COMP2511 Revision - Summary O-O Design & Programming

O-O Design & Programming (University of New South Wales)

COMP2511 Revision

Lecture 1 and 2:

Inheritance: A form of software reusability in which new classes are created from existing classes by absorbing their attributes.

- Sub-class: More features and specialised
- Super-class: More general
- “Is-a” relationship
- Don’t use inheritance unless all or most inherited attributes and methods make sense.
- Java a class can only extend one class - single inheritance.
- Multiple inheritance, where a class can have two or more super classes
    - Problems arise when a superclass’s behaviour is inherited in two ways. (Diamond Inheritance)
    - In Java we can work around multiple inheritance through interface.

Method Forwarding: Methods from a “has-a” class to define some on the now methods.

Abstract Classes: Declare classes that define only part of an implementation, leaving extended classes to provide more specific implementations.

- Cannot be instantiated.
- If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass itself is abstract.

Interfaces: Are like (100% abstract classes).

- All methods are implicitly abstract
- Variables must be static and final, ie constants.
- Class can implement multiple interfaces.
- Class MUST provide implementations of interface methods.

Overriding: Replacing a super-class implementation of a method with a different one.
Rules for Method Overriding:

1. Argument list should be exactly the same as that of the overridden method.
2. The access level cannot be more restrictive.
3. A method declared ‘final’ can not be overridden.
4. Constructors cannot be overridden.
5. Return type of the overridden method should be the same or a sub-type of the return type in the super class.

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

Overloading: Defining methods with the same name and different arguments or return type.

Immutable: Once a class is created we cannot change it.

Access/Visibility Modifiers:

Public	Visible to the world
Private	Visible to the class only
Protected	Visible to the packages and all subclasses
No modifier (default)	Visible to the package

Lecture 3

Design Smells

Rigidity	● Tendency of the software being to change even in simple ways. ● Single change causes a cascade of change to other dependent modules.
Fragility	● Tendency of software to break in many places when a single change is mode.
Immobility	● Design hard to reuse ● Design has parts that could be useful to other systems, but effort need to disentangle is too high.
Viscosity	● Software: Changes are easier to implement through hacks rather then design preserving methods ● Environment: Development environment is slow and inefficient
Opacity	● Tendency of module to be difficult to understand
Needless Complexity	● Contains constructs that are not currently useful. ● Developers ahead of requirements.
Needless Repetition	● Design contains repeated structures that could be unified through abstraction.

Common Bad Code Smells:

- Duplicated code
- Long method
- Largeclass
- Long parameter list; data clumps

Good Design: Aims for a system with loose coupling and high cohesions among its components.

- Loosely coupled: Allow components to be used and modified independently
- High cohesions: Easier to maintain and less frequently changed and higher probability of reusage.

Improving Design:

1. Extract methods
2. Rename variables
3. Move method -> A method should be on the object whose data it uses
   a. Make code reusable through Encapsulation and Delegation (the act of one object forwarding an operation to another object).
4. Replace Temp with Query, remove local variables that are not needed.
5. Replacing conditional logic with Polymorphism

Refactoring: Process of restructuring (changing the internal structure of software) to make it easier to understand and cheaper to maintain, WITHOUT changing its external, observable behaviour.

- Improves design of software
- Reduces code smells
- Helps find bugs
- Helps improve efficiency
- Help conform to design principles.

Design Principles

Design Principles: A basic tool or technique that can be applied to designing or writing code to make software more maintainable, flexible and extensible.

- Help eliminate design smells
- Over-comformance leads to needless complexity

Principle of Least Knowledge (Law of Demeter): Classes should know about and interact with as few classes as possible. Helps design loosely coupled systems.

- **Rules:**
  - A method in an object should only invoke methods of:
    - The object itself
    - The object passed in as a parameter to the method
    - Objects instantiated within the method
    - Any component objects
    - NOT OBJECTS RETURNED BY A METHOD

LSP (Liskov Substitution Principle): Well-designed inheritance “Subtypes must be substitutable for their base types”.

