Object Oriented Eng Analysis (COE 528): Notes

Adam Szava

Winter 2022

Introduction

This document is a compilation of my notes from Object Oriented Eng Analysis (COE 528) from Ryerson University. All information comes from my professor's lectures and online resources.

Contents

1 Week 1: Java Review

This course focuses on software development using Java. In software development there are four main activities:

1. Requirement Elicitation and Analysis
    - Features (Also called functional requirements)
    - Security
    - Maintainability
    - Scalability
    - Portability/Compatibility
    - Usability
    - Accessibility
    - Reliability (Can run for an extended amount of time after started)
    - Availability (Is available to begin often)
2. Object oriented design (UML)
    - Identify classes and relationships.
    - Identify methods (API) and write comments
3. Implementation
    - Translate design into code.
4. Testing the software for functionality.

This course discusses **design pattern** which are a way to design code based off of some design constraints.

Java Review

The main ideas of object oriented programming (OOP) are:

- Classes (a blueprint of an object)
- Objects
- Encapsulation and scope
- Inheritance
- Polymorphism

Data Types

We have two different kinds of data types:

1. Primitive Data Types:
    - int
    - float
    - double
    - byte
    - short
    - char
    - boolean
    - long
2. Reference Data Types:
    - obj
    - int[]
    - String
    - ... and many more

When using primitive data types, the variable is passed by value into methods and other assignments. With reference data types only the reference is passed into the method or other assignments. For example in *pseudo-code*:

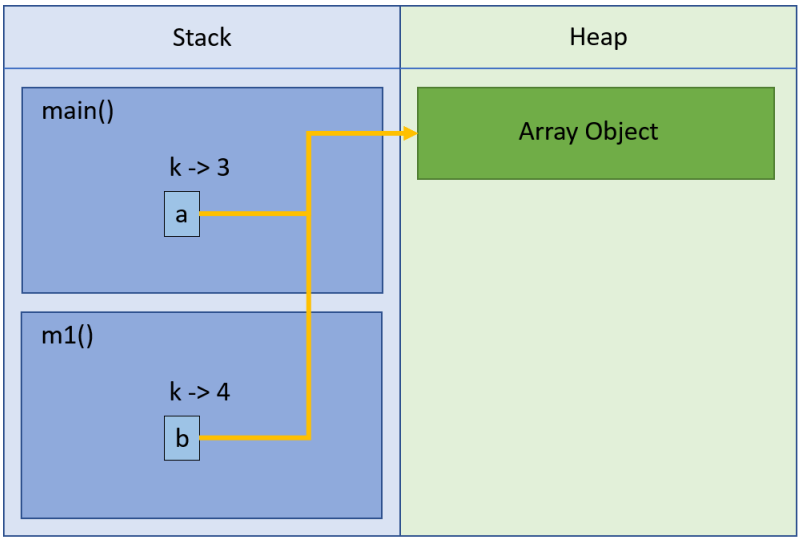
```
int x = 5;
int y = x;
x = 9;
print(x);
print(y);

Output:
9
5
```

Stack versus Heap

The stack is a section of memory allocated to storing primitive data types and reference variables. The heap is a section of memory allocated to storing data related to objects.

For example the following pseudo-code's memory consumption can be visualized as:

```
main(){
int k = 3;
int[] a = {...}
}

m1 (){
int k = 4;
int[] b = a;
}
```



Comparing Objects

Unless overridden, the following two lines behave exactly the same:

```
obj1.equals(obj2);

obj1 == obj2;
```

... as in the comparison checks whether the two reference variables point to the same object in the heap.

2 Week 2: Java Review

This week continues the Java review.

Final Keyword

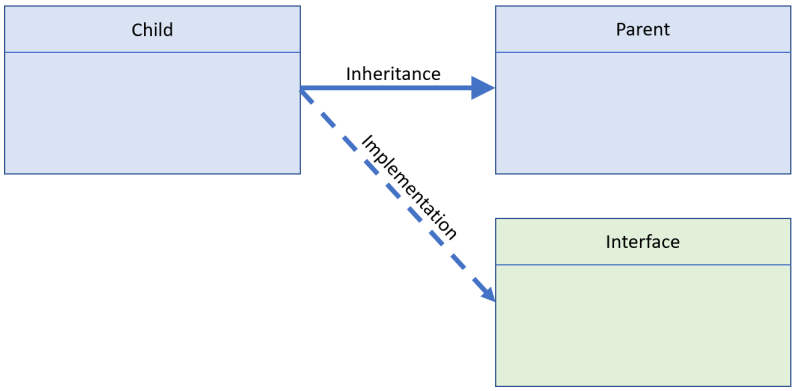
The final keyword is used in the following ways:

- Final variables are constant and must be declared immediately or in constructor.
- Final classes cannot have subclasses.
- Final methods cannot be overridden.

