.1 Facade

Problem: requires a common unified interface to a set of different implementations or interfaces within a subsystem. Otherwise, there may be undesirable coupling to many objects within the subsystem, and changing the subsystem will cause lots of issues.

Solution: define a single point of contact to the entire subsystem, the facade object that wraps the subsystem, while hiding the subsystem objects. This facade object is a single unified interface to this subsystem and is responsible in helping collaboration with this subsystem.

UML

- Facade can is an object (or singleton) that exports methods which are proxies to relevant methods of objects within the subsystem package
- Other packages should interact with the subsystem only through the facade methods. Can overload the method if there are multiple different underlying implementations
- Packages are denoted as tabs on a rectangle around classes in the package
- Visibility modifiers can be on class names

7.2 Decorator

Problem: how to dynamically add behavior to individual objects at run-time without changing the interface presented to the client. This is irrespective of the combination, the number of features, the behaviors added. Inheritence is not feasible because it is static and will only apply to one combination and one class.

Solution: encapsulate the original concrete object inside an abstract interface, then create a decorator with extra behaviors that also inherit from the interface and contain an concrete object to decorate with the same abstract interface. Use recursive composition to allow unlimited decorator layers to be added to each instance.

The client only cares about the abstract interface methods. But when an object is constructed using decorators, a single method call will trigger: concrete implementation, decorator 1, decorator 2, etc, via recursive composition.

An example

- Shapes with draw methods from the Drawable interface
- Concrete implementation in the form of a rectangle
- Decorator also Drawable that takes another Drawable and adds a red border.
- Decorator that adds a blue background
- Can make an instance that is a rectangle with red borders and blue backgrounds by creating the concrete implementation and wrapping it twice via decorators

The decorator can also be abstract. Abstract means an interface or abstract, concrete means an actual implementation.

7.3 Builder

Goal: simplify the creation and hide the representation of a complex object.

Problem: how can the same construction process create different variants of a complex object (abstact builder), how can we simplify creating a complex object that is independent to its representation.

Solution: encapsulate creating the parts of a complex object in a separate builder object. Other class will then delegate object creations to the builder objects instead of instantiating the object directly.

Steps are

- Create an abstract builder that outlines the building methods and exports a method for creating the object
- Use inheritence to create subclass concrete builders that implement the building methods
- A third class can use the concrete builder to create different variants of a complex object via the superclass creation method, without seeing the underlying implementation (which are hidden in the concrete builders)

8 Adapter, Concrete/Simple Factory, Singleton

8.1 Adapter

Adapter pattern

- Problem, how to combine incompatible interfaces and to provide a stable interface to similar services/components with different individual interfaces
- Solution, convert the original interface of the components to another shared interface, using an intermediate adapter object to translate between the interfaces. (interface not as the programming language interface, but service interface)
- The adapter pattern converts interfaces of classes to another common interface that the client expects. It allows us to add more similar services with different interfaces into the existing program
- Benefits of decoupling client from external service interfaces and implementations, and that external interface changes are encapsulated by changes only in the adapter

To use the adapter pattern

- Make an abstract adapter that contains abstract methods to represent the new common interface
- Make concrete implementations of the adapter for each different external service, and for each implementation, write the common interface methods using the respective external service interface/class

- The client can then make an adapter (the specific concrete adapter doesnt matter), and use the common interface methods exposed on that adapter to indirectly use the external services.

Without using the adapter, there will be high coupling between the client and the external services. It also exposes the design to external variations in the services. An adapter uses the GRASP concept of: indirection, pure fabrication, and protected variations using polymorphism.

Problems with adapters: who creates the adapter, and which concrete implementation of the adapter should be created? A potential answer uses high cohesion, in that creating adapters is application logic since adapters are not domain objects, so we use pure fabrication to make an adapter factory.

