Software Engineering II

Jordan Pyott

2021-05-20

Contents

Course Information

The Software Engineering (Part II) builds on the material introduced in SENG201 (Introduction to Software Engineering) and is intended as a companion course to SENG302 (Software Engineering Group Project).

In this paper, you will learn about software processes, including Agile Software Development (e.g., Scrum, Kanban), effective source code and documentation management, resilience engineering, acceptance testing, software metrics and software design principles.

At the end of each lecture it is expected to finish the prep slides before the next lecture

Lecture Material

Resources

Assignment Submissions

Course Outline

Lectures outline

Term 1 - Agile Software Development

- 23/02 Recap on software engineering methods
- 26/02 Scrum 101
- 02/03 (Agile) Requirement analysis
- 05/03 Scrum team management
- 09/03 Continuous integration
- 12/03 Continuous delivery
- 16/03 Testing and mocking
- 19/03 Resilience & reliability
- 23/03 Software quality metrics
- 26/03 Software architecture 101
- 30/03 User Experience 101

Term 2 - Object-oriented design principles

- 27-30/04 Generics & Collections
- 04-07/05 Object-oriented design
- 11-14/05 Design pattern I
- 18-21/05 Design pattern II
- 25-28/05 Design by contract

- 01-04/06 TBD

Lab/tutorial outline

Term 1

- 01-03/03 Tutorial - Scrum (1)
- 08-10/03 Tutorial - Scrum (2) [Assignment 1]
- 15-17/03 Conceptual modelling and `JPA`
- 22-24/03 Acceptance testing with Cucumber
- 29-31/03 `Facking` and mocking with `Mockito`

Term 2 (Provisional, please check forum for latest updates)

- 03-05/05 Generics & Collections
- 10-12/05 Design
- 17-19/05 Design Patterns I
- 24-26/05 Design Patterns II
- 31-02/06 Design by Contract

Assignments deadlines

- 08-10/03 Assignment 1 - Scrum tutorial 2 (during lab time), self-enrol via group forming page
- 01/04 Assignment 2 - Reflection report
- 04/06 Assignment 3 - Acceptance testing and design patterns

Grades

- Assignment 1: Scrum tutorial 15%
- Assignment 2: Reflection report 5%
- Assignment 3: Coding assignment 20%
- Final Exam: 60%

Lectures

Lecture One: Course kickoff and recap of software development methods

SCRUM

Scrum is a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest value. Scrum is a lightweight framework that helps people and teams to generate value through adaptive solutions for complex problems, scrum was created by `Ken Schwaber` and `Jeff Sutherland`.

Scrum is a method of generating agile development within a workplace/team, it does this by using adaptive planning and works to achieve early delivery of the product in order to assess and improve the work flow within a development team.

Here are the lead roles in a `SCRUM` environment workplace.

1. A Product Owner orders the work for a complex problem into a Product Backlog.
2. The Scrum Team turns a selection of the work into an Increment of value during a Sprint.
3. The Scrum Team and its stakeholders inspect the results and adjust for the next Sprint.
4. Repeat

Here is an Overview of Scrum from Scrum.org.

Lecture Two: Scrum software development framework

The Five Values Of Scrum

- Openness
- Focus
- Respect
- Courage
- Commitment

Initial startup for scrum

Some bootstrap effort is needed before starting

- start from a vision of the product
- refine its objectives
- discuss with the stakeholders, including the Product Owner, create an initial backlog with user stories

- agree on working mode and standards

`Scrum` and `Kanban` are quite different here

- `scrum` larger initial phase is needed before planning the first sprint
- `kanban` task-oriented, so faster start-up

Part I - what are we going to do?

- the PO presents highest priority stories or epics
- estimate the complexity of each story
- use previous velocity (i.e. number of points delivered previously)
- first draft of sprint backlog
- a sprint should be as coherent as possible

Part II - how are we going to do it?

- selected stories are broken down into tasks by the team
- tasks are described thouroughly (i.e. anyone can pick it up)
- estimations of tasks are set collaboratively, i.e. reaching a consensus
- the team knows the steps to implement each story
- the team knows what each task needs to produce

Lecture Three: Decipher users, their expectations and plan the work

Scrum Roles

Product Owner

- translates user demands into stories
- maintains and prioritises the list of things to do
- negotiates content of releases and timing with the team

Scrum master

- acts as a coach
- facilitates communication inside and outside the team
- represents managements, but protects team members

Team aka developers

- everyone is a developer
- everyone works on everything

Epic:

- large piece of work that may be implemented over many scrums

Story:

- a well defined piece of work
- confined within a sprint (unless too much is allocated)

Product backlog:

- contains tasks (stories)
- everything that still needs to be done
- maintained by PO
- some estimations may be missing

Sprint backlog:

- items that will be handled within our current sprint
- allocated from product backlog and estimated

Lecture Four: Leadership, team work and accountability

Dealing with Unplanned

- Distinguish between development issues and bugs
  - issue: a problem identified at the latest before a review
  - bug: a problem discovered at the earliest during later regression tests

Dealing with impediments

- Long meetings: stick to essential stakeholders and keep them time-boxed
- Illness: unavoidable, but refrain from ignoring it
- broken builds: implicit top priority for the whole team to fix it
- tools: always frustrating, but you need to fix it first
- third parties: even more frustrating, consider alternatives
- scope creep: pay special attention to review the stories and broken down tasks
- unreliable PO: learn how to deal with them
- team problems: retrospectives and seek for assistance
- external incentives: remember Scrum is about teamwork

Lecture Six: Software architecture

The reason to model is in order to raise the abstraction level, this is what we are trying to achieve in software engineering, because it is easier to work on abstractions than to work and the ground level.

Representation are good (visual diagram), but you need the following:

- to understand the notation
- to understand the context and vocabulary
- the purpose of the model must be distinct and clear
- should be unambiguous

In computer science we use class diagrams and domain models (used in Lecture 3)

- Every organisation has their own rules
- They manipulate their own vocabulary and concepts
- concepts have relations to each others

Domain concepts *domain driven design*

- will be the one you logically manipulate
- will be the one you store in the database
- should be responsible for their states and logic *i.e. encapsulation*

These diagrams is to understand the concepts, not to have an implementation of the logic

Domain model example for a warrant of fitness system:

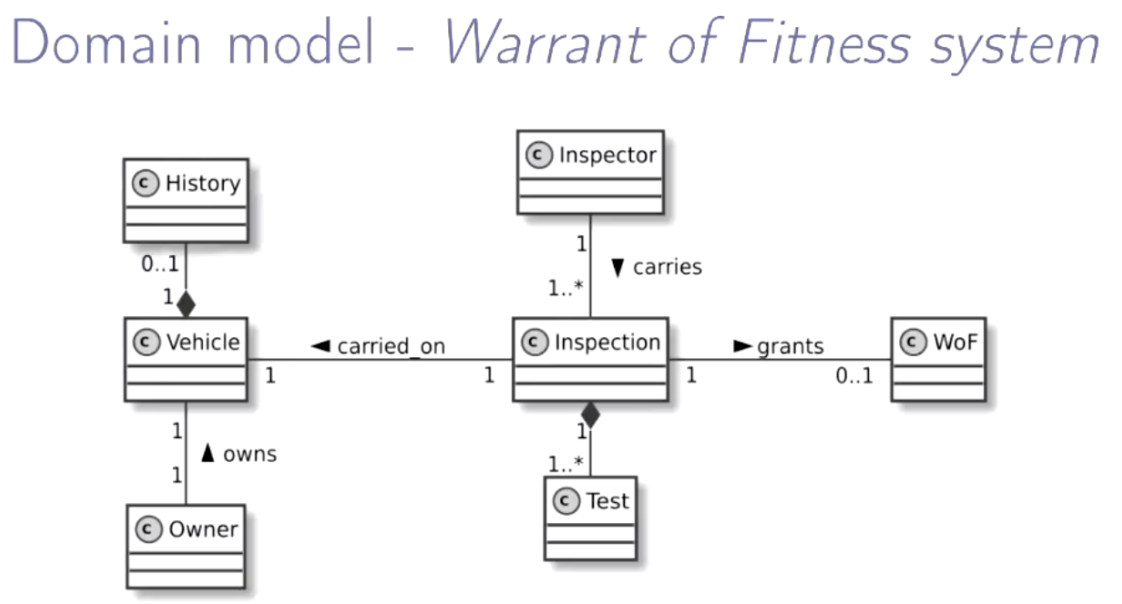


Figure 1: Domain model

Architecture tactics

- Relating to one attribute or decision
- tackle one concern at a time

Architecture styles and design patterns

- offer re usability
- must be taken into account in the entire project
- may encompass multiple tactics

Both are shaping the design of your system early

> NOTE: Architecture drift occurs quicker and more often than you might expect
> Re-engineering may be impractical or painful to implement as a result of this

README files

- Explains the context and objectives
- Authors, contributors, versioning and other pointers
- specify deployment procedure, testing and dependencies
- describe content and refer to licensing

Wiki pages

- put your external analysis, wire frames, architecture, decisions
- manual tests
- keep them organised in categories
- keep it updated
- terms and conditions

Lecture Seven: Staged and automated testing

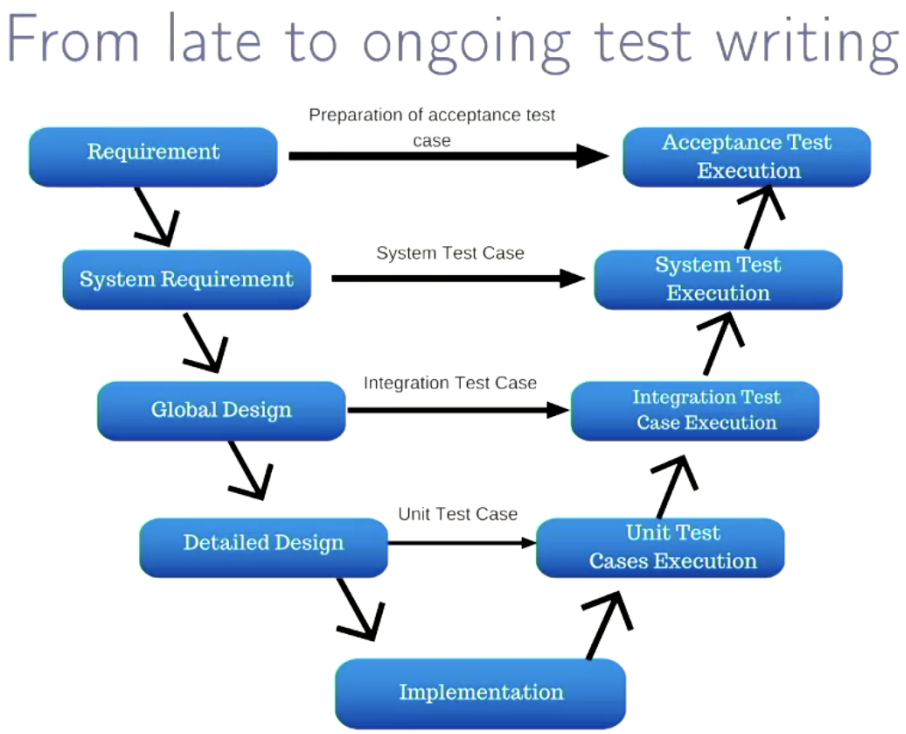


Figure 2: Testing model

Objectives of testing:

- Validation: demonstrate the software fulfils its requirements
- Verification: identify erroneous behaviour of the system

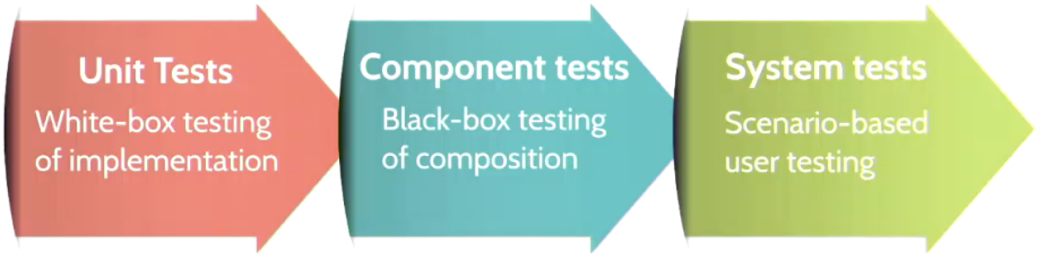


Figure 3: Stages of testing

Unit testing

Any piece of code should be tested

- every feature should be testable
- should fake or simulate human input and should be self sufficient
- *is hard because of the nature of code in a code-base*
- code assertion are primordial for regression tests (making sure old tests are still working as expected)
- explicit verification of pre/post conditions, (checking ranges of inputs)

Component testing

Testing of identifiable and isolated parts of a system

- hidden and interchangeable implementation

On top of being skeptical on input data

- Make components fail, check for differing failures
- Stress testing or message overflow
- If a call order exists, try to call operations in the other order

System testing

Finally, the last stage of testing

- Integrate third-party components or systems
- Should be preformed by *dedicated* testers and surely not only by the developers

Scenario-based testing

- main usages, full interaction flows and should be modelled
- Start from graphical user interactions and go through to opposite end

Trace and record tests executions (wiki or spreadsheet)

- Input values, expected output and observed output
- meta data like who and when issues occur, and a tracking issue associated with this issues id

Figure 4: Agile staged testing

> Make sure that tests handle Null values, this is an extremely common mistake
> cover invariant properties are valid, check incoming parameter's, assert are useful for regression testing

Acceptance testing

User stories are always accompanied by acceptance criteria

- do you remember about INVEST?
- A story is a promise for a conversation, with examples of usage
- define application interfaces *isolating UI*
- use dependency injection, *inversion of control*
  - *Dependency injection: a technique in which an object receives another object that it depends on*
- Any `async` functions must be synchronous in order to test

Automating acceptance testing:

- use playback tools sometimes, *selenium, Serenity*
- testing directly on a GUI may be time consuming

As sprint reviews must be planned and prepared

- ensure all acceptance criteria are running as expected
- may need to rework some stories, avoid trying to fix nasty bugs last minute

Lecture Eight: Ethics and resilience engineering

> NOTE: I am not sure how much of this will be assessed?