Class Relationships

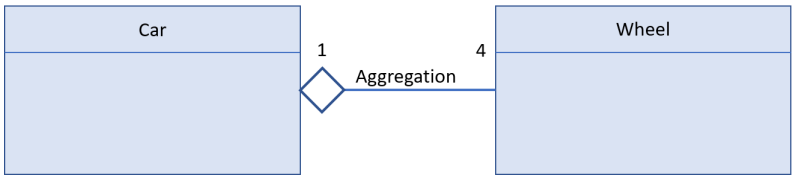
There are 3 main class relationships:

1. Is - A relationship, drawn as:



   This relationship represents a class which *is a* child of a parent class, or a class which *is an* implementation of a interface.

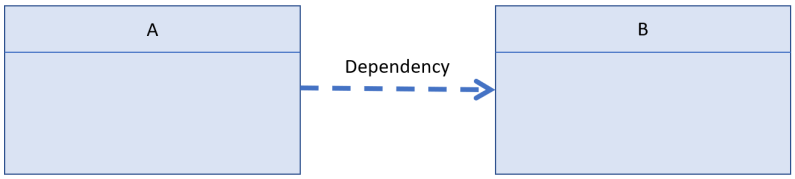
2. Has - A relationship, drawn as:



   Aggregation relationships have what's called a *multiplicity*. The example in the image has a multiplicity of 4 since 1 Car has 4 wheels. Generally we use the following symbols to denote multiplicity:

   - $*$ denotes 0 or more.
   - $+$ denotes 1 or more.
   - $a...b$ denotes a to b.

3. Uses relationship, drawn as:



This type of relationship means that the class A uses a method from the class B.

3 Week 3: Process Models, Abstraction/Decomposition, Procedural Abstraction

Process Models

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

As the scale and complexity of a project increases the development methods and project management must increase in formality. In this course we are interested in formal development methods.

The goal of software engineering is to acheive high quality (non-functional requirements) and productivity (input/output relationship of resources. Usually measured in lines of code per person month.) The methods used must also accommodate change and must be able to handle complex problems.

A **process** is defined as a sequence of steps to achieve some goal. A **software process** is defined as a sequence of steps performed to produce software with high quality, and within a budget/schedule.

A **process model** specifies a general process that is optimal for a class of problems. A project may select its process using one of the process models. Commonly a process model has the following phases:

- Requirement elicitation and analysis
- Design
- Coding (implementation)
- Testing
- Delivery (usually not included)

Different process models perform these phases in different ways and orders. We will discuss *five* process models:

- Waterfall (traditional)
- Prototyping (traditional)
- Iterative (traditional)

- Timeboxing (traditional)
- Agile (modern)

Waterfall Process Model

The waterfall method is the simplest model, you just do all the of the phases in linear order. A phase only starts when the previous has finished. The waterfall method is very document-heavy meaning there are formal documents between each phase.

The following are the important considerations about the waterfall process method:

- **Strengths:**
  - Simple to understand
  - Easy to execute
  - Intuitive and logical
  - Easy contractually
- **Weaknesses:**
  - All or nothing delivery.
  - Requirements are frozen early, not able to be changed.
  - Disallows changes in general.
  - No feedback from users.
  - Encourages requirement bloating (additional unnecessary features at the beginning since it cant be changed later).
- **Types of projects:**
  - Well understood problems.
  - Short duration projects.
  - Automation of existing manual systems.

Prototyping Model

This type of process model is where you create a prototype of the project with only key features at the beginning, features which need to be better understood. Then you do requirement analysis again, and go on to do the rest of the steps again in a linear sequence.

- **Strengths:**
  - Helps requirement elicitation.
  - Reduces risk.

  - Better and more stable final. product

- **Weaknesses:**
  - Front heavy (a lot of work at the start).
  - Higher cost and schedule.
  - Encourages requirement bloating.
  - Disallows later change.