8.2 Concrete Factory

Factory pattern

- Problem of: who should create complex objects with special logic, when we want to separate the object creation logic out for cohesion
- Solution, create a purely fabricated object called a factory that handles the object creation logic

Simple factory benefits

- Simplification of GoF abstract factory
- Separate of creation responsibility into helper objects
- Hide complex creation logic
- Allows performance enhancing memory management strategies like caching or recycling in the factories

Simple factory implementation

- Make a factory class that has simple methods to create/return the object instance, often as an interface so the factory can return any implementation
- Hiding object creation logic (like choosing which concrete class to use) inside the factory class. For example, the factory class can query a file to select the implementation to use

Comparison with other design patterns

- Factory is a method that creates/returns an object
- Factory method pattern has the parent class use abstract factory methods in its non-abstract methods, and we use inheritence of the parent class to select and implement these abstract factory methods in the child classes
- Abstract factory pattern, make an abstract factory class exporting abstract factory methods. Concrete factory classes will then inherit the abstract factory class and implement the factory methods. The client can then use the abstract factory interface without needing to read the implementation of the factory
- Builder, builder patterns are more complicated versions of constructors. While factory patterns are for less complicated version sof constructors.

Issues of factories: who creates the factory and how do we access it, only one instance of the factory is needed, and how do we get visibility to this single instance everywhere? A potential solution is to use the singleton pattern to provide a single access point with global visibility.

8.3 Singleton

Singleton pattern

- Problem: exactly one instance of the class (singleton) is allowed, and this object need a global and single access point
- Solution: define a static method of the class that returns the singleton

Singleton benefits

- Ensure one instance, and a global access point to it
- The singleton itself creates the singleton object, and ensures that there is only one

Singleton implementation

- Add a private static attribute that stores the singleton
- Add a public static method that returns the singleton. Lazy initialization means to create this singleton when the method is used and the singleton does not yet exists.
- Clients can use the public static get instance methods to access the singleton
- Singletons are represented with a one in the class box of the design sequence diagram

Why aren't all singleton methods static so that we don't need to call get instance

- May want subclasses, and static methods can't be overriden as they are not polymorphic
- Remote communication mechanisms like RPC/RMI does not support static methods
- The class is not a singleton in all contexts

Can use a singleton on a factory to make a singleton factory.

9 GOF 3

9.1 Strategy, Template patterns

Strategy pattern

- Problem: how to design for varying but related algorithms, and design for the ability to change using these algorithms in runtime.
- Solution, define each algorithm/policy/pattern in their separate classes, with a common interface for the client to use. The relevant class instances are then passed into the strategy methods by the client
- The client, or context object (since it is the object that gives the strategy a context), will hold a strategy interface instance by expert information principle. When using the client, the client will call the strategy interface method that in order calles the subclass strategy method.

- Behavioral design pattern since it focuses on how the strategy classes interact with the remaining system, and such functionality can be changed in runtime. Doesn't use indirection, but does follow protected variation

Template pattern

- Problem, how to build generic components that are easy to extend and use without code duplication
- Solution, define a family of algorithms (these are subclasses), encapsulate each one and make them interchangable. Can design the invariant parts in the parent and leave the subclass to implement the variable behaviors

A template is a method that defines an algorithm in a series of steps, with some of the steps being abstract. This allows us to merge the shared algorithm part together.

9.2 Composite pattern

Composite

- Problem: how to combine policies, since when multiple policies are applicable, there must be a resolver for the policies.
- Some policies cannot be combined, and the goal is to find the optimal policy: either for the user or for the user. It treates a group of objects the same way (via polymoprhism) as a non-composition object
- Solution, define classes for composite that implements the same interface as atomic objects. Use recursive composition on a list of interface objects (many relationship with the interface). Exports a method that evokes the same method on all of a composite's child items.
- Similar to the decorator, except that it doesn't add new featuers but rather aggregate existing features/atomic policies

A composite strategy pattern combines the strategy pattern with the composite patterns. Each composite class wraps over a list of strategy interfaces, and also itself is a strategy. This composite strategy can combine each individual strategies, or select one given some parameter.

