COMP2511: Object Oriented Programming

Introduction to Java:

* Platform independence: Solved the problem at the time where different binaries were only compatible to certain machines with C code.
  + Write once (machine independent) 🡪 Run multiple times across multiple platforms
* Each platform needs a virtual machine (Java runtime environment) to run the byte code

Differences between Java and C:

* No explicit pointers
* Automatic memory management 🡪 Automatic garbage collection for unused memory
* Java compilers 🡪 Just in time java compilation can result in speedup over static C compilers
* Java multithreading is in base Java, whilst you need libraries for C
  + Just in time compilation + multithreading can result in faster Java code that optimises based on multithreading
  + Increasingly, Java is actually more performant than C

Design paradigms: see “Designing a class” (below)

Types of Programming:

* Procedural programming:
  + Action oriented
  + Groups of actions are formed into functions and functions are grouped to form programs
* OOP:
  + Create user-defined types called classes.
  + Classes contain data about the entity it represents and the methods that manipulate the data
  + Object: Instance of a user-defined type (i.e. instance of a class)
    - OOP encapsulates data (attributes) and methods (behaviours) into objects
    - Have the properly of information hiding

Inheritance:

* New classes (subclasses) are created from existing classes by absorbing their attributes and behaviours
  + Subclass: Extends the super class. Adds behaviours, properties, etc. from the super class
  + Things common between entities inherit those common properties from the super class
* Inheritance relationships form a tree-like hierarchical structure
* Method overwriting: Methods are defined by the closest method to the current class instance

Relationships:

* “Is-A” Inheritance relationship:
  + An object of a subclass is treated as an object of the super class
  + Use inheritance to model an “Is-A” relationship
    - DON’T USE inheritance unless it makes sense to do
    - E.G. Mathematically circles are a subset of all ovals. However, a circle class shouldn’t inherit from an oval class because an oval class can have methods to setHeight and setWidth 🡪 These attributes and methods conflict with the definition of a circle
* “Has-A” Association relationship:
  + Class object has an object of another class to store its state or do its work
    - “Has-A” reference to another object
  + These relationships are examples of creating new classes by composition of existing classes

Designing a class:

* Consider the functionality, different ways the object can be created, properties of data, etc.
* Always try to keep data private (local)
  + Only accessible through methods
* Break up classes with too many responsibilities.
  + Consider factoring out common attributes & behaviours between multiple classes into a superclass
  + Then use suitable “Is-A” or “Has-A” relationships

Domain Modelling using UML:

* Designing and modelling classes are one of the most important practices in OOP
  + How do we model the inter and intra-module relationships
    - Relationships between classes that comprise a module
    - Relationships between the modules themselves
  + How should the methods that interact between classes work?
    - Classes may have class-independent data
* Universal Modelling Language: UML class diagrams to represent domain models
  + Assumptions 🡪 Users can’t communicate their exact requirements
    - Clarification document to client so you do not get screwed on the final demo.
  + Types of relationships in UML:
    - Dependency: A depends on B
    - Aggregation: A class contains another class.
      * A is associated with B. B persists even if A is deleted
      * More associated with physical resources
    - Composition: More restrictive aggregation, contained class is integral to the containing class. (cannot exist outside of the container)
      * A is associated with B. B loses meaning / is discarded when A is deleted
      * More associated with virtual objects / resources
    - Undirected Association: Does not assume dependence.
    - Directed Association: Indicated which class has knowledge of the other