Because we are using artificial intelligence for almost everything now from employment, academic integrity and almost anything else. Because of this as software developers, it is important to contain our personal bias into account when developing software.

A problem with almost all codes of ethics is that they try to put people into boxes and categories, this is not always possible due to the individuality experienced by a set of users.

The ethics behind things at the moment have blurred lines, e.g. for self driving cars, is it ethical to prioritise life inside the car rather than prioritise life outside the car.

The environmental impact of software development and maintaining software, block-chaining for example uses large amounts of power as we are constantly generating new hashes (computationally difficult to do). Note, their is a conversation that needs to be had about weighing up the pros and cons of this development.

Don't catch unintentional errors in your code, due to the fact that this will allow you to miss unintentional errors being parsed.

Capacity management

Programmers waste enormous amounts of time thinking about, or worrying about the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

- Design code in a way that it can be traced (using loggers)
- use threads carefully, *starvation and deadlocks are very bad*
- write dedicated tests and monitor logs
- Protection systems:
  - system specialised in monitoring the execution of another, trigger alarms or invoke corrective programs
- Multi version programming
  - concurrent computation
  - can be hardware with different items or providers
  - can be software with different development teams

  - mismatch detection with voting system
  - triple redundancy:
    * usually only one thing breaks at a time, if two cases are working correctly and a third is not, the problem is likely with the third

Visibility

- Programming contracts often define a *need-to-know* principle

Validity

- check formatting and domain of input values, including boundaries
- explicitly or regression testing

Exceptions

- NEVER EVER display the stack trace to the user (500 errors should be handled)
  - this is the easiest way to reverse engineer and hack into the server side system

Both client side checking and server side checking are necessary, you cannot handle a request without both.

The four R's of resilience engineering plan:

- Recognition: how an attack may target an identified resource
- Resistance: possible strategies to resist to each thread
- Recovery: plan data, software and hardware recovery procedures
- Reinstatement: define the process to bring the system back

This means we should have **backup** and **reinstalling** procedures, that should be specified on top of deployment.

Lecture Nine: Continuous Integration

The integration problem:

This is the process of combining `units` into a product

- After you developed your part
- … and tested on our side

Phased Integration:

Multiple variables, interfaces and technologies

- Global variables
- Different error-handling assumption
- Weak encapsulation

These are where and how we define environment variables for testing, *DO NOT ADD THESE TO THE REPO*

Continuous Integration:

Working software is the primary measure of success, If we cannot run and see it, it is **not** progress.

> This is when we have progress on our personal machine but it is not uploaded to any current repository

There are eXtreme programing principles that were outlined by Martain Fowler known as Fowler's Principles.

These are outlined here:

- maintain a single source repository
- automate the build
- make your build self-testing
- everyone commits to the `master/main` branch every day
- every commit should build the branch on an integration machine
- fix broken builds immediately
- keep the build fast
- test in a clone of the production environment
- make it easy for anyone to get the latest executable
- everyone can see what's happening
- automate the deployment phase

Figure 5: Fowlers Principlese

Managing repositories if it is handled in a Single source repository, this makes it easier to maintain, helps to be `centralized` or use *subversion* and `distributed` *git* having two levels of commits. To be distributed means that each person working on the task is allowed to maintain their own branch `feature branches`, to be centralized is to have a `main` branch *maybe multiple main branches such as [dev, master, release]*. This helps with implementation, working in teams and streamlining development. *Once again Linus Torvalds has made our life easy*.

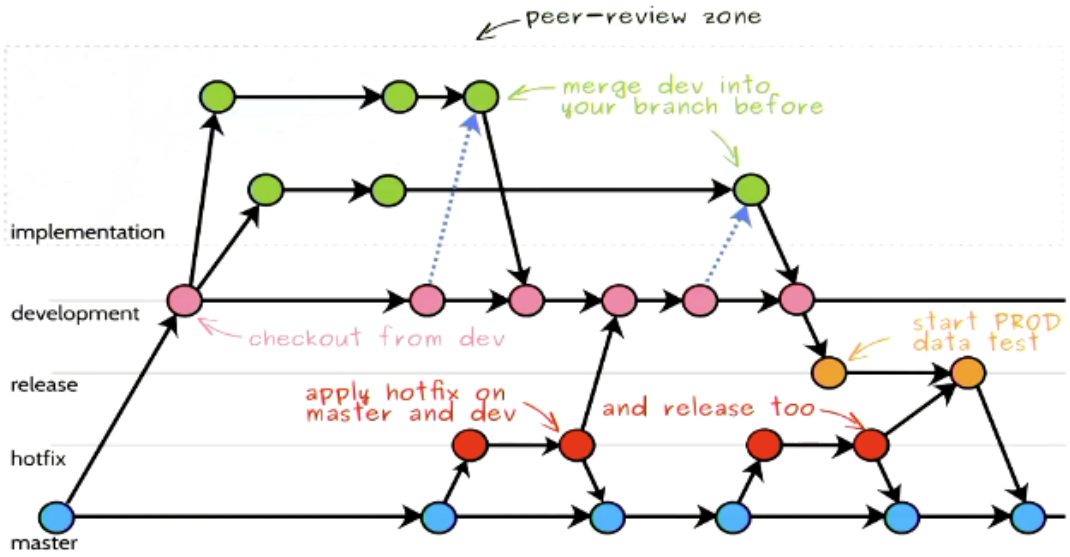


Figure 6: Driessen's Branching and Merging

Figure 6 shows a development strategy that we will be using within our SENG302 project, this is a good idea as it includes a release branch (only push to this at the end of each sprint), a dev branch (usually only modified after a story is fully complete), story branches (only merged into dev when finished), master branch (updated from dev when dev is pushable).

Lecture Ten: Usability, but more than that...

User experience encompasses many aspects

- Usability: meet users requirements
- Efficiency: no fuss, no bother
- Satisfactory: with simplicity and elegance

As a UX designer, you need empathy

- putting yourself in your users' shoes
- know about their prior knowledge and target knowledge
- understand their frustrations and desires

Basic tools for UI designers

- Inputs
- Forms
- Modals

- Infographics
- Pictures
- Whitespace

Use Wire frames, mock-ups and prototypes in order to design web applications and websites. This allows for a point of comparison and stops you from loosing track of the original design.

Lecture Eleven: Design Principles - One

- Cohesion
  - Put things in the right place
  - Keep connections local, don't have to keep on getting from multiple locations
  - Reduces complexity
- Coupling
  - Value independence (low, loose, …)
  - Value simplicity (simple interfaces)

> Principle: cohesion is good, coupling is bad

The one problem that makes computer science and software design is simply this: **complexity** and the only real solution we have to this problem is **decomposition**

Complexity	Decomposition
Size	Break up
number of parts	hide parts
connectedness	decouple
change	abstract

Methodologies to solve this problem

1. Top-down approach
   - Functional decomposition
   - Stepwise refinement
   - Transaction analysis
2. Bottom-up

    - Combination of lower-level components

3. Nucleus-centered
    - Information hiding (Parnas)
    - Decide about critical core (algorithms) then build the interface around it
        - This is object orientation (the currently recommended approach)

4. Aspect-orientated
    - Separation of concerns

Nucleus centred design

Information hiding (David Parnas)

- Identify design decisions with competing solutions (a choice of sorting algorithm)
- Isolate details behind interface
- Motivation for object orientation

The idea is if we build our idea of implementation, we have many things in common with all displays such as classes, polymorphism and interfaces.

Decomposing complexity

- Given a student class, where should we put the `display_student()` function?
    - Should it be contained within the class or not?
    - The idea is that if we are using `getter()` then we are putting information within the wrong class
        * This is an example of coupling
        * These should only be used if we need information in multiple places

Hiding

> NOTE: There is a distinction between encapsulation and hiding information

- Encapsulation: means drawing a boundary around something. It means being able to talk inside and outside of it
- Information hiding: is the idea that a design decision should be hidden from the rest of the system to prevent unintended coupling

Encapsulation is a programming language feature, information hiding is a design principle. Information hiding should inform the way you encapsulate things, but of course it doesn't have to, they are NOT the same thing.

Encapsulation leak

```java
public class Student {
    private Set<Enrolement> enrolements;

    public Set<Enrolement> getEnrolements() {
        return enrolements;
    }
}
```

This is considered a bad idea, the better approach is to use the following

```java
public class Student {
    private Set<Enrolement> enrolements;

    public Set<Enrolement> getEnrolements() {
        // returns a copy of the collection, not the actual class
           itself
        return Collections.unmodifiableSet(enrolements);
    }
}
```

This is a better approach.

Coping with change

- Finding the solid bits.
  - Make them the framework of your program
  - Principle: encapsulate that which varies (expected to change)
- Find the wobbly bits.
  - Hide them away

Hide design decisions

- If you chose it, you should hide it
  - Data representations
  - Algorithms
  - IO formats
  - Mechanisms
  - Lower-level interfaces

This is not always the case, we are hard wiring information, this is where we can use DAO's (Data access objects) in order to access objects. It is better to build an interface in order to hide information.

Decoupling

Principle: we should use interfaces in order to hide information, the abstractions are going to be stable, this is because they are easier to maintain and easier to understand

Lecture Twelve: Design Principles - Two

Principle: make your system open for extension, but closed for modification

We want to design our interfaces to stay the same, even if we change the information behind it.

The Open-closed principle

Software entities should be open for extension but closed for modification.

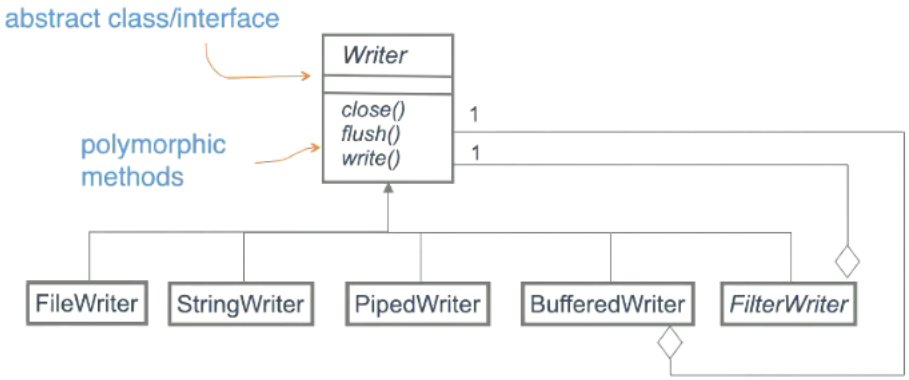


Figure 7: open closed principle

Tell, don't ask

- Decisions based entirely upon the state of an object should be made within the object itself
- Objects should tell each other what to do (via methods)
- This avoids asking for information from an object and then using it to make decisions about that object
- Encourages *keeping related data and behaviour together*

We should know the state of the object without having to ask the class for its information

Keep the design simple

- Aim for simplicity
- KISS (google more information about these terms)
- YAGNI (google more information about these terms)

If we are doing constant refactoring, this is a sign that you are over complicating tasks, this defines our next principle

Principle: we shouldn't have to over-engineer our solutions

The overarching principle

Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying series of small behaviour-preserving transformations, each of which "to small to be worth doing". However the cumulative effect of each of these transformations is quite significant. By doing them in small steps you reduce the risk of introducing errors. You also avoid having the system broken while you are carrying out the restructuring-which allows you to gradually redactor a system over an extended period of time.