9.3 Observer pattern

Observer pattern

- Problem: different subscribers are interesteed in the state changes of a publisher, and want to react in their own unique way when the publisher publishes an event. The publisher also wants to maintain low coupling to the subscribers
- Solution: define an listener interface that the publisher holds and the subscriber implements. The subscriber will call the publisher to subscribe, and the publisher can notify the listener on event changes.
- The both the publisher and observer can be abstract

9.4 Summary of related GoF Patterns

Related patterns

- Composites are similar to decorators, in that they both use recursive composition to combine an arbitrary amount of object.
- The difference is that composites don't introduce new features but rather wraps the existing classes together by delegating method calls to its wrapped classes. The decorator pattern aims to add new features on top of existing classes/interfaces (and usually it is only one class).
- We often use decorators and composites together
- Adapater provides a different shared interface, while decorators provide an enhanced implementation of the interface
- Decorators allows us to change the skin of the object, while strategy allows us to change the inside of the object
- Both template method and strategy solve the problem of separating a generic algorithm from its details
- Template method uses inheritence, strategy pattern uses delegation
- Often subclasses calls the superclass methods, but in template pattern, the parent calls the child methods (the hollywood principle, don't call us, we call you)
- Calling a strategy object does not require a structured design like in template method

10 Architecture Analysis

GRASP and GoF patterns mainly focuses on designing software classes in the application logic layer. Architecture analysis focuses on a holistic overview of the entire system.

Software architecture

- The large scale organization of elements in a software system
- Ideally a design that meets high impact system requirements, supports business logic, and straightforward to maintain and change
- Involves a set of important design decisions: the structure elements (what should be software classes be), interfaces (how are the elements composed), collaboration (how these elements work to achieve business logic), and composition (how the elements are grouped progressively into larger subsystems)

Architecture analysis

- An activity that identifies factors which will influence the architecture, understand their variability and priority and resolve them
- Mostly focuses on non-functional requirements in the context of functional requirements
- Goal is to reduce the risk of missing a critical factor in the system design, and focusing more on high priority requirements, aligning product to business goals

Functional requirements are requirements that are required for the software to work. Non-functional requirements are requirements that is not required for the software.

The factors to identify

- Architecturally significant requirements are requirements that have a LARGE impact on the design when not considered
- Variation points, factors that can change
- Potential evolution points, factors that can be updated laters

Architecturally significant functional requirements

- Auditing
- Localization
- Printing
- Reporting
- Security

Non-functional requirements

- Usability like UI aesthetics
- Reliability, availability of the system
- Performance, response time, throughput
- Supportability, testability and maintability

Requirements can affect the design decisions

- Reliability and fault tolerance requirements implies designing for multiple backups
- Adaptability and configurability requirements implies that we need to provide adapters to handle variations
- Brand name and branding requirements implies that the design should accommodate the branding values

Steps for architectural analysis

- Identify the architectural factors within requirement analysis. Investigate factors in detail
- Analysze alternatives for each factor and create solutions. These are architectural decisions

Priorities depends on

- Inflexible constraints like safety
- Business goals
- Other goals

Architecture factor table

- Documentation that records the influence of factors, their priorities, and their variability and flexibility requirements

- Consists of: factor, scenarios, variability (current and future evolutions), factor impacts on stakeholders and architecture, priority, difficulty

Technical memo

- A document that records alternative solutions for noteworhty issues and decisions
- Contains: issue, solution summary, factors, solution, motivation, unresolved issues, alternatives

Summary

- Mainly focuses on system-level, large-scale problems that are resolved using large scale fundamental design decisions
- Need to address the interdependencies of factors and their trade-offs
- Must generate and evaluate alternatives

11 Logical Architecture

Logical arhitecture

- The large scale organization of software classes into packages, subsystems, and layers
- Ignores the hardware specifics like networking, computers, or operating systems
- Essentially LA defines the package that software classes are in

Layered architecture

- Layers are coarse grained grouping of classes that has a cohesive responsibility for a major aspect of the system.
- Examples are: UI, application layer, technical services (DB)
- A strict layered architecture has each layer only calling services by the layer below, like a network. Infrequent in software architectures
- Relaxed layered architecture has each layer calling multiple services from below layers. More frequent in information systems