- When subclasses don’t adhere to the super class, methods must be manually changed for each edge case. This is a breach of Open Close Principle.

Encapsulates what varies: Seperate the varying components of the code with the code that stays constant.
- Fewer unintended consequences from code changes and more flexibility.
- Makes it easier to alter the varying parts without affecting other parts.

Program to an interface, not to an implementation: Program to a super-type.
- DONT: Dog d = new Dog(); d.bark();
- DO: Animal dog = new Dog(); a.makeSounds();
- We want to exploit polymorphism by programming to a super-type so that actual runtime object is not locked.

FAVOUR COMPOSITION OVER INHERITANCE: Instead of inheriting behaviour, objects get behaviour by being composed with right behaviour objects and delegate to behaviour objects.
- Allows encapsulate a family of algorithms
- Enables the behaviour to change at run-time

Design Patterns

Design Patterns: A design pattern is a tried solution to a commonly recurring problem.
- Generally repeatable solution.
- Represents a template for how to solve a problem
- Captures design expertise and enables this knowledge to be transferred and reused
- Provide share vocabularies and terminologies
- IS NOT A FINISHED SOLUTION, just a general template
- Three Categories:
    - Behavioural: Pattern that identify common communication patterns among objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.
    - Structural: Patterns that ease the design by identifying a simple way to realize relationships among entities.
    - Creational: Patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Strategy Pattern

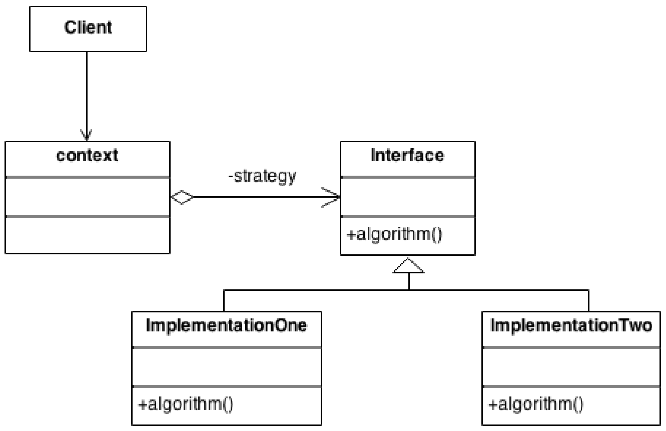
Strategy: Defines a set of algorithms that can be used interchangeably.

Intent:

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in the derived classes.

Problems it Solves:

- Prevents breach of Open-closed principle.
  - Client is only coupled to an abstraction, the realization doesn't matter.
  - Minimizes coupling.
- Prevents large methods involving many condition/switch statements.
- Scenario: Modes of transportation to a destination is an example of a Strategy.
  - There are many strategies: Driving, Flying, Boating….
  - In the end the we still get from point A to point B



Checklist:

1. Identify an algorithm (behaviour) that the client would prefer to access through a flexible point.
2. Specify the signature for that algorithm in an interface.
3. Bury the alternative implementation details in derived component classes.
4. Clients of algorithm couple themselves to the interface.

Examples:

- Sorting a list using different sorting algorithms (bubble, quick, etc)
- Search (Binary search, BFS, DFS, etc)

State Pattern

Intent:

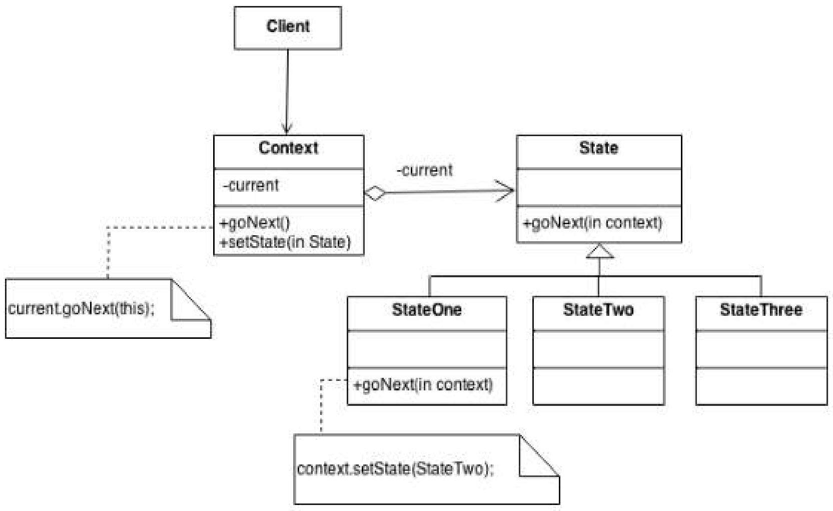
- Allow object to alter its behaviour when its internal state changes. The object will appear to change its class.
- An object orientated state machine
    - A machine that can be un one of a finite number of states at any given time.
    - Defined by:
        - A list of states
        - Conditions for each transition
        - Its initial state
    - Identical stimuli trigger different actions depending on the current state.

Problems it Solves:

- A monolithic object's behaviour is a function of its current state, and it must change its behaviour at run-time depending on that state.
- An application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

Checklist:

1. Identify an existing class, or create a new class, that will serve as the "state machine" from the client's perspective. That is the "wrapper class".
2. Create a State base class that replicates the methods of the state machine interface. Each method takes one additional parameter, an instance of the wrapper class (the context).
3. Create a State derived class for each domain state. These derived classes only override the methods they need to override.
4. The wrapper class maintains a "current" State object.
5. All clients request to the wrapper class are simply delegated to the current State object, and the wrappers object "this" is passed.
6. The state methods change the "current" state in the wrapper object as appropriate.



Composite Pattern

Intent:

- Compose object into tree structures to represent whole-part hierarchies.
  - Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- “Directories contain entries, each of which could be a directory.”
- Manipulate a single instance of the object just as we would manipulate a group of them.

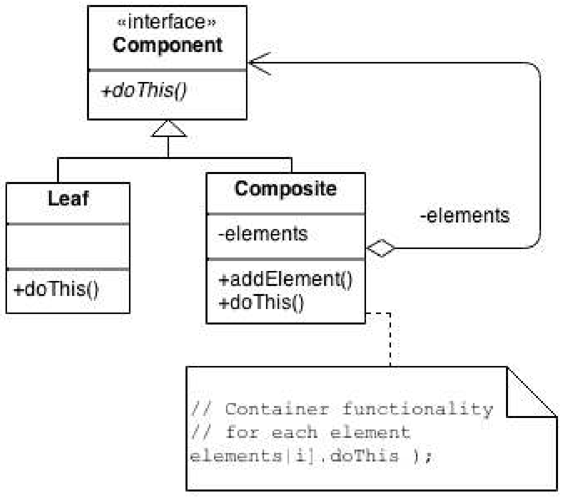
Problem it Solves:

- Application needs to manipulate a hierarchical collection of “primitive” and “composite” objects.
  - Processing of a primitive is handled different to a composite one.
  - Having to query each object before attempting is NOT desirable.

Uniformity	Type Safety
Include all child-related operations in the Component interface and vice versa.	Only define what is needed for leaf and component class.
Client can treat both leaf and composite function uniformly.	Client needs to treat leaf and composite objects differently
Useful for dynamic structures where children types change dynamically.	Useful for static structures, no chance leaf becomes composite and vice versa.