The terms that have been outlined in the last few lectures now gives us some more valuable language in order to describe ideas in object orientated design. Below is an explanation of how these ideas are mapped to a class diagram.

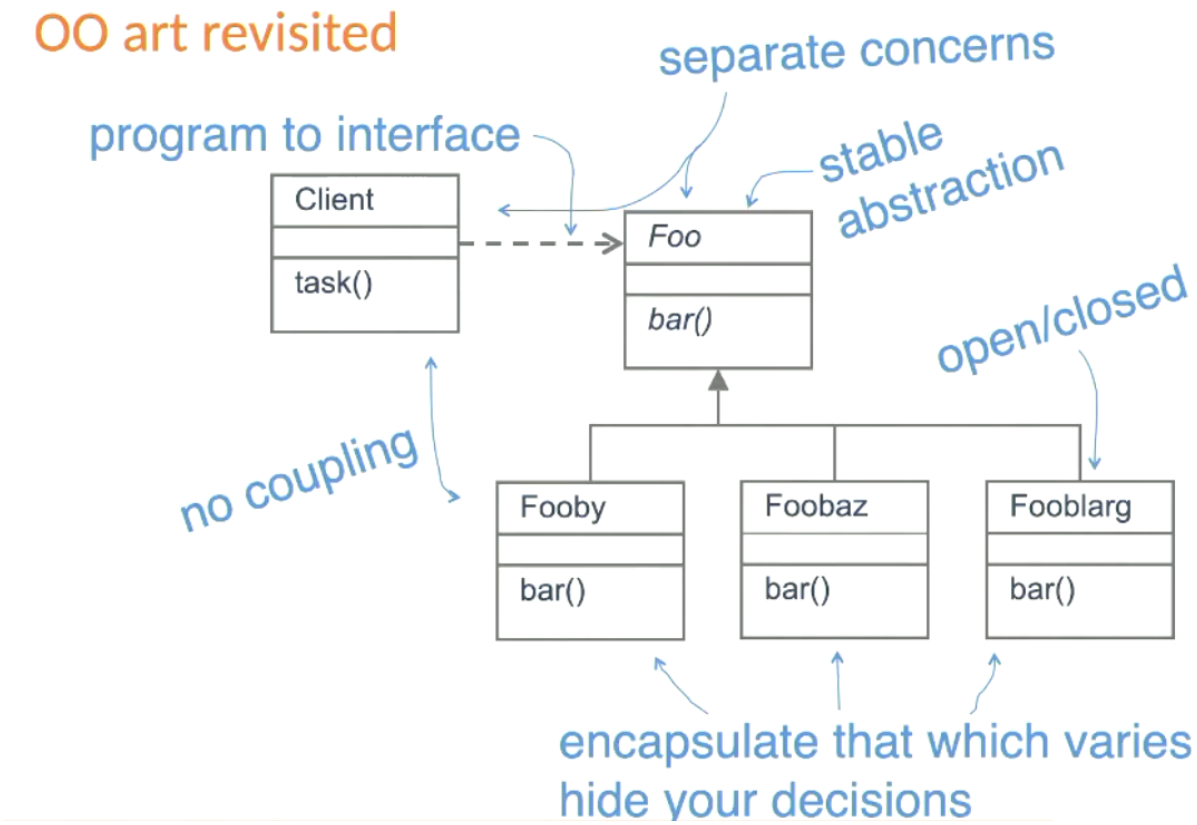


Figure 8: OOP

The yin and yang of OOP design

Data or behaviour modelling?

- OO is data modelling
  - Focus on data internal to object
- OO is behaviour modelling
  - Focus on services to external world

> NOTE: UML is now the standard for modelling this information

Inheritance: The Dark Side

Inheritance Mistakes

- Inheritance for implementation
- “Is a-role-of”
- Becomes
- Over-specialisation
- Violating the LSP

If a stack is inherited from a vector, it will have a bunch of properties that a stack should not contain (such as adding to the middle of the vector, this is because a property has been inherited from the vector class and that voids the purpose of the stack).

Principle: favour composition over inheritance

Is a-role-of

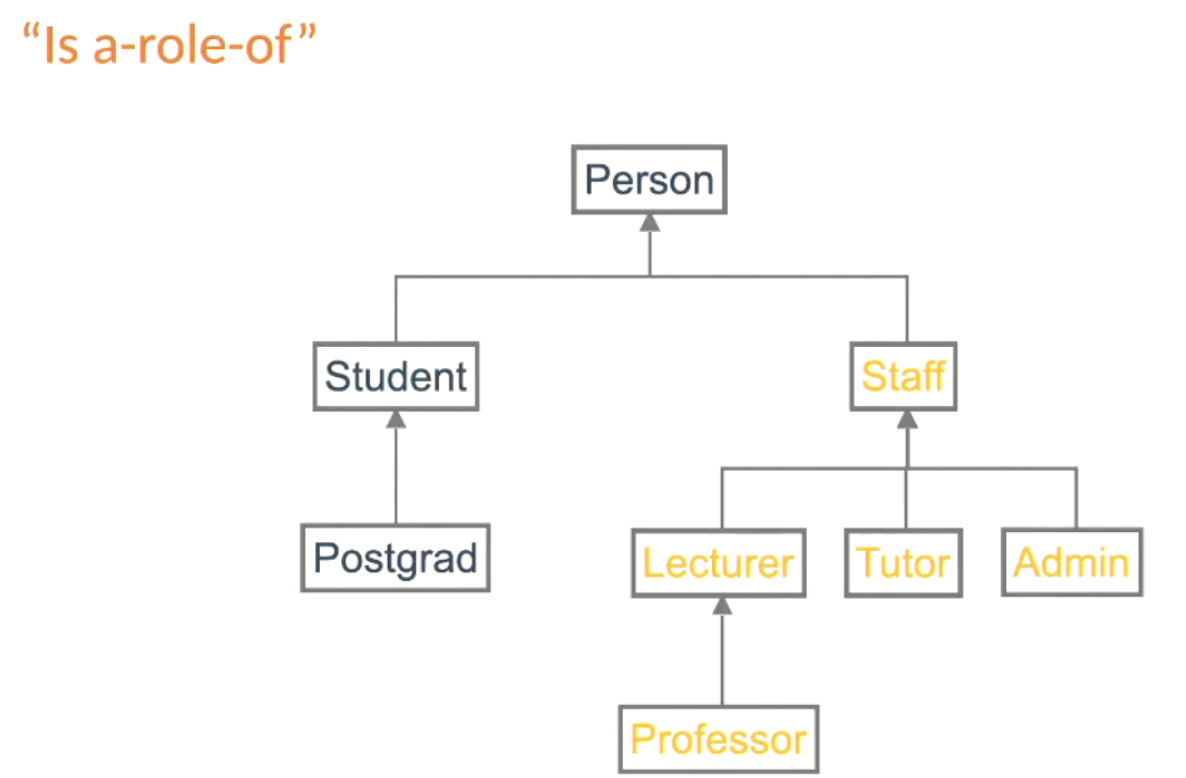


Figure 9: is a role of

What happens if a post-grad is acting as a lecturer?

How do we separate concerns?

This is a case of favouring inheritance over composition having lower structure inheritance would solve this.

Principle: Inheritance is NOT dynamic, it should not be changing!

Over-specialisation

Programming to the interface (not to the direct object) will prevent us from going to the lowest level of a hierarchy, this will allow us to not over-specialize to a specific object that does not contain the correct parameters we are expecting.

Liskov Substitution Principle

If we have an object of type T and we are given an object of type S, and they both extend an object O, then they should act similarly and should be mostly swappable *this does not hold for abstract classes*.

Following this structure may have overwriting parent methods for some classes and not others, this is not great as it violates this principle, there is *always* a better way to handle this, the substatution principle boils down to this **when we expect a parent class, a child class should always act the same way**.

Lecture Thirteen-1: Design Principles - Three

Single responsibility

- Every module or class should have responsibility over one part of the functionality only. That responsibility should be fully encapsulated
  - *this may be better to swap to two smaller classes*
- Responsibility = reason to change
- One class/module knows too much
- Typically found in controllers, initialization, etc.

Interface segregation Principle

- No client should be forced to depend on methods and interfaces that it doesn't use
- Risk of "white lies" in code
- Split interfaces into cohesive, smaller and more specific ones.
- Reduces to cleaner and smaller coupling
- Increases re-usability and possibly reducing future redundancy

Dependency Inversion Principle

1. High-level modules should not depend on low-level modules. *Both should depend on abstractions*
2. Abstractions should not depend on details. Details should *depend on abstractions*

This allows for scalability and extendibility and again **Reduces coupling** and reduces the amount of refactoring.

Dependency Injection: achieving dependency inversion

If we have some object that is hard coded, could be a high level interface or a lower-level development object. We could achieve Dependency Injection through a constructor by eliminating dependencies from methods.

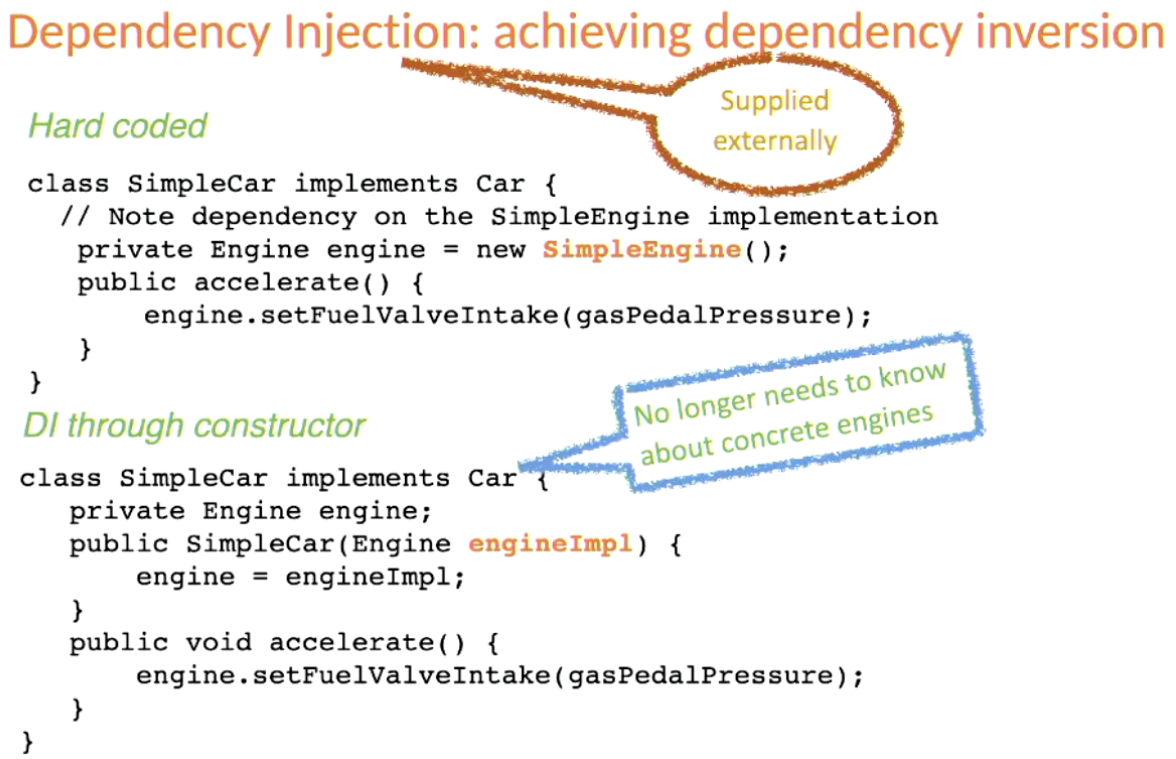


Figure 10: Dependency injection on an example of substatution

We should use the highest abstraction available to us (as this gives us the least dependencies). *Again using interfaces rather than concrete classes*

SOLID Principles

- The first five object orientated design principles
- By Uncle Bob (Robert Martin)
- Single Responsibility
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion

Lecture Thirteen-2: Design patterns

Design patterns are a high-level abstraction that allows us to easily structure models into an easy to understand and is a way of constructing *Literate Programming*.