UML package diagram

- Illustrates the LA
- Each box represents a layer
- One layer depends on another is represented as dashed arrow towards the dependency

Common Layers

- UI layer
- Application layer
- Domain layer
- Technical services layer
- Foundation layer

Guidelines in logical architectures

- organize large scale logical system into distinct cohesive layers from high application specifics to low general services
- Main seperation of concerns
- Collaboration in coupling from higher layers to lower layers, avoid lower layers coupling with higher layers

Benefits of layered architectures

- Changes will not ripple through system due to coupling
- Separating application and UI will increase reuse
- Increase reuse by seperating general technical services with business/application logic
- Low coupling across areas of concerns, allowing division of work in development

12 Software Development Process

UP artifacts

- Business modeling, Domain model
- Requirements, Use case model
- Design, architecture package uml diagram, interaction diagram, class diagram

Software development process

- Unified process is an iterative and incremental software development process. Very abstract series of steps
- Inception, elaboration, construction, transition. Don't need to worry about the specific terms in the UP like Milestone, release, increment, or production release.

Inception

- The initial short step to establish a common vision and basic scope of the project
- Answers questions like: what is the vision of this project, feasibility, buy or build, cost, stakeholder agreement
- Not a waterfall process, and inception is not creating all requirements and believable plans.

Outcome of inception

- Common vision and scope of project
- Creating business case that addresses cost
- Analysis 10% of all use cases
- Analysis of critical non-functional requirements
- Preparing development environment
- Maybe prototypes that clarify requirements

- Go or no-go decision

Inception artefacts

- Vision and business case describing high-level goals, constraints, business use-cases. An executive summary to
- Use-case diagram and use-case text
- Iteration plan describes what is done in the next elaboration step

Not inception if

- Took more than a few weeks
- Attempted to define most of the requirements
- Estimates are reliable
- Defined the software architecture
- Planned sequence of work fully
- Didn't produce a business case
- Wrote use-cases in detail. Or didn't write any use-case in detail.

Elaboration is where design and modelling usually happen

- The initial series of iterations to build the core architecture, resolves high risk elements, defining most requirements, estimating overall schedule and resources
- Should program and test the core/risky part of the software architecture, making an executable architecture
- Majority of requirements discovered and stabilized
- Major risks are mitigated
- Not a phase where the models are fully developed for implementation in the construction step

Elaboration artefacts

- Domain model, design model, software architecture document, data model, use-case storyboards
- Around 80% of use-cases are reliably defined

From requirements to design

- Iteratively doing the right thing in the local scope
- Provoking inevitable changes early. While iterative methods embrace changes, we should sort out the inevitable changes early in the iterations. This achieves a more stable goal and requirements
- Usual timeframes: a few days for use-case writing and domaining modelling; a few weeks for proof-of-concept, finding resources, planning, and setting up developement environment

Use-case iteration conversion

- An iteration should focus on a single use-case
- However, a use case may be too complex to implement in one iteration. Therefore, different parts of a single use-case can be allocateed to different iterations in series
- Sometimes and extend an iteration to include another use-case

How to design objects

- Code. From mental model to code via design-while-coding. Ideally with an IDE that supports refactoring
- Draw then Code. Drawing UML first then switching to Code
- Only draw. With tools generating everything from diagrams. Kinda a misnomer as it still involves a programming language attached to the diagram
- When using Draw then code, the drawing overhead should be worth the effort (by reducing refactoring and creating a cleaner/maintainable design)

Agile modelling

- Modelling with other developers with lightweight uml drawings
- Creating serveral static and dynamic models in parallel. Hand draw or use uml tools.
- Should be done near iteration start

Object design vs uml notation skills

- UML is simply a reflaction of the design decisions
- UML should use correct notation for good communication. However, object design skill is of the greatest importance, not UML notation skill
- Object design requires knowing the principles of responsibility assignment (GRASP) and design patterns (GOF)