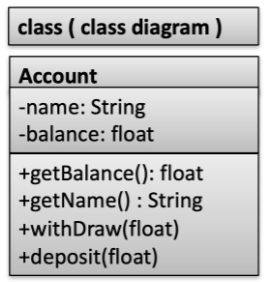


Figure 1 UML class representation

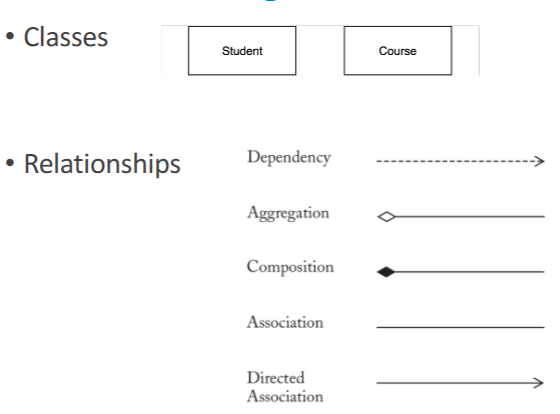


Figure 2 UML Diagram Syntax

* UML representations: see relationships above
  + Arrow: Inheritance = directed association
  + Add cardinality to relations (1:0..N, 1:1, etc.)
* Domain Models: Visually represent domain concepts and the relationships between them
  + Requirements analysis 🡪 External behaviour
    - Features and actors within the system
  + Domain modelling 🡪 Internal behaviour
    - How the components of the system produce the expected behaviour
  + ^ The two above concepts are mutually dependent
  + High-level abstractions should be used instead of technical details (low-level)

Attributes vs Classes:

* Should something be an attribute or a class?
  + If it cannot be represented by a primitive type (number, string) 🡪 Likely should be represented by a class

Approaches to subclasses and inheritance:

* Philosophy: Don’t duplicate code, replace duplication with relationships
  + “Has-A”: Uses method forwarding (methods from the has-a class to define some new methods)
  + “Is-A”: Extends super methods, modifications to super class propagate to your implementation
* Abstract class: Define framework of methods in super-class that acts like a common interface across all subclasses
* Interface: Classes that implement interfaces must implement all the interface functionality

Polymorphism: An object’s ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy

* Objects can be cast as the abstract class or the interface it implements, and these casting can result in different behaviours for the same method
* Dynamic binding: Methods are bound to objects after creation 🡪 Even after it is cast to the superclass

Method forwarding:

* If a class implements an interface, we can reuse code from other classes that implement the interface - called method forwarding
* Create new class that corresponds to another class that implements the interface method --> "Has-A" Relationship
* Allows us to selectively choose methods --> Sometimes we don't want to inherit every method from another class.

Method overriding:

* Methods in a class override methods in the superclass if the methods have the same name, return type, and position of typed arguments.

Method overloading:

* A single method name can have multiple implementations by varying the arguments or return types

Constructor:

* Super classes’ no argument constructor is implicitly added if the first argument of the constructor is not a super() or this() call
  + Calls super() up to the default Object class in java
* Constructors are implicitly inserted into a class if they are not defined

**Good Design Principles:**

* Assume software requirements can and will change
  + Flexibility and extensibility to accommodate these changes
  + Maintainable, reusable, extensible design
* Design smell: Violation of key design principles. We want to avoid these!
  + Rigidity: Too difficult to change
  + Fragility: Breaks in many places when a single change is implemented
  + Immobility: Hard to reuse
  + Viscosity: Changes are easier to implement through ‘hacks’ over ‘design preserving methods’
  + Opacity: Difficult to understand modules
  + Needless complexity / Repetition: Start with readable code.. You don’t need to introduce unnecessary complexity to save like 2 CPU cycles
  + Divergent change: One class is commonly changed in different ways, for different reasons
    - This denotes that there is too much responsibility that is placed on the class
  + Shotgun surgery: Opposite to divergent change 🡪 Lots of small changes to a lot of different classes
    - High coupling for one purpose
* Characteristics of good software:
  + Coupling: How strongly software elements are connected to other elements
    - Interdependence between components, high coupling = dependence on the internal workings of another component
    - Aim for loose coupling: Independent modifications and usage
  + Cohesion: How related the responsibilities of the software elements are within a software component
    - The degree to which all elements of a component or class work together
    - If many modules call a class
  + Loose coupling & high cohesion is good
* Test-driven development: Test cases before software 🡪 Immediate testing to make sure code is correct
* SOLID principles: General guidelines you might want to follow
  + Single responsibility principle: Many responsibilities in one class should probably be split
  + Open-closed principle: Open for extension, closed for modification
  + Liskov substitution principle: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program
  + Interface segregation principle: Many client-specific interfaces are better than one monolith interface
  + Dependency inversion principle: One should depend on abstractions, not concretions