Checklist:

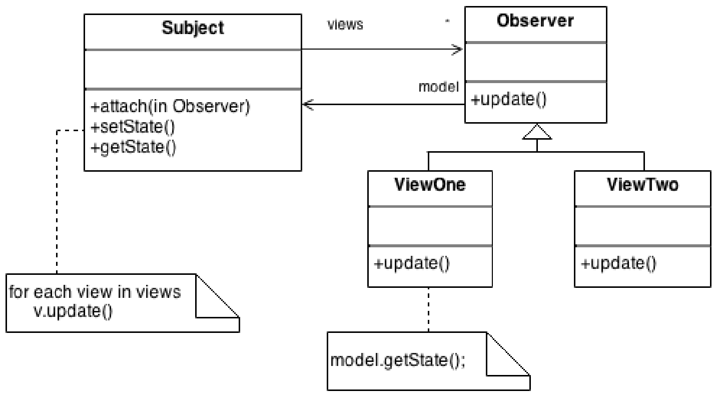
1. Ensure that your problem is about representing “whole-part” hierarchical relationships.
2. Consider the heuristics, divide domain concepts into leaf and composite classes.
3. Create the “lowest common denominator” component interface that both leaf and composite can implement.
4. Composite declared to be able to contain other components.



Observer Pattern

Intent:

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated immediately.
- Encapsulate the core, in a subject abstraction, and the variable components in an observer hierarchy.
  - Subject: Maintains a list of observers and to notify them of state changes by calling their update().
  - Observer is to register (and unregister) themselves on a subject and to update their state when they are notified.
- Aims to:
  - Allow communication without making objects tightly coupled.
  - Automatically update an open-ended number of observers, when subject state is changed.
  - Dynamically add and remove observers
- Two types:
  - Push: subject passes the changed data to its observers.
  - Pull: subject passes a reference of itself to observers, observers then get the data they need from it.



Problems it solves:

- When monitoring requirements must be scaled.
- Even handling system.

Checklist:

1. Differentiate between the subject (independent) and the observer (dependent) functionality.
2. Subject is only coupled to the Observer base class.
3. The client configures the number and type of Observers.
4. Observers register themselves with the subject.
5. The subject broadcasts events to all registered Observers.
6. The subject may “push” information at Observers or Observers may “pull” the information they need from the Subject.

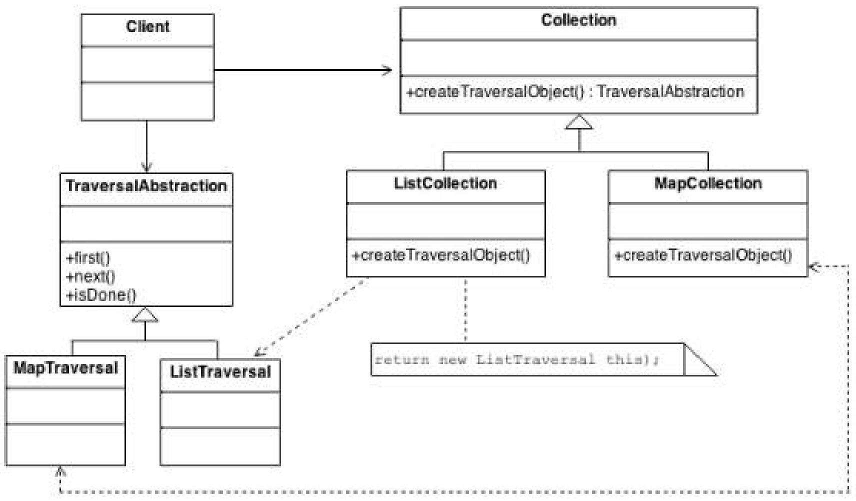
Iterator Pattern

Intent:

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying implementation.
- Polymorphic traversal

Problem it solves:

- Need to “abstract” the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.
    - Take responsibility for access and traversal out of the aggregate object and put it into an iterator object that defines a standard traversal protocol.



Checklist:

1. Add a create_iterator() method to the “collection” class, and grand the iterator class privileged access.
2. Design an “iterator” class that can encapsulate traversal of the “collection” class.
3. Client ask the collection object to create iterator object.
4. Client use the *first(), is_done(), next(), and current_item() protocol to access the elements of the collection class.