Design patterns particularly the object orientated design allows for more code reuse, readability and allows modeling to map to real world concepts through the use of objects.

What is a design pattern

- Distilled wisdom about a specific problem that occurs frequently in Object Orientated design
- A reusable *design* micro-architecture
- The core of a design pattern is a simple class diagram with extensive explanation
- Patterns are *discovered*, as appose to written/invented
- Solution to common problems
- Imperfect mapping directly to software

We have three main types of design patterns, `Creational patterns`, `Structural patterns` and `Behavioral patterns`.

Creational patterns

Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Singleton	Ensure a class only has one instance, and provide a global point of access to it.

Figure 11: Creational Patterns

Structural Patterns

Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Façade	Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Proxy	Provide a surrogate or placeholder for another object to control access to it.

Figure 12: Structural Patterns

Behavioral Patterns

Chain of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate the request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in that language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Figure 13: Behavioral Patterns

Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State	Allow an object to alter its behavior when internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable, Strategy lets the algorithm vary independently from clients that use it.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	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.

Figure 14: Behavioral Patterns two

Lecture Fourteen: Design patterns - First look at specific patterns

A design pattern by definition is *A solution to a problem within a specified context*

In this segment we will go over some of the specific design patterns and some principles and lessons we can take from these.

The Iterator Pattern (Cursor)

- Problem
  - Sequentially access the elements of a collection without exposing implementation
  - Allow for different types of traversals
  - Allow multiple traversals at the same time
- Solution
  - Move responsibility for traversal from the collection into an Iterator object. It knows current position and traversal mechanism
  - The collection creates an appropriate Iterator

We need to use design patterns in order to achieve the following

- Correctness

- Resources
- Structure
- Construction
- Usage
- Reducing coupling

Often when we are implementing design patterns, we are going to have to take trade offs for certain necessary achievements we value within our project, it is important to be aware of these trade offs when choosing a design pattern for your project.

It is impossible to have an *optimal* solution, arguments can be made for which patterns are better than others however it really comes down to **which design pattern is best for the situation**.

The GOF style of documenting patterns

- Name
- Intent
  - Brief synopsis
- Motivation
  - The context of the problem
- Applicability
  - Circumstances under which the pattern applies
- Structure
  - Class diagram of solution
- Participants
  - Explanation of the classes and objects and their roles
- Collaborations
  - Explanation of how the classes and objects cooperate
- Consequences
  - Discussion of impacts, benefits and costs to implementation
- Implementation
  - Discussion of techniques, traps and language dependent issues
- Sample code

- Known use cases
- Related patterns and comparison

Documenting pattern instances

- Map each participant in the GoF pattern to correspond element
- Interface/Abstract
- Concrete
- Association

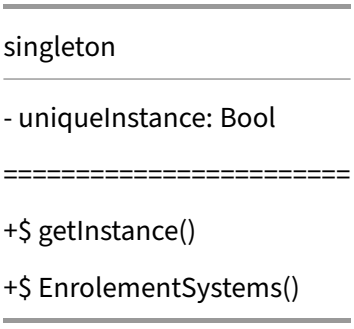
Use these terms to map items from the model to our classes in our class diagrams.

Singleton Pattern

This pattern ensures that a class only has one instance and provide a global point of access to it

- Problem
  - Some classes should only have a single instance
  - How can we ensure someone doesn't construct another one
  - How should other code find the one instance
- Solution
  - Make the *constructor private*
    - Making this in-accessible prevents accidental making of the item
  - Use a *static attribute* in the class that holds *one instance*
  - Add a *static getter* For the instance

Here is a UML example of what an object might look like using this pattern.



This is an example of a singleton object, that has static public methods and a private instance to check if the class has already been instantiated, note this is a lazy implementation of this design pattern for this class.

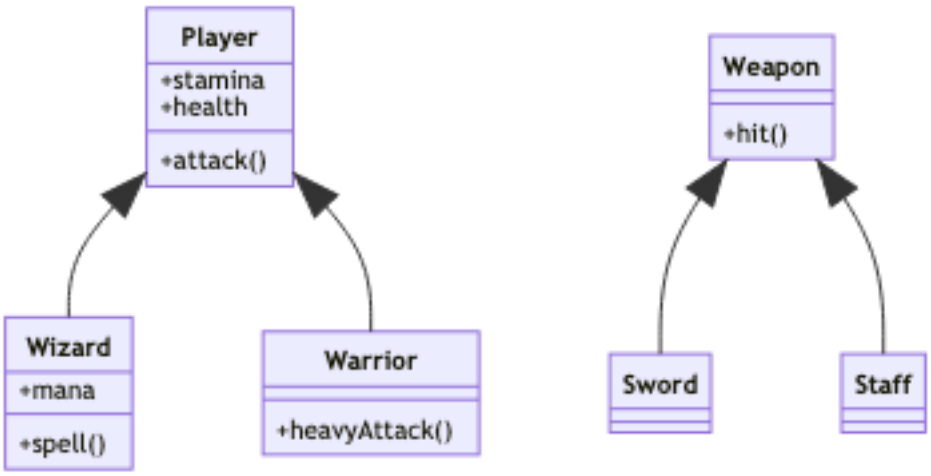
Lecture Fifteen: Specific Creation Design Patterns - continued

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

- This allows for changing of how we instantiate a class and we do not need to change code produced using the factory method.
- **Problem**:
  - Normally, code that expects an object of a particular class does not need to know which subclass the object belongs to.
  - Exception: when you create an object you need to know its exact class, the `new` operator increases coupling! *don't use factory in this instance*.
  - Need a way to create the right kind of object without knowing its exact class.

> Here is an example of a Virtual constructor, (factory method)

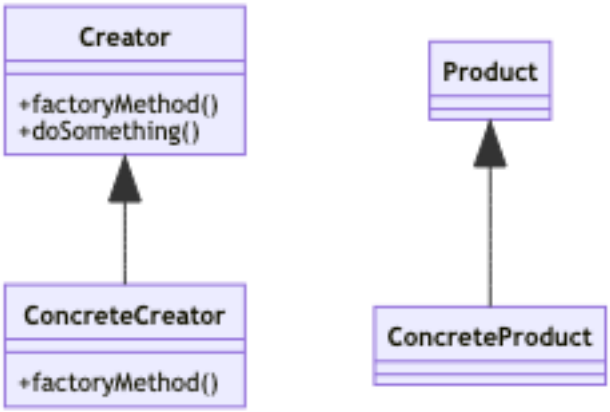


If the players `attack()` function looks something like the following:

```
if (player == Wizard) {
  weapon = new Wand();
} else if (player == Fighter) {
  weapon = new Sword();
}
weapon.hit(enemy);
```

This is a problem, because now the player also has to have knowledge of the weapon class, therefor changing the weapons class will affect the player class. This is an example of coupling, and should be avoided if possible.

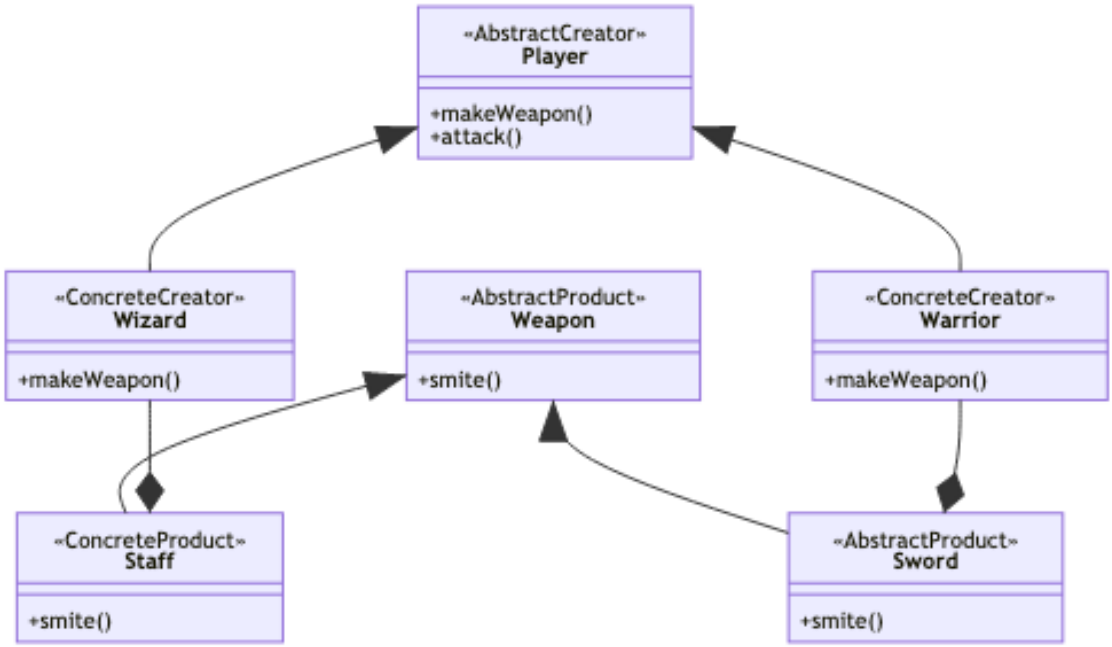
- **Solution:**
  - Move the *new* into an abstract method.
  - *Override that method to create the right subclass object*.
  - In practice this follows the general structure:



As we can see, `factoryMethod()` is abstract in `Creator` class, and is being overwritten by the concrete creator.

The concrete creator is linked to each concrete product in order to create these instances of classes.

> Here is an implementation of the factory method



Now that we have implemented a design, we can see how it maps to the Gang of Four design pattern.

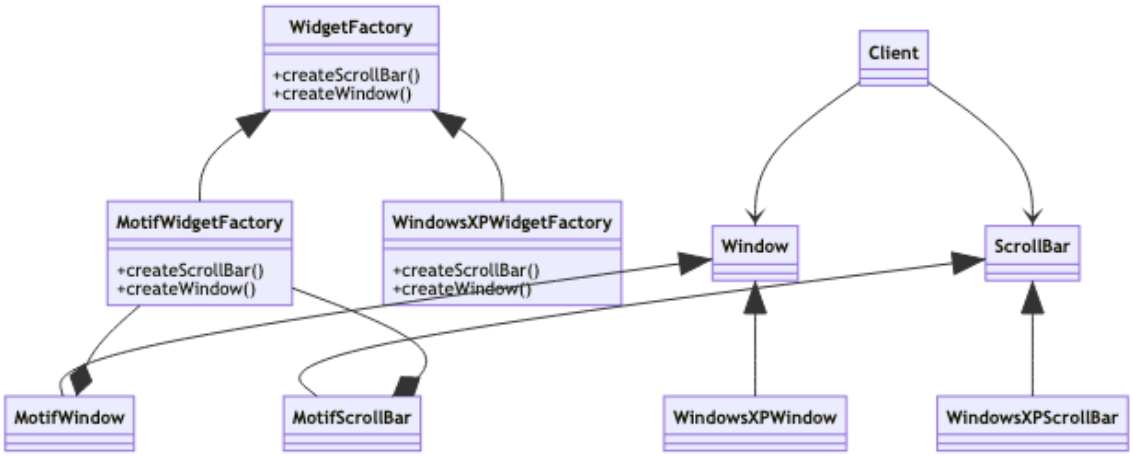
GoF	Our Design
AbstractCreator	Player
Concrete Creator	Wizard
Concrete Creator	Warrior
AbstractProduct	Weapon
ConcreteProduct	Staff
ConcreteProduct	Sword
FactoryMethod	‘makeWeapon()

We can have fancier methods with this design, `factoryMethods` can have parameters, and `factoryMethod()` is able to produce more than one type of product.