Law of Demeter: Principle of least knowledge. You (class) talk only to your friends

* Classes should know about and interact with as few classes as possible
  + Assume as little as possible about other classes 🡪 Information hiding
* Reduce interaction to ‘immediate friends’ and ‘local objects’ 🡪 Aids loose coupling within the system
* Friends comprise of:
  + Object itself and variables in the scope/global
  + Objects passes as parameters
  + Objects instantiated within the method
  + Any component objects
  + NOT objects returned by a method

How to solve problems without inheritance: Inheritance causes all the methods to be featured 🡪 Can be a lot of additional baggage that does not make sense to include

* Delegation: Delegate the functionality to another class
* Composition: Reuse behaviours using one or more classes with composition
  + Design principle: Favour composition over inheritance

Liskov Substitution Principle: Subtypes must be substitutable for their base types

* Things that sound right in English doesn’t necessarily translate to code
  + E.G. Square “Is-A” Rectangle
  + But setHeight and setWidth methods for Square don’t make sense to substitute for rectangle because it does unexpected behaviour (length of all sides instead of just height/width)

A picture containing graphical user interface

Description automatically generated

Figure 3 LSP example

Method Overriding: Rules

* Access level cannot be more restricted
* Static methods are not overridden, as they are associated with a class

Restructuring: Changing the internal structure of software

Refactoring: Restructuring software to make it easier to understand and cheaper to modify without changing its external, observable behaviour

* When to refactor? A general guide
  + Adding functionality 🡪 Makes it easier for future changes
  + Bugfix
  + Code review
* Methods should be allocated based on where it makes sense to utilise the information
  + If the majority of data comes from a class, consider migrating the method to that class
  + “Can this implementation be encapsulated within another context that makes more sense?”
  + Can consider branching based on an attribute in another class as bad 🡪 This can be encapsulated in the class that holds the data
* Prior Goals: Have verbose unit testing before Refactoring
* Post-Refactoring: Unit tests still pass
  + External behaviour doesn’t change, internals of the refactored code can be changed
* Degrees of refactoring:
  + Low Level
    - Moving around code, variable declarations for ‘magic numbers’
  + High Level 🡪 Mostly only if the current code is bad
    - Fundamentally changing the design patterns
    - Changing language idioms (safety, brevity)
    - Performance optimization
    - ^ Proper testing is especially important here
* <https://refactoring.com/catalog> --> Examples of refactoring changes

Why do we need design patterns?

* Abstract class (inheritance)
  + Need to suppress methods if not all subclasses can use the functionality
* Interface
  + No code reusability if multiple internal implementations are needed to implement the interface methods
* What do you do? Use design patterns!
  + Strategy Patterns: Type of design pattern where you define a family of classes, encapsulate each one and then make them interchangable
    - Use Interface variables as attributes for some guaranteed functionality. Functionality is defined in the implementing classes.
    - Pass in objects, don’t use strings w/ switch statements
    - Pass functionality into subclasses / objects, create new objects to abstract away combinations of objects
      * E.G. Car(new electricMotor(), new drivingSystem()) 🡪 Tesla()
      * Passing in objects that implement certain interfaces