- **Types of projects:**
  - Novice users.
  - Areas with high requirement uncertainty.
  - Heavy reporting based systems can benefit from prototype UI.

Iterative Model

The iterative process model is where you create the software in increments and each increment adds some functionality and is complete in itself. This can be viewed as a sequence of waterfalls for each iteration. Feedback from one iteration is used in future iterations.

- **Strengths:**
  - Regular deliveries.
  - Can accommodate changes naturally.
  - Allows user feedback.
  - Avoids requirement bloating.
  - Naturally prioritizes requirements.
  - Allows reasonable exit points (when development can end).
  - Reduces risk.

- **Weaknesses:**
  - Overhead of planning each iteration.
  - Total cost may increase.
  - System arch and design may suffer.
  - Rework may increase.

- **Types of projects:**
  - Time is important.
  - Risk of long projects cannot be taken.
  - Requirements not know and evolve.

Time Boxing Model

In the time boxing model, you break the project into multiple teams, which all have their own set of requirements, implementation and testing to do. The teams start at different times so that the project comes together quickly after the first team has finished.

- **Strengths:**
  - All the benefits of iterative.
  - Planning for iterations is easier.
  - Very short delivery time.
- **Weaknesses:**
  - Project management is more complex.
  - Team size is larger.
  - Lapses can lead to losses.
- **Types of projects:**
  - Short deliveries are important.
  - Flexibility in grouping features.

Agile Model

This is a modern umbrella term for many models based off of the prototyping and iterative ideas, commonly agile process models include the following principles:

- Working software is the measure of progress.
- Software should be delivered in small increments.
- Even late changes should be allowed.
- Prefer face to face communication over documentation.
- Continuous feedback to customer.
- Prefer simple design which evolves.
- Delivery dates are decided by the empowered teams.

The most common type of agile model is called **extreme programming** (XP) which starts with short descriptions of the customer's needs called "user stories". The stories are then combined in different ways to make the formal requirements. Planning is then done, and development is done in iterations. High risk features are done first as a sort of "prototype".

- **Strengths:**

  - Agile and responsive.
  - Short delivery cycles.
  - Continuous feedback and better acceptance.
- **Weaknesses:**
  - Can tend to be ad-hoc.
  - Lack of documentation.
  - Continuous code change is risky.
- **Types of projects:**
  - Requirements change a lot.
  - Customer is deeply engaged in the project.
  - Size is not too large.

Abstraction and Decomposition

A large project is very complex, thinking about it as one unit (a *monolithic* unit) is unreasonable and it must be **decomposed** into smaller ones. The smaller independent programs called modules interact with one another in simple, well-defined ways.

Generally the following principles help decomposition:

- Each sub problem is at the same level of detail.
- Each sub problem can be solved independently.
- Solutions to sub problems can be combined to solve the original problem.

Abstractions assist in making good choices for pieces of the decomposition. The following are some types of abstractions:

- **Procedural Abstraction** is when you break the problem into new operations (like static methods and instance variables)
- **Data Abstraction** is when you introduce new data types. (Classes)
- **Iteration Abstraction** is when you iterate over item sin a collection without considering details of how the items were obtained.
- **Type Hierarchy** is where you abstract away from individual types to families of related types.

Procedural Abstraction

Procedures combine two types of abstraction in a way that allows us to abstract a single action or task.

The first type of abstraction used is called *abstraction by parametrization* which is where you abstract from the identity of the data being used. This is done in the form of parameters into a method. When we use a parameter we are abstracting away from the actual value of the input, into variables. The actual value of the data is irrelevant, but the presence, count, and type of parameters are relevant.

The benefit of this type of parametrization is that a large number of computations can be described easily.

The second type of abstraction used is called *abstraction by specification* which is where the behaviour that the user can expect from the module is specified, and is abstracted away from implementing the behaviour.

What is done by the module is specified, how it is done is not.

Abstraction by specification has two advantages, locality (you can read the functionality of an implementation without having to actually evaluate the implementation) and modifiability (is a better implementation of a module becomes available you can change it without affecting how the program interacts with other modules since the specific implementation was not specified).

Informal specification includes the following three comments:

1. *REQUIRES:* this clause states any pre-conditions on the input.
2. *MODIFIES:* this clause identifies all modified inputs (passed by reference).
3. *EFFECTS:* this clause defines the output behaviour.