We can also throw in the class itself `makeWeapon(self)` in order to make decisions based on things like height, level…

Abstract Factory

We might define an abstract class as a `Window`, and then we can provide other components to this family of components to draw the window, such that we create classes for `Scrollbar`, `Application` and allow them to both be drawn on the window, we end up with the following setup:



Here is how this abstract factory design maps to GoF

GoF	Our Design
AbstractFactory	WidgetFactory
ConcreteFactory1	MotifWidgetFactory
ConcreteFactory2	WindowsXPWidgetFactory
AbstractProductA	Window
ConcreteProductA1	MotifWindow
ConcreteProductA2	WindowsXPWindow
AbstractProductB	ScrollBar
ConcreteProductB1	MotifScrollBar
ConcreteProductB2	WindowsXPScrollBar
…	…

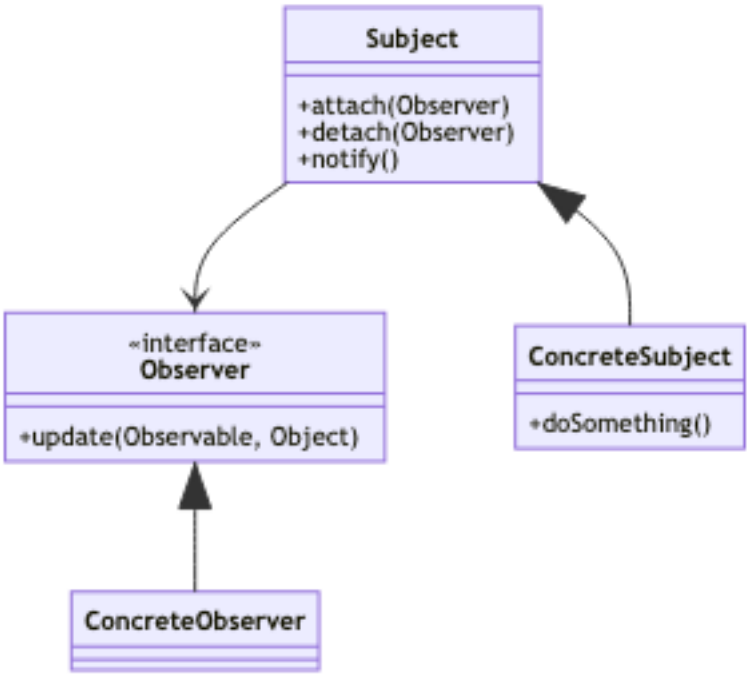
Observer Pattern

- Problem:
  - Separate concerns into different classes, but keep them in sync

        * Separate GUI code from model
    - Avoid tight coupling

- Solution:
    - Separate into subject and observers
        * Can have many observers for one subject
    - The subject knows which objects are observing it, but does not know anything else about them
    - when the subject changes, all observers are notified

Here is an observer mapped to Gang of Four



Java was built onto the Observer pattern, however this is depreciated from `Java 9` onwards.

- Allows many Observers to many subjects
- Adds a *dirty* flag to help avoid notifications at wrong time
- Swing `EventListner's` are another variant of Observer
- Java 11 has 93 classes with `Listner` in their names

This maps to Gang of Four as follows:

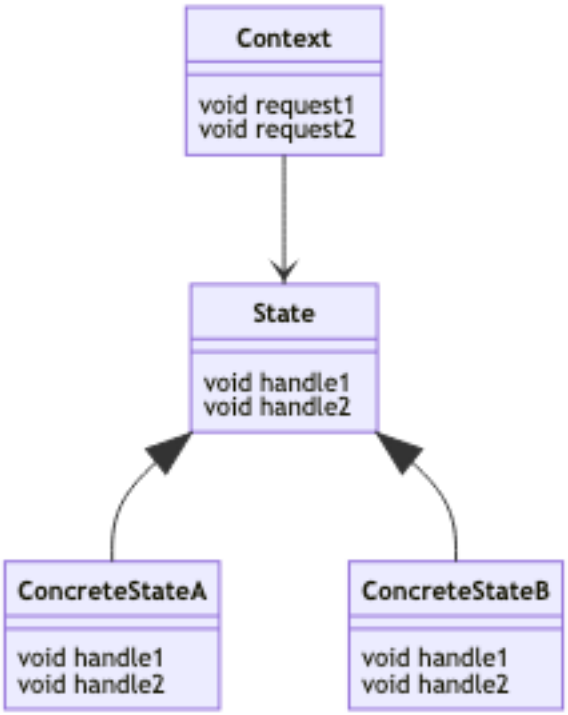
Gang Of Four	Signal App
Subject	SignalFace
Observer	SignalFaceObserver
Concrete Subject	SignalFace
doSomething()	display()
getterA	getColor()
Subject - Observer Relationship	SignalFace - SignalFaceObserver Relationship
Cocrete Subject - Concrete Observer Relationship	SignalFace - 2DSignalFaceGUI Relationship

Lecture Sixteen: State, Command, Template Patterns

State Pattern

The intention is to allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

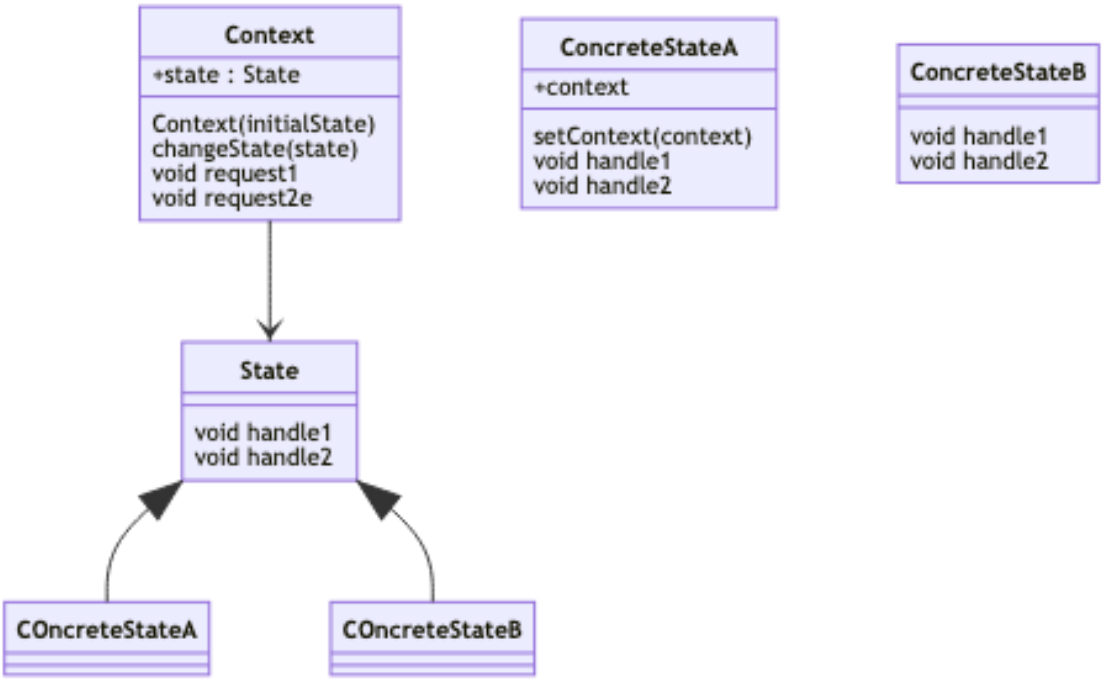
- State chart is a representation of an objects behaviour.
  - How can we add states without updating the general behaviour?
- State-specific behaviour encapsulated in concrete classes
- State interface declares *one or more* state specific methods - should make sense to all concrete states
- Context communicates via State interface



Implementing State Pattern

- Context, create states can perform state transition - by replacing context's state object
- If transition criteria fixed, then can be done in context - concrete state classes then can be independent
- Initial state set in context
- Explicit transitions prevent getting into inconsistent state

> Here is how this would be implemented in practice



Here is a real implementation of this

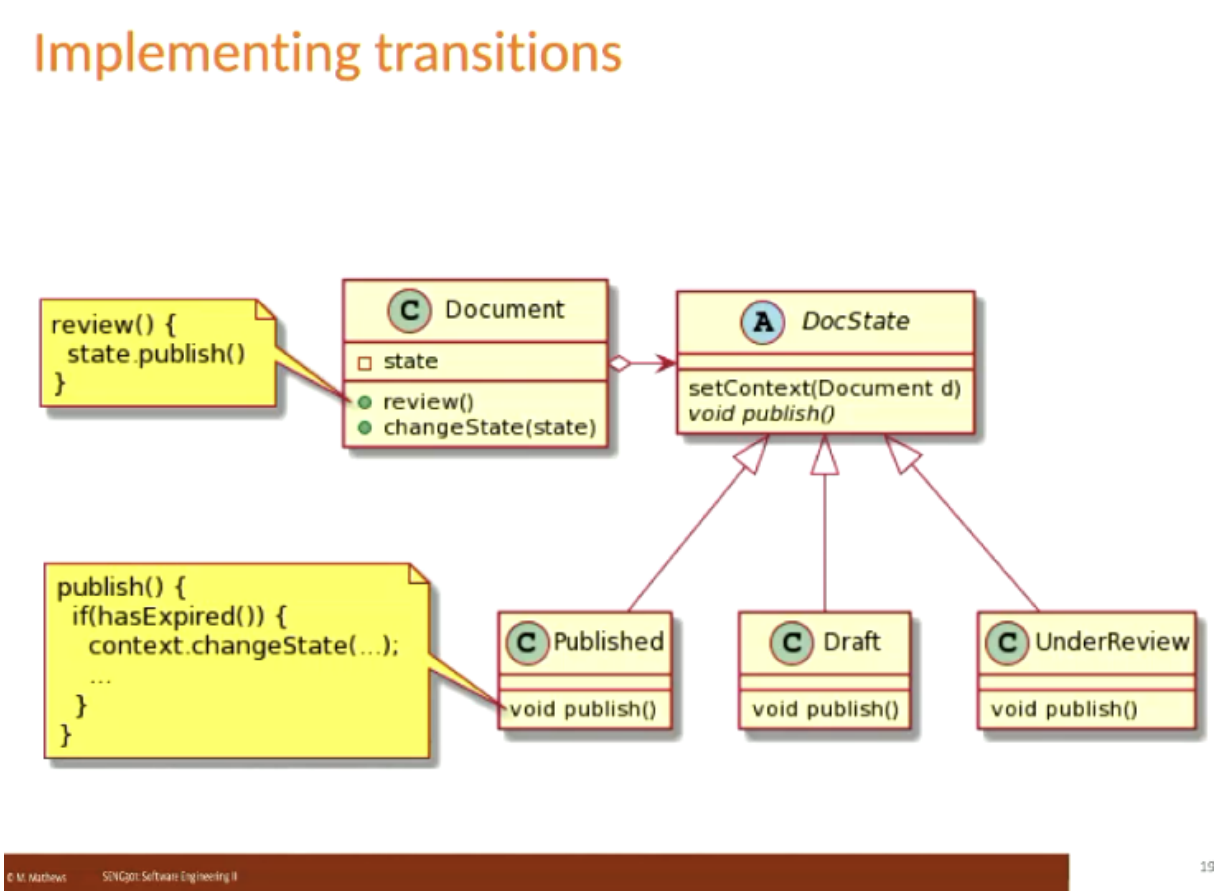


Figure 15: State diagram for a document

Command Pattern

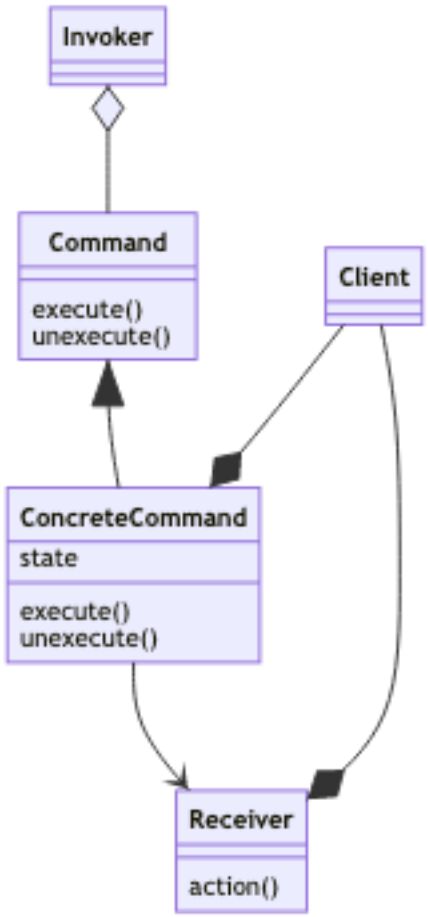
- Encapsulate a request in an object to:
  - Parameterize clients with different requests
  - queue or log requests
  - support undoing operations

Command pattern setup

- Command
  - Declares interface for executing operation
- ConcreteCommand
  - Implements `execute()`
  - Defines binding between receiver object and an action

- Client
  - Creates ConcreteCommand and sets receiver
- Invoker
  - Asks command to carry out requests
- Receiver
  - knows how to carry out requests (can be any class)

Here is how this maps to a class diagram:



Here is an example of how this might map to a real world case for a Smart Home.