Design Pattern: Tried solutions to commonly recurring problems

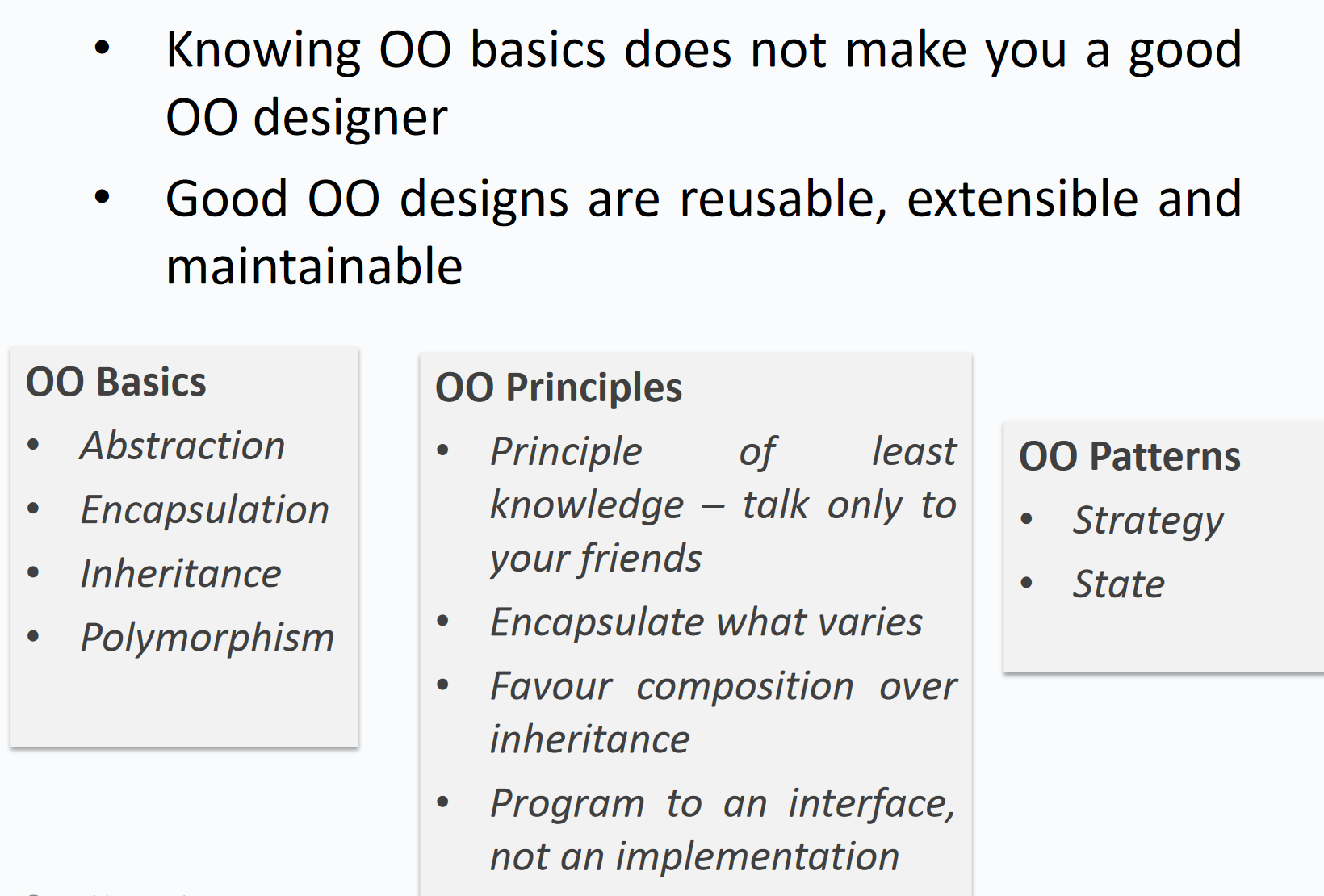


Figure 4 OOP Design Philosophy

State Pattern:

* Finite-State-Machine: Abstract machine that can be in exactly ONE state at a given time
  + Composed of: list of its states, conditions for transition and the initial state
* State: Description of the status of the system
* Transition: Set of actions when a condition is fulfilled / event is received
  + Identical stimuli trigger different actions depending on the current state
  + Entry and exit actions may be associated with transitioning to/from states
  + See State Transition Table (x: states, y:inputs 🡪 state change)

State-Machine uses:

* Frequently used in UI
  + Different aspect ratios
  + Event driven interface

Design Principles:

* Identify and encapsulate behaviours that have varying implementations
  + E.G. If flying can have different behaviours between different animals, we can encapsulate flying using a Flyable interface
  + Allows us to alter / extend the varying parts without affecting the parts that do not depend on that particular implementation detail
* Program to an interface, not an implementation
  + Program to a super type to allow for generalisation in the future
  + E.G. utilise dynamic binding
    - Dog d = new Dog(); d.bark(); 🡪 Change to 🡪 Animal d = new Dog(); d.makeSound();
  + Exploit polymorphism by programming to the super type
  + Behaviours 🡪 NO LONGER locked to a concrete implementation. Now we can use a set of “behaviour classes” that implement the interfaces. Classes are associated with behaviours and they do not need to know the implementation details

Diagram

Description automatically generated

Figure 5 Design Pattern: Behavioural Interfaces 🡪 Implemented by a set of implementing classes

A picture containing diagram

Description automatically generated

Figure 6 Integrating Instances with Behaviours

* Instance variable quackBehaviour and flyBehaviour are assigned based on what class of Duck
  + Implementing behaviours are assigned to instances
* HAS-A >>>> IS-A
  + Use **composition over inheritance** for behaviours

Strategy Pattern: Design pattern for the above duck example

* Motivation: You need to adapt the behaviour of an algorithm at runtime
* Intent:
  + Define a family of algorithms, encapsulate each one, and make them interchangeable
  + Strategy pattern is a behavioural design pattern that lets the algorithm vary independently from the context class using it
* Drawbacks: Increases the number of objects and relies on the client to be aware of the possible strategies
* Examples:
  + Sorting a list: Quicksort, bubblesort, mergesort, etc
  + Search: DFS, BFS, A\*, etc

Diagram

Description automatically generated

Figure 7 Implementation of a Strategy Pattern

Observer Pattern: Used to implement distributed event handling systems

* Use case: when you want to use event-driven programming.
* An object maintains a list of its dependents
  + Object 🡪 called a subject / observable / publisher
  + Dependents 🡪 called observers / subscribers
* Notifies observers automatically of any state changes in the subject

How can you maintain the observer pattern with dynamically allocated observers and subjects?

* Aims:
  + One-to-many dependency between objects without tight coupling
  + Automatically notify an unbounded number of observers when the subject changes state
  + Dynamically add and remove observers

Graphical user interface, text, application

Description automatically generated

Figure 8 Observer Pattern Possible Solution

* Subject:
  + Attach observer, detach observer, notify all observers
* Observer:
  + Update itself given a subject

Diagram

Description automatically generated

Figure 9 Observer Pattern UML Example

Model View Controller (MVC):

* Model never directly connects with the view
* Consider the botnet
  + Controller is the interface which gets requests and returns the results of any function calls using the JSON arguments
  + Models is the database and the code that works on the database
  + View is the templates and user-facing display of the returned data

**Diagram

Description automatically generated**

Figure 10 Model View Controller MVC

User Centred Design:

* Specify context of use
  + E.G. Amazon has a busy interface because they want to sell you lots of items, whilst Apple has a more minimalist design philosophy
  + Quality of experience is far more likely to be a product differentiator compared to product experience
* Specify requirements.
* Create design solutions
* Evaluate design

Composite Pattern: Aim is to manipulate a single instance of an object just as we would manipulate a group of them

* No discrimination between a single (leaf) object vs a composite (group) object
  + Composition of similar objects
  + E.G. Calculating size of file is the same as calculating the size of a directory
* Possible Solutions:
  + Uniformity: All child related operations are in the component interface
    - All the Leaf objects have the same interface as Composite objects, except the leaf objects don’t need to add children
    - Useful for dynamic structures where children types change dynamically (E.g. Leaf 🡪 Composite and vice versa)
  + Type Safety: Only define child related operations in the Composite class
    - Client treats Leaf and Composite objects differently
    - Useful for static structures where the child-related operations are not performed on unknown objects of type Component (E.G. through dynamic binding)

Diagram

Description automatically generated

Figure 11 Composite Pattern: Type safety vs uniformity

Design by Contract:

* Responsibilities are clearly assigned to different software elements, clearly documented and enforced during development through unit testing/language support
  + Clear functionality demarcation helps prevent redundant checks
  + Simplifies code and makes maintenance easier
  + Can crash if required conditions are not satisfied
* Why use this design paradigm?
  + This is an alternative to **defensive programming**. We don’t write checks for everything to save on performance
    - Defensive programming might have overheads that are unnecessary for systems that do not need 24/7 availability
    - No clear demarcation between responsibility between components that all check the input
    - Makes maintenance more complex, then every client in the pipeline will need to be modified when input is changed
    - Advantage: Guarantees reliability, safety, security, etc.
  + In return, our software may break out of our specified responsibilities
* When does this apply?
  + E.G: Student grading example. If a Grade class has to go down a marking pipeline with defensive programming methodologies
    - Every single client will perform checks on the data, which can lead to redundant checking
  + Want more efficiency compared to defensive programming, willing to take tradeoffs with security / validation

Design By Contract (DBC) Terminology: Software elements should define a contract / specification that governs its interactions with the rest of the software components

* Pre-condition: What is expected by the contract
  + Undefined behaviour if violated
  + Weaker in a subclass
* Post-condition: What does the contract guarantee
* Invariant: What does the contract maintain
  + Constraints which should always be true before and after the execution
  + Maintain invariants between the method calls (can violate between the pre and post-condition)
  + Property that doesn’t change from the class or any object derived from the class

What is a contract?

* Declarative and must NOT include implementation details
* Precise, formal, verifiable

Exception: Object that gets propagated up the runtime stack until a calling method catches the exception

* Kinds of exceptions:
  + Checked exception: Most common exception; Caught and handled by application
  + Error: Exceptional condition that are external to the system (usually cannot anticipate or recover from these)
  + Runtime exception: Internal application; Usually from bugs or logic errors
* For errors and runtime exceptions, we may not want to handle them and instead let the application break (for debugging purposes)

User defined exceptions: Extend Throwable

Testing Methodologies:

* Bombard with random data:
  + Random data is often considered unbiased

Random Numbers: Software produces pseudo-random numbers. Cannot be truly random

* Pseudo-random numbers are numbers that are predictable

Contravariance:

Covariance

Iterator Pattern: Accessing each element in an aggregation object sequentially, without exposing the underlying implementation detail

* E.G. Sequentially iterating through the elements of a list, without exposing how the List is implemented (ArrayList, LinkedList, etc.)
* Class implements Iterable<T>

Diagram

Description automatically generated

Figure 12 Iterator Design Pattern

Decorator Pattern: Attaches additional responsibility to an object dynamically at the runtime

* Decorator class has a component, extend the decorated component by transparently forwarding all requests to it
  + The additional responsibility is adding functionality before and after forwarding
  + Pick and choose the layered decorators to add the behaviour
    - Real world example: Layering input streams for filtering and transforming data. This is a data processing pipeline
* Original creation of component is unchanged
  + Decorator and Component are abstract classes or interfaces
  + Decorations cannot exist without the component they decorate
* Unwrap decorators by returning the contained component 🡪 This is logically the default behaviour without the decorator (decorators don’t change the component’s fields)

Diagram

Description automatically generated

Figure 13 Decorator Pattern

Text

Description automatically generated

Figure 14 Decorator Pattern: How components are wrapped by decorators

Diagram

Description automatically generated

Figure 15 Decorator Pattern example: Coffee shop

Adapter Pattern: Provides adapted interface to an interface that the client expects

* Adapter contains an instance of the class that it wraps
  + Adapter makes calls to the wrapped object (caller doesn’t reference wrapped object)
* Input 🡪 Map (Adapter Pattern) 🡪 Expected Output

Diagram

Description automatically generated

Figure 16 Adapter Pattern

Template Pattern: Define the overarching algorithm in the super class (generally an abstract class)