Decorator Pattern

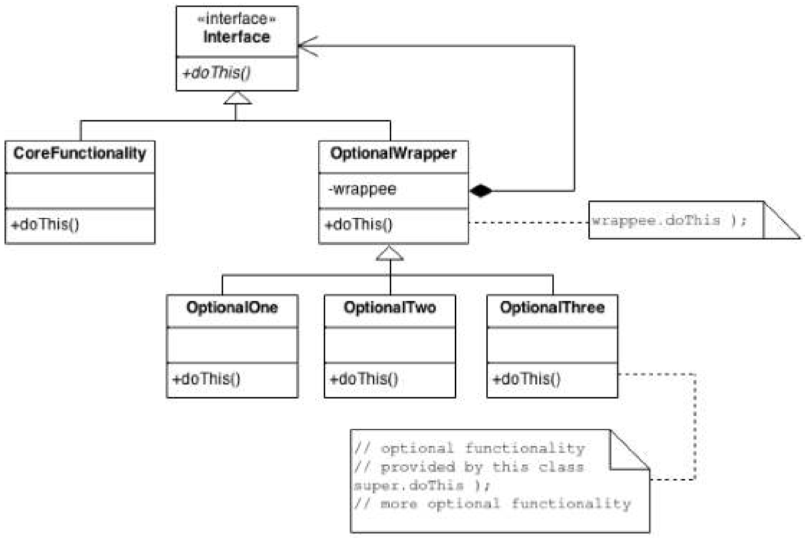
Intent:

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

Problem it Solves:

- You want to add behaviour or state to individual instances at runtime.

Checklist:



1. Ensure the context is a single core component, several embellishments or wrappers, and an interface that is common to all.
2. Create a “Lowest Common Denominator” interface that makes all classes interchangeable.
3. Create a second level base class (Decorator) to support the optional wrapper classes.
4. The Core class and Decorator class inherit from the LCD interface.
5. The Decorator class declares a composition relationship to the LCD interface, and this data member is initialized in its constructor.
6. The decorator class delegates to the LCD object.
7. Define a Decorator derived class for each optional embellishment.
8. Decorator derived classes implement their wrapper functionality and delegate to the Decorator base class.
9. The client configures the type and ordering the Core and Decorator objects.

Adapter Pattern

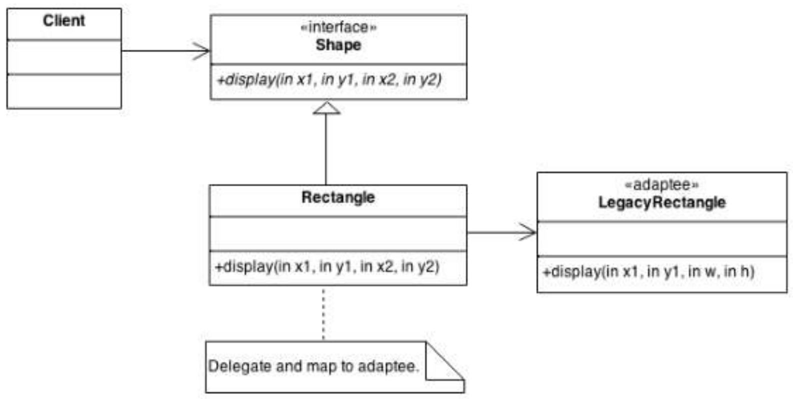
Intent:

- Convert the interface of a class into another interface clients expect.
    - Lets classes work together that couldn't otherwise due to incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system.
- THEY DO NOT OFFER ADDITIONAL FUNCTIONALITY
    - Purely mapping old to new.

Problem:

- An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not complete with the philosophy and architecture of the system being developed.

Checklist:


Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in w, in h)
Delegate and map to adaptee.

1. Identify the players: the components that want to be accommodated and the components that that needs to adapt.
2. Identify the interface that the client requires.
3. Design a "wrapper" class that can "impedance match" the adaptee of the client.
4. The adapter/wrapper class "has a" instance of adaptee class.
5. The adapter/wrapper class maps the client interface to the adaptee interface.
6. The client uses (is coupled to) the new interface.