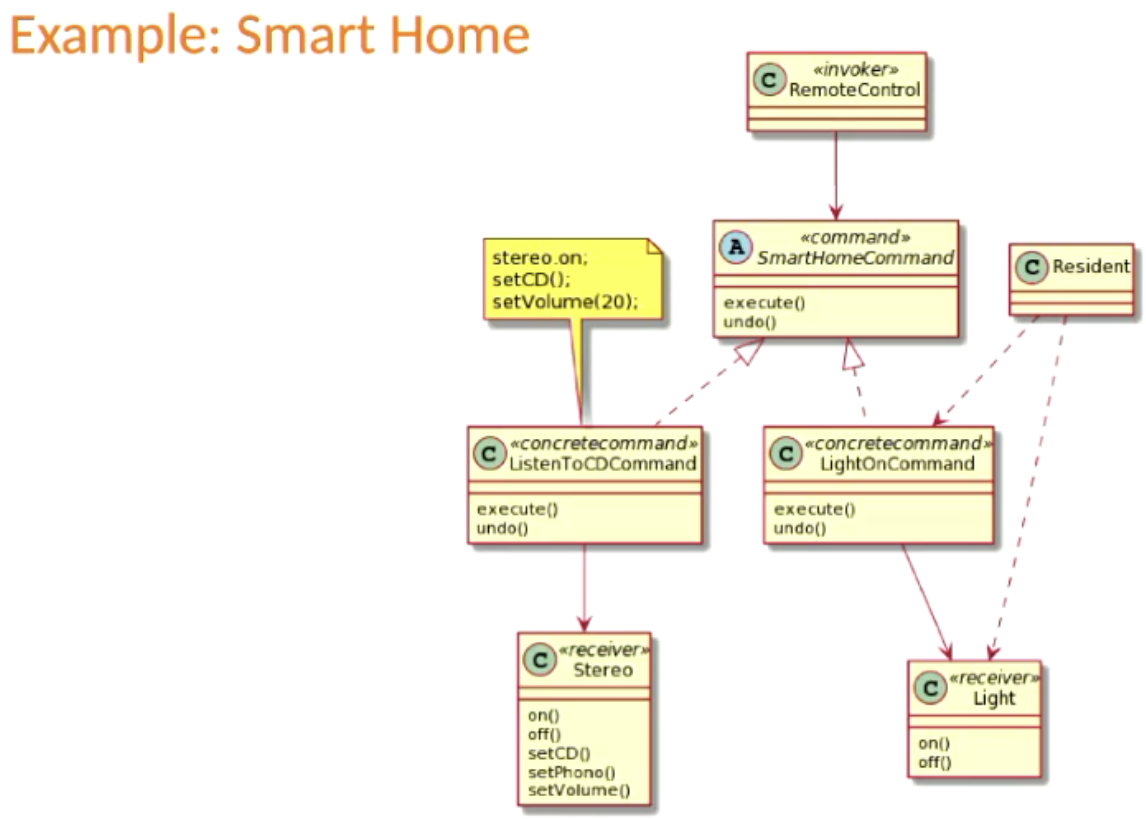


Figure 16: Smart Home Example

GoF	Our Design (example)
Command	SmartHomeCommand
ConcreteCommand	LightOnCommand
ConcreteCommand	ListenToCDCommand
Receiver	Stereo
Receiver	Light
Invoker	RemoteControl
Client	Resident

Template Method: Encapsulating algorithms

We can use functions in order to define the actions within a game, we can have multiple games that

follow the same general sequence. We can use abstract template classes in order to have an abstract game design that can be templated towards specific instances of game, i.e. `Chess`, `Monopoly`

Here is an implementation of an abstract class to implement in this idea.

```
abstract class Game {
  protected int playersCount
  abstract void initalizeGame();
  abstract void makePlay(int player);
  abstract boolean finishGame(); abstract void printWinner();
}
```

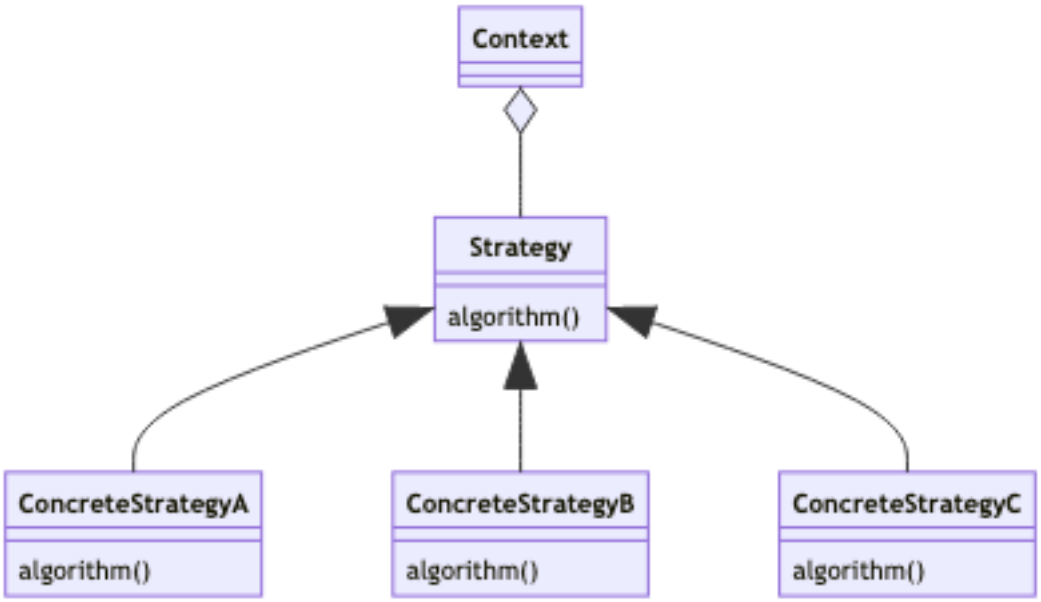
Hooks in template method

- Can include hooks so subclasses can hook into algorithm at suitable points
- Hook declared in abstract class with empty or default implementation
- Subclasses are free to ignore hook

Lecture Sixteen: Strategy, Composite, Decorator Design Patterns

Strategy

- Problem
  - Change an object's algorithm dynamically rather than through inheritance
- Solution
  - Move the algorithms into their own class hierarchy
- Notes
  - context know different strategies exist
  - Strategy needs access to relevant context data *parameter or reference*



Here is how this model could be mapped to a real example:

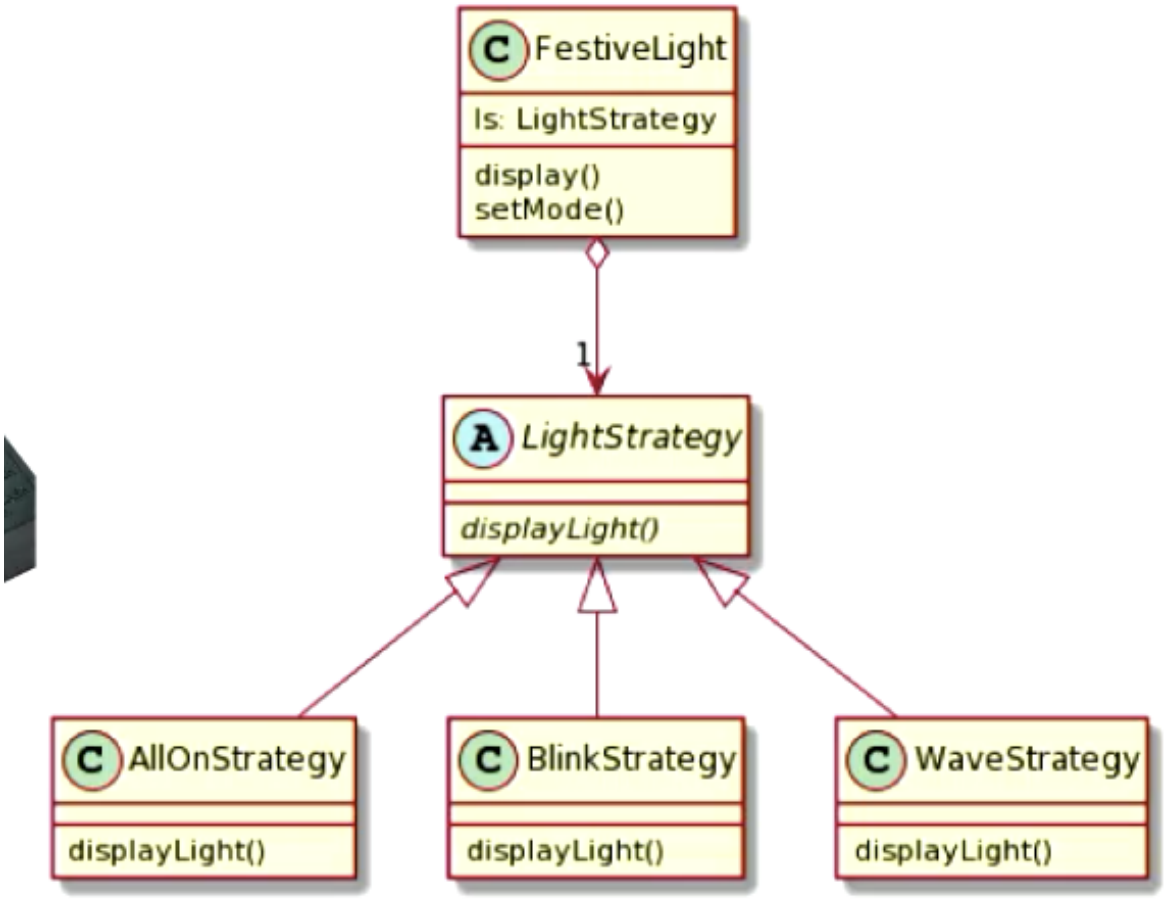


Figure 17: Festive Lights Strategy Implementation

GoF	Our Design
Context	FestiveLight
Strategy	LightStrategy
ConcreteStrategyA	AllOnStrategy
ConcreteStrategyB	BlinkStrategy
ConcreteStrategyC	WaveStrategy

Composite Pattern

- Problem
  - When an object contains other objects to form a tree, how can client code treat the composite objects and the atomic objects uniformly?
- Solution
  - Create an abstract superclass that represents **both** composite and atomic objects.
  - Easy to add new components
  - Common for child to know parent
  - Can make containment too general