* Template method gives the algorithm (method calls each step (helper method) which needs to be executed)
  + Primitive operations: Operations with default implementations or must be implemented by subclasses
  + Concrete operations: Cannot be overridden
  + Hook: The helper methods that hook into the overarching algorithm to add functionality 🡪 In subclasses
    - Hook operations are generally left empty in the superclass.
    - Hook 🡪 The helper method that is being implemented in the subclasses. Entirely dependent on the functionality that the subclass wants to provide
    - Primitive 🡪 Functionality that is always required. General implementation can be in super class or specific implementation in subclass
* Areas with possibly varying behaviours within the algorithm are designated to helper functions
  + Subclasses override the helper functions to produce their custom behaviour
* Inversion of control: Subclass implements common (invariant) parts of the behaviour
  + This means that subclasses do not control the behaviour of the parent class
  + Parent class calls the operation of the subclass

Template vs Strategy pattern:

* Template: Inherit in class
  + Alterations from extending parts in subclasses
* Strategy: Add behaviour at runtime
  + Alterations by supplying different strategies at runtime
* When to use X?
  + Template pattern has invariant regions of code (the algorithm) in the abstract class with the dependent code being run in subclasses
  + Strategy pattern classes are independent to each other, with no defined algorithm which they add behaviour to

Factory method: Create objects by calling a factory method

* What does it solve?
  + How can an object be created so that subclasses can redefine which class to instantiate?
    - Subclasses can override the factory method to create a specific concrete implementing class
  + How can a class defer instantiation to subclasses
* Define a method for creating objects
  + Objects are now created by calling the factory method
* Constructors can only create one class
  + Factories are an abstraction for constructors so you can generate multiple types from one factory

Abstract Factory Pattern: Create family of objects using a factory class

* Extending class of abstract factory give you the specific objects within a family of object
* Client selects which factory type to use
* Use cases:
  + Different OS have different ways of making certain objects, family of distinct products for each OS

Diagram

Description automatically generated

Figure 17 Abstract Factory Structure

Builder Pattern: Extract object construction code out of its own class and move to a separate builder object

* Used substituting the constructor of complex objects which require lots of operations.
* Components of the builder pattern:
  + Director: Directs constructor algorithm. Provides order of execution.
  + Builder: Builds the object. Has methods which the director uses to build the object
* Why use the builder pattern?
  + Builder can be substituted with other builders yet share director functionality for code reusability



Figure 18 Design Pattern Overview

Diagram

Description automatically generated

Figure 19 Builder Pattern Structure

Singleton Pattern: Creational design pattern that lets your ensure that a class has only one instance, while providing a global access point to this instance

* Solution:
  + Make default constructor private 🡪 Prevents other objects from using the **new** operator with the Singleton Class
  + Static creation method that acts as a constructor. Following calls after initial construction returns the cached object
  + Static construction method always returns the same object
  + Should put **synchronized** access modifier on getInstance() method for singleton class
* Use cases: When you only want one instance of a class
  + Log files
  + Potentially database class?
  + Model class?

Graphical user interface, text, application, email

Description automatically generated

Figure 20 Singleton Pattern

Visitor Pattern: Add behaviour in separate visitor class instead of integrating into existing class

* Problem: It should be possible to define a new operation for (some) classes of an object structure without changing the classes.
  + Client’s software could be dependent on instantiation of class so it would be a hassle to modify
* Solution: Pass in instantiated object into one of the visitor’s methods, providing necessary data to for the added behaviour in the visitor
  + Visitor class defines a set of methods for each different type of class

Diagram

Description automatically generated

Figure 21 Visitor Pattern

* Element you are visiting must implement the Visitable/VisitorElement interface
  + Has accept(Visitor) method that calls the visitor v’s visit method
* Visitor:
  + Has visit method that adds behaviour
* Element / Visitable:
  + Has accept method that calls visit(element)
* Use case: Moving operations into visitor classes is beneficial when
  + Many unrelated on an object structure are required
  + Classes that make up the object structure are known to not change
  + New objects need to be added frequently
  + Algorithm involves several classes in the object structure 🡪 Manage all in one location
  + Works on several independent class hierarchies
* Limitation:
  + Extensions to class hierarchy require visit methods to be added