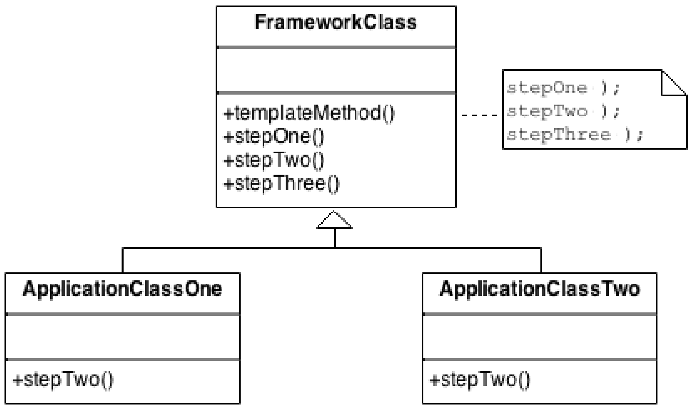
Template Pattern

Intent:

- Define the skeleton of an algorithm in an operation (using an abstract class), deferring some steps to client subclasses.
    - Lets subclasses redefine certain steps of an algorithm without changing the algorithms structure.
- Base class declares algorithm ‘placeholders’, and derived classes implement the placeholders

Problem that it solves:

- Two different components have significant similarities, but demonstrate no reuse of common interface or implementation.



Checklist:

1. Examine the algorithm, decide which steps are standard and which steps are peculiar to each of the current classes.
2. Define a new abstract base class to host the framework.
3. Move the shell of the algorithm (now called ‘template method’) and the definition of all standard steps to a new base class.
4. Define a placeholder method in the base class for each step the requires variation.
5. Invoke the placeholder method(s) from the template method.
6. Each of the existing classes declares an “is-a” relationship to the new abstract base class.
7. Remove from the existing classes all the implementation details that have been moved to the base class.
8. The only details that will remain in the existing class will be the implementation details specific to each derived class.

Visitor Pattern

Intent:

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Address new operations/behaviours to the existing objects, without modifying them.
- Seperation of an algorithm from an object structure on which it operates.

Checklist:

1. Confirm that the current hierarchy will be fairly stable and that the public interface of these classes is sufficient for the access the Visitor class will require.
2. Create a Visitor base class with a visit(ElemntXxx) method for each Element derived type.
3. Add an accept(Visitor) method to the Element hierarchy. The implementation in each Element derived class is always the same-
    a. accept (Visitor v) {v.visit(this) ;}
4. Element hierarchy is coupled only to the Visitor base class, but the Visitor hierarchy is couple to each Element derived class. If the stability of the Element hierarchy is low and the stability of the Visitor hierarchy is high, consider swapping the hierarchies.
5. Create a Visitor derived class for each “operation” to be performed on Element objects. visit () implementations will rely on the Element’s public interface.
6. The client creates visitor objects and passes each to Element objects by calling accept().

Factory Method

Intent:

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
    - Defers instantiation to subclasses.

Problem:

- Creating an object directly within the class that requires (uses) the object is inflexible.
- It commits the class to a particular object.
- Makes it impossible to change instantiation without having to change the classes.

Abstract Factory Method

Intent:

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem:

- Application is to be portable, it needs to encapsulate platform dependencies.

Builder

Intent:

- Seperate construction of a complex object from its representation so that the same construction process can create different representations.

Problem:

- An application needs to create the elements of a complex aggregate

Singleton

Intent:

- Ensure a class has only one instance, and provide global point of access to it.
- Encapsulated “just in time” or “initialization one first use”
- All singleton has features in common:
    - Make default constructor private.
    - Static creation method.
    - A way to access this singl, ie. getInstanceOf()

Design By Contract:

Design by Contract (DBC): At design time, responsibilities are clearly assigned to different software elements, clearly documented and enforced during the development.