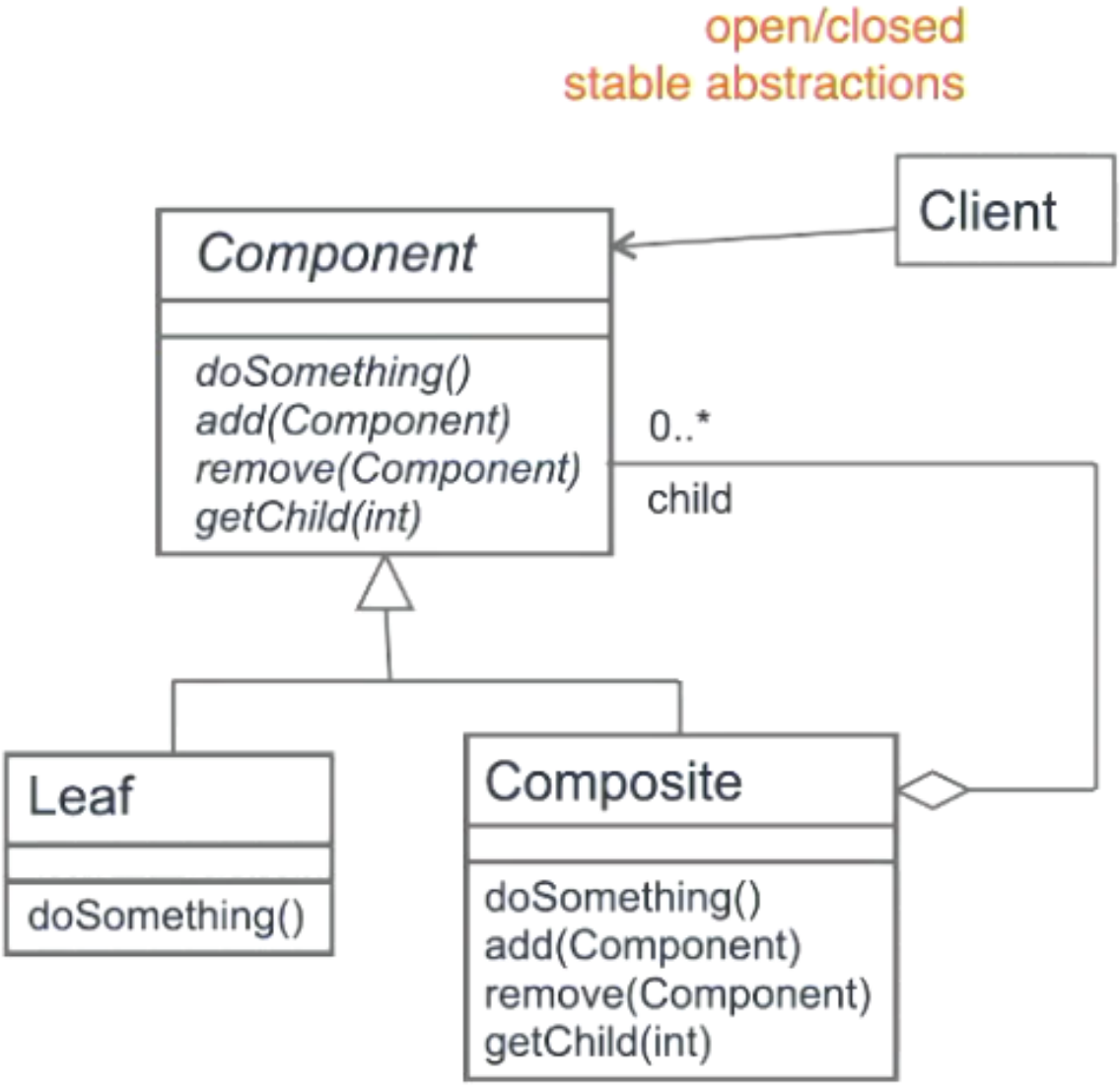


Figure 18: Composite Pattern Generic Mapping

Here is a real mapping to a real world scenario

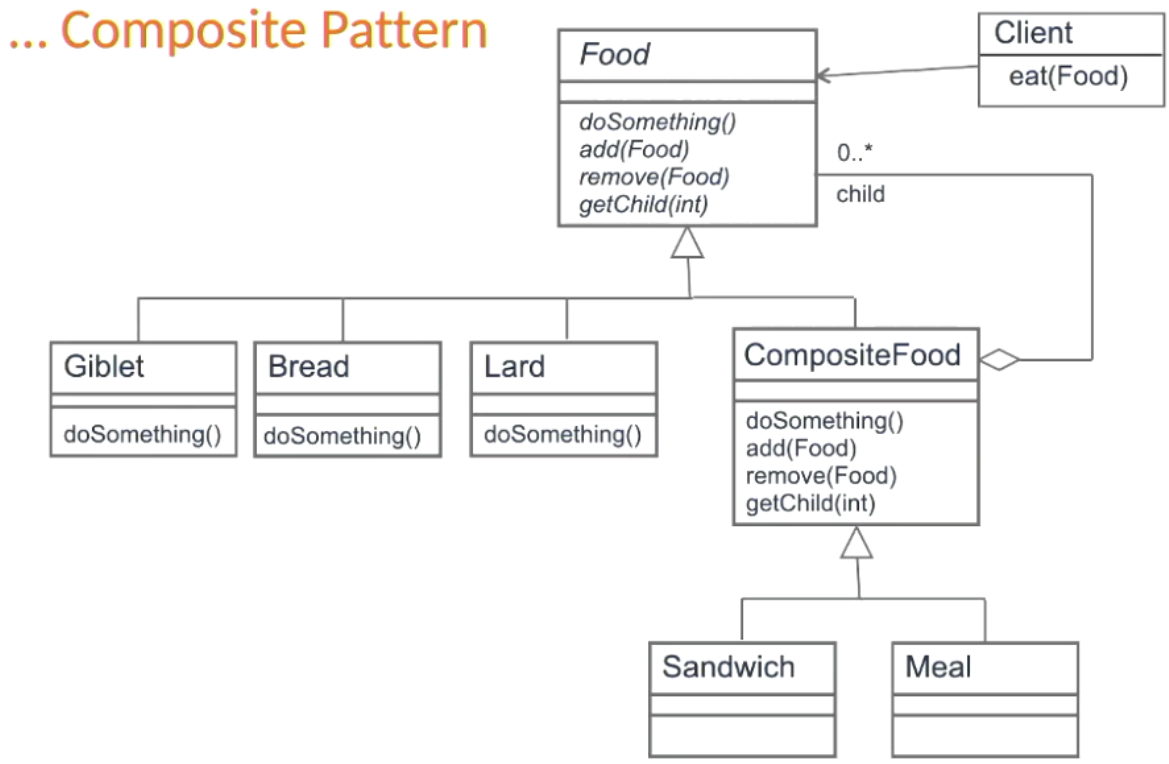


Figure 19: Composite Pattern Example Mapping

We can see how the generic mapping maps directly to the composite example.

> Good exercise: Write out GoF mapping tables.

Decorator Pattern

The goal of this pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality

- Problem
  - What if we want to have multiple states?
- Solution
  - Using aggregation instead of sub-classing
- Notes
  - Decorator is asking *whats different?*

- Favour composition over inheritance explanation
  * Inheritance: designing types/classes over what they *are*
  * Composition: designing types/classes over what they *do*
- Concrete decorator knows the component it decorates
- Business rules - e.g. number, order of decorations
- We can end up with a lot of classes, works best with large differences between classes

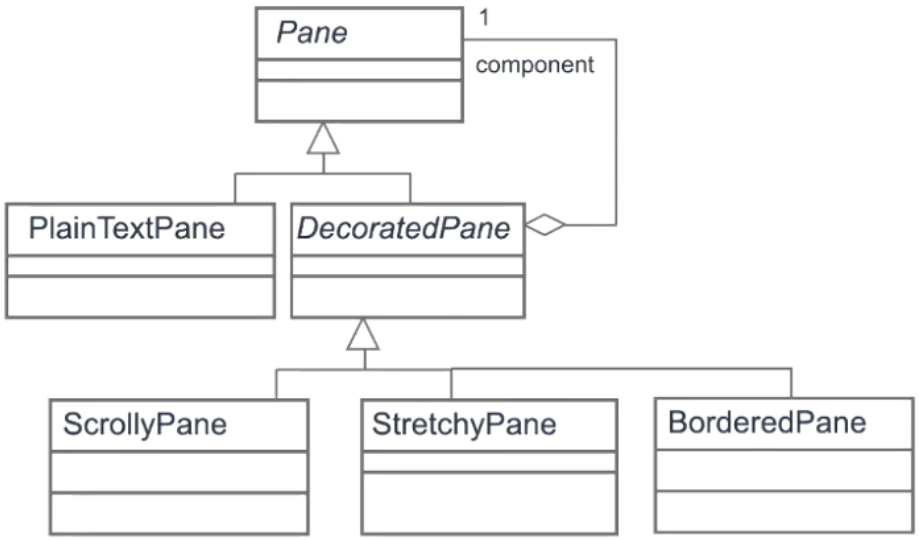


Figure 20: Class diagram of Decorator pattern

> Key parts of collections and design patterns is intent, problem, solutions

Combining Patterns

Abstract Factory meets Singleton

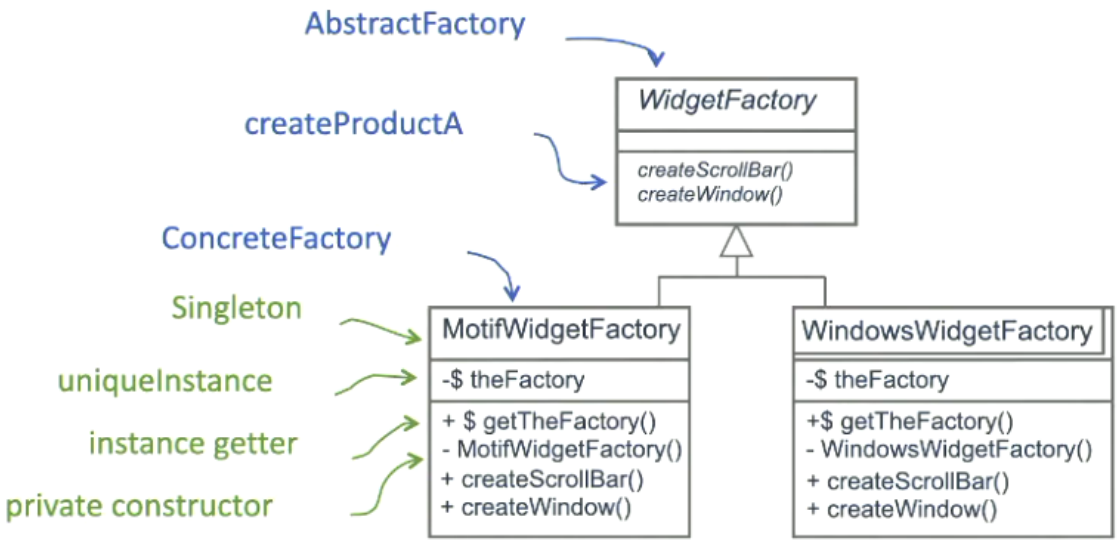


Figure 21: Abstract factory + singleton

Iterator meets Factory Method

The below implementation is a simple overview, not how they are actually mapped.

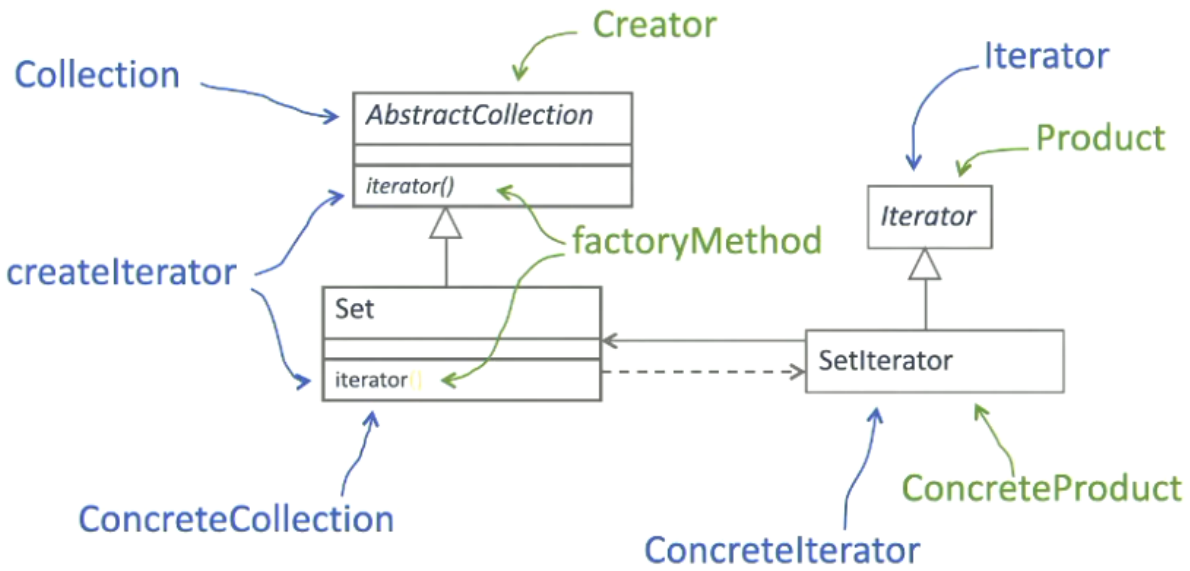


Figure 22: Factory + Iterator

Lecture Seventeen: Design by Contract

- Contract for `dbcLecture()`
  - When the lecture begins, the *student* will:
    * Be present
    * Know OO
    * Be conscious
  - When the lecture ends, the *lecturer* will
    * Teach and share knowledge with those in the lecture

Lots of things are built on contracts, some of them implicit and some explicit, designing by contract is a mutual trust between two classes. For example a `Client` has some call to `Server` requesting a `service()` from the server.

This is done by setting up the following:

- **Setting preconditions:** before calling service, client checks its request is ready.
- **Postconditions:** server promises it has done the job.
- **Invariant:** Something the service promises will **always** be true.

> If the server and client are expecting certain values, then they are bound by an implicit contract in design.