- Aborts/Crashes if required conditions are not met.
- Software element should define a contract that governs its interaction with the rest of the software components.
    - Pre-condition: What does the contract expect?
        - If true, it can avoid handling cases outside of precondition.
        - Must be true prior to executing section of code
    - Post-condition: What does the contract guarantee?
        - Return value(s) is guaranteed, provided precondition is true.
    - Invariant: What does the contract maintain?
        - Some values must satisfy constraints, before and after the execution.
- Benefits:
    - Clear demarcation, helps prevent redundant checks -> simpler code
    - Do not need to do error checks for conditions that do not meet preconditions.
    - Given preconditions client can expect the specified post-condition.
- Preconditions in Inheritance:
    - A redefinition in a subclass may lessen the precondition but not increase it.
    - A redefinition in a subclass may be more specific however the post-condition must benefit the client.
    - Invariants must stay the same in subclasses

Exceptions

- Use asserts to valid data first
- If an exception can be handled meaningfully in a method, the method should catch the exception rather then declare it.
- If a subclass method overrides a superclass, it most not throw more exceptions then the original superclass.
- Programmers should handle checked exceptions
    - Checked errors are out of the control of the program
        - IOException, SQLException
- If unchecked exceptions are expected, you must handle them gracefully.
    - Unchecked errors are errors at runtime (programming errors)
        - VirtualMachineError, OutofMemoryError

Generics in Java

Generic: Enable types (classes and interfaces) to be parameters when defining:

- Classes
- Interfaces
- Methods

Benefits:

- Removes casting and offers stronger type checks at compile time
- Allow implementation of generic algorithms, that work on collections of different types, can be customized, and are type safe.
- Adds stability to your code.

Generic Types: A generic class or interface that is parameterized over types.

- Example E and T

Wildcards: Represent and unknown class.

- Upper boundary: <? sxtends upperType>
- Lower boundary: <? super lowerType>

Collections in Java

Collections framework: A unified architecture for representing and manipulating collections.

- Contains:
    - Interfaces: Allows collections to be manipulated independently of the details of their representation.
    - Implementations: Concrete implementation of the interfaces
    - Algorithms: THe methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
        - Said to be polymorphic.

User Centered Design

1. Visibility of System Status
2. Match between system and real world
3. User control and freedom
4. Consistency and standards
5. Error prevention
6. Recognition rather than recall
7. Flexibility and efficiency of use
8. Aesthetic and minimalist design
9. Help user recognize, diagnose, recover from errors
10. Help and documentation.

Code Smells

Generally:

- **Bloaters:** Code, Methods and classes that have grown in size, that are hard to work with.
- **OO Abusers:** Result from incorrect or incomplete application off OO principles.
- **Change Preventers:** Code changes are difficult.
- **Dispensables:** Code that is pointless and unnecessary.
- **Couplers:** Excessive coupling between classes

Fixes to:

- Long Method:
  - Extract Method
  - Replace Temp with Query
  - Introduce Parameter Object
  - Replace method with method object.
- Large Class:
  - Bundle group variables via Extract Class or Extract Sub-Class
- Long Parameter List:
  - Placing query call inside the method body via replace.
- Data Clumps:
  - Move behaviour to the data class via Move Method
  - If its repeating in a field/attributes consider extract class.
  - For parameters Introduce Parameter Object
  - For methods consider passing the entire object (Preserve Whole Object)
- Refused Bequest:
  - Replace inheritance with Delegation
  - Inheritance push down method
    - Extract Sub-class class
- Duplicate Code:

    - Extract Method
    - If duplicate is similar but not identical: Form Template Method
    - If two methods do the same thing but different algorithm: Substitute Algorithm
    - Extract Super Class
- Feature Envy:
    - Move method to relevant class
    - Extract Method
- Divergent Change:
    - Encapsulates what changes and then Extract that as a Class
- Shotgun Surgery:
    - Move method or Move Field to put all changes into a single Class
- Data Classes:
    - Move method to the class that the data is actually being held
- Lazy Classes:
    - Move data from lazy class to the "upper" class.
- Switch Statements:
    - Replace switch with a Polymorphic Solution