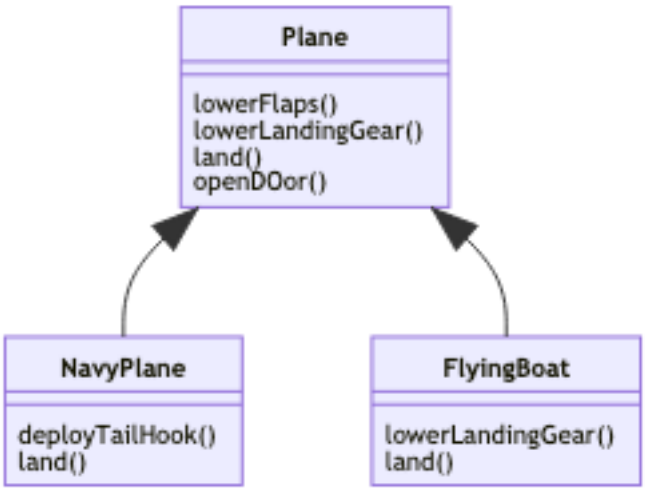
One good method of documenting preconditions and postconditions is to do so in the docstrings, this will allow users to more easily see how to use external functions.

- Precondition: *stack is not empty*
- Postcondition: *stack unchanged, returned last pushed object*

Contracts can also be used when doing testing, we want to know what happens if we meet the preconditions or not, see what happens to our output values if the post conditions are not met. This comes with some side affects.

- Unit Testing
- AT's
- Assertions
  - Do preconditions hold?
  - Are invariants invariant?
  - Do post conditions hold?
  - Are there side-effects?
  - Do exceptions occur?

Contracts and Inheritance



Precondition in this example

- Precondition for `Plane`: Flaps down & gear down
- Precondition for `NavyPlane`: Flaps down & gear down & hook down

Because the precondition of the `NavyPlane` is different and requires more than the precondition for the parent class `Plane`, we would have to specifically ask if every plane is a navy plane in order to determine whether to put the hook down, this is a bad idea as it means that we must address different instances of the same parent class differently.

> This is known as `not obeying the contract that has been inherited`, for more information about this visit Lecture Eighteen

Lecture Eighteen: Inheriting a Contract

Contracts can be inherited from parent classes:

Inheriting a contract

- Contracts are inherited
  - Preconditions can be loosened
  - Postconditions can be tightened
  - Postconditions can be tightened

Contract guidelines

- No preconditions on queries
  - it should be safe to ask a question
- No fine print
  - Don't require something the client can't determine i.e. preconditions should be supported by public methods
  - It is OK to have postconditions a client cannot verify
    * This is the services problem, not us.
- Use real code where possible
  - Better to say `isEmpty()` rather than `the stack must not be empty`
- Can't show all semantics in code
  - `pop()` returns last pushed object
- No hidden clauses
  - The preconditions are sufficient and complete
- No redundant checking
  - Don't check that preconditions are satisfied inside server!

Redefining Inheritance

Previously we have described inheritance as the following `A ChildClass is a type of Parent`, we can better define a as inheritance as `a subclass that conforms to the contract of the parent class`.

Specifying Contracts: Formal Methods

- Mathematircally based
- Z, B, VDM, ...
- Pre-conditions, post conditions and inveriants specified
- Reasoning
- Code generation

Here is an example of how we can specify a contract:

$$
\begin{array}{|l}
\hline
AddBirthday \\
\Delta BirthdayBook \\
name? : NAME \\
date? : DATE \\
\hline
name? \notin known \\
birthday' = birthday \cup \{name? \mapsto date?\} \\
\hline
\end{array}
$$

Figure 23: contract specification

Formal support for contracts is not commonly used, however it is supported in `UML` as a part of Object Constraint Language. It also can be shown in code, we can use the `assert` method in Java, *Note: not implemented in java by default*

A philosophy for using exceptions

- Use Java exceptions if a contract violation occurs
- Handling violations
  - If possible fix problem, otherwise
  - if possible try an alternate approach, otherwise
  - clean up and throw an exception
- Interfaces **are** contracts
  - whenever a contract can be recognised independent from a particular implementation, an interface should be considered
  - Interfaces can be composed; one class may implement many interfaces.
  - Interfaces can be extended to specialise contracts.

Contracts and state machines

Do we want ot represent state behaviour as apart of the contract?

Here is an example of state being used as apart of a contract form to represent how traffic lights might change, *orange before green*, *orange before red*.

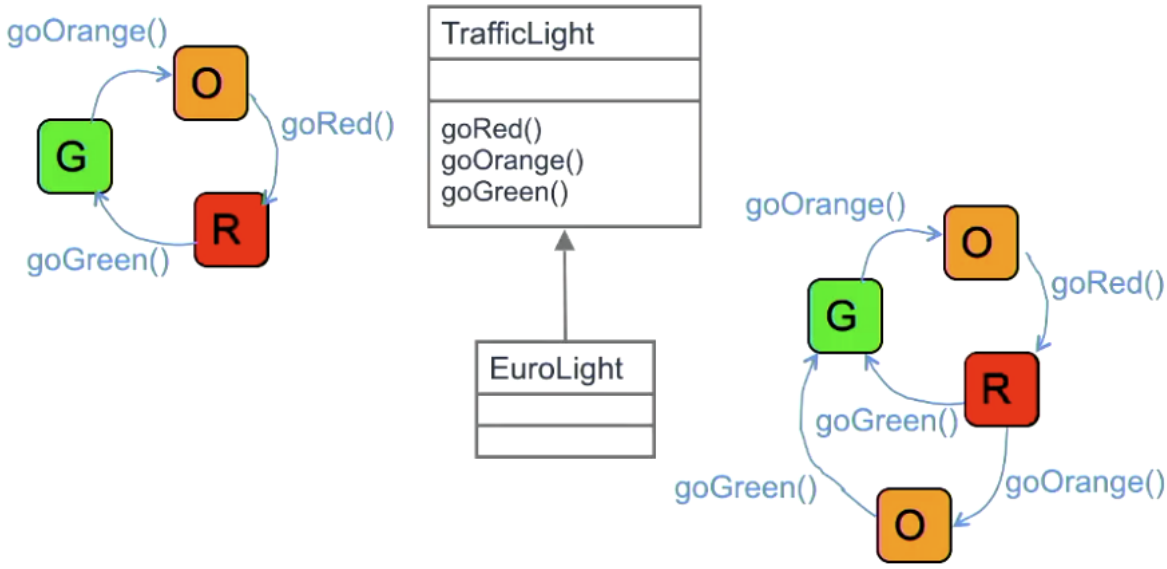


Figure 24: state contract idea

Inheritance: The Dark Side

- Inheritance for implementation
  - No intention to honour the inherited contract
- Is-a-role-of
  - Merging of two contracts
- Becomes
  - Switching contracts
- Over-specialization
  - Contract more specific than necessary
- Violating the LSP
  - Breaking the contract

Lecture Nineteen: Code Smells

What are code smells?

Code smells are soft rules that determine whether code is well written or poorly written

When we talk about code smells:

- Code smells are built of huristics, not hard rules
- Design/software rot: code left alone smells bad *gets bad over time*
- We should prioritise when refactoring code to match our heuristics
- Refactor catalog
  - Split
  - Join
  - Move
  - Extract
  - Rename
  - …
- How large is large?
  - Code smells are subjective
    * We can determine a code smell without evidence, however evidence will help in order to be justified in your smell
    * Easier to detect in code we know rather than unfamiliar code
  - Can be informed by metrics and measurements *code length, coverage etc.*
  - Gather data, manually or automatically
  - Analyse results, statistics, visualisation
  - Percentiles

We can look at the **Morphology** (Shape) of code, do we have too much information in our class? Am I called often enough to be justified in the code base? Am I redundant? Am I reusable?. These are important questions to answer in order to find redundancy and code smells.

We can use terms to define these ideas:

Fan Out

- How many other functions do I call?
- Am I too big?

Fan In - How many others call me? - Am I reusable?

We can group terms into *fan in* vs *fan out*, by doing this we can graph them in order to determine which functions are *better* than another.

Length Metrics

- Number of lines of code (LOC)
  - Number of statements?

  - Comments included?
  - Declarations included?
  - Whitespace included?

- Amount of logic
  - Branching and conditionals
  - Number of decisions

- Include nesting?

Long Method Smell: Specific code smell

Methods are too long

Problem because?

- Methods are doing too many things
- Single responsibility principle
- Cohesion

How long is *too* long?

- Length Of Code, length metrics
- Counting rules
- Distribution analyses
- Percentiles
- Correlated metrics
- Visualisation

By removing long methods, we are likely to make code reusable due to the nature of removing long methods.

What should we do?

- Break into multiple methods
- Refactoring: **Extract Method**

Large class smell

“A class with too much code is prime breeding ground for duplicated code, chaos and death” - Martin Fowler

We can end up with the idea of a `GOD Class`, A class that knows far too much of the code base, this happens often if a class is not maintained correctly, we get difficult code to understand, maintain, and adds to the complexity of the code base. This also creates a single point of failure.

Classes should have a single responsibility, to prevent coupling and cohesion.

Long parameter list

When this happens, we tend to have complex understanding of the code, generally a bad sign that the code is being implemented in a poor way.

- If parameter list is too long, what does that tell us about the method?
  - How complex is method?
  - Single responsibility
  - God method?
  - Coupling
- Solutions
  - Introduce a parameter object
  - Preserve whole object
  - Replace parameter list with a method call

Duplicated code

- Once and only once, “there should be one single source of truth for an instance”
  - We don’t want to change multiple methods to achieve the same task in different instances

This happens often when we end up copy and pasting without fully understanding code, or copy and pasting from other places in the Repository.

- Solutions
  - Extract Method

Lecture Twenty: Code Smells - continued

- Dead code:
  - Source code that might be executed but the result is never used. Unreferenced variables / functions that aren't called are not dead code, as they're automatically removed via compiler / linker.
- Unreachable code:
  - Code you can't get to – often in a switch statement. Could be after a return statement. No control flow path to the code. Makes code harder to read, takes up more memory, might take up more cache (e.g. instruction cache on the CPU).
- Deactivated code:
  - executable code that isn't executed now (as it is part of a different version/release).
    - if(IE6) {...}
- Commented code
  - Code that has been commented out. Misleading. Difficult to read and maintain.
  - Why was it commented out? Is it needed? Can we delete it?

Figure 25: Dead unreachable, deactivated and commented code

Switch statement smell

- Large switch/if statements
- What does this mean
- Conditional complexity
- Code for switch may extend over the full code base
- OOP should have rare switch statements
  - Use polymorphism, Implement OOP correctly
- LSP
- Solution:
  - Switch on types? Replace conditional with polymorphism
  - Switch on type code? Replace type code with subclasses

  - Often checking against null? Introduce null object

If switch performs simple actions then leave it! There is a sweet spot where these are extremely useful.

Comments

- Comments are for humans
- Integrity: are comments up-to-date? Can we trust them?
- Comments vs Javadoc
- Is the code to complex to understand
- Time/energy in keeping comments up to date vs how much developers trust comments
- Solutions
  - If expression/method is too difficult and complex, extract variable

Type embedded in name

Using names like `strValue()` may be useful for identifying variables type. -> Can be achieved with normal lsp?

Speculative generality

- Making general solutions because you are speculating or anticipating what we might need in the future
- Do not over-engineer your solution
- Do not speculate about tomorrows problems
- How do we balance this with planning for extensibility?
  - Collapse of hierarchy
  - Rename method
  - In-line class (opposite of Extract class)

Inappropriate intimacy

- How much does one class know about another?
- Coupled?
- Solutions:
  - Move methods
  - Move fields
  - Change bi-directional association to unidirectional

    * e.g. Stack/Vector example: composition vs inheritance

Indecent exposure

- Exposing your internals
- OOP was under scrutiny for being slow, because we have to address an object rather than directly grab a variable however this can be stopped by implementing OOP correctly.

Feature Envy

- Methods making extensive use of another class, they are *envious* of a specific method

Shotgun Surgery

- Changes made to many parts of a system
- Single responsibility violated?
- Move methods/data?
- Create new class?

Opposite of divergent change - many changes in the same class

Can tests smell too?

Yes! There are many tests that are used to implement constraints for code smells, it is not uncommon in todays world that we can have testing files that are larger than the code base we are testing. As we push more towards testing, it is more and more likely we will start enforcing optional constraints for programmers to work within, in order to increase readability and understanding within the code base.