If the behaviour is not defined for some inputs (like negative numbers for example) we say that the module is a partial procedure, otherwise it is a total procedure. Partial procedures are less safe than total ones as the code my not behave as expected when incorrect parameters are entered.

In general, if the purpose of a procedure is difficult to state it should probably be decomposed more. If code is repeated multiple times in a project, it should probably become a procedure.

Procedures should be designed to be minimally constrained. Sometimes this means the output is undetermined (meaning more than one possible output), in such a case the implementation is deterministic (chooses one of the options).

We promote the generality (the classes out inputs the procedure can handle) and simplicity (well-defined and easily explained purpose) of procedures.

4 Week 4: Data Abstraction

In Java, *Data Abstraction* consists of a set of objects and a set of methods, implemented by a class or an interface.

When we use Data Abstraction we do not care about the representation of the object. The representation of an object is the exact way data is stored in its allocated memory, such as under different variable names or different primitive data types or structures. The user has access to the methods of the data abstraction which allows the user to interact with it without needed to know the representation begin used.

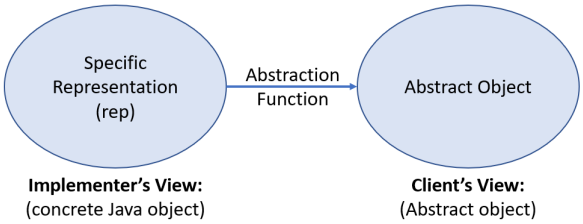
We specify a data abstraction by putting a comment at the beginning of the class:

```
class ClassName {
// OVERVIEW: brief description of the behaviour of the data type
.
.
.
}
```

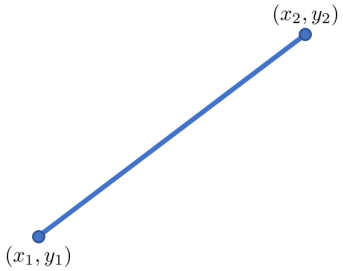
A good overview shows how the abstract object can be described in terms of well understood concepts. The overview clause must also state the mutability of the object.

Representations, Abstraction Functions, Rep Invariants

The representation (or *rep*) of a class is the method in which data is stored by the computer and used by methods in the class. The *abstraction function* is how you go from the rep to the abstract object which can be understood by a human.



For example, consider a line which goes from point (x1, y1) to a point (x2, y2), as in:



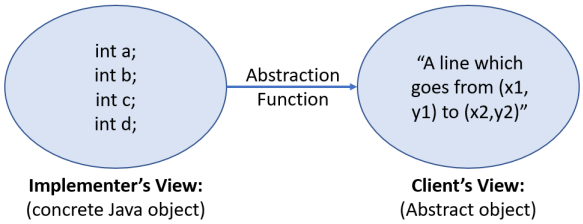
To make any line in this form we clearly need to store 4 numbers. The rep is the **way** we store those numbers. The following are two possible reps:

```
Rep 1:
class Line {
int a;
int b;
int c;
int d;

//constructor

//methods

}

Rep 2:
class Line {
int [] nums = new int[4];

//constructor

//methods

}
```

The **abstraction functions** depend on the rep. The abstraction function captures the designer's intent in choosing a particular rep, and it tells us what each piece of data (instance variables) represents in the actual abstract object being represented.

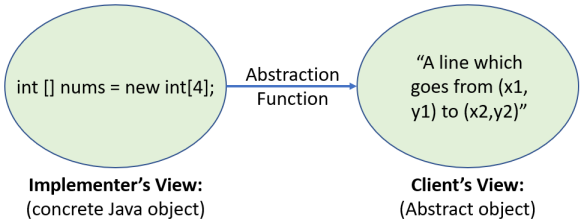
The abstraction function for **rep 1** could be:

```
// Line l = new Line();
// l.a represents x1, l.b represents x2
// l.c represents y1, l.d represents y2
```



The abstraction function for **rep 2** could be:

```
// Line l = new Line();
// l.nums[0] represents x1, l.nums[1] represents x2
// l.nums[2] represents y1, l.nums[3] represents y2
```



There is no one rep for an object, and there is no one abstraction function to go from the rep to the abstract object. The **rep invariant** is the common assumption made in all reps of a particular object. This is a property that all legitimate objects satisfy. In the previous example, the rep invariant could be that all reps store 4 integers.

As Functions

The **abstraction function** as a function is a function defined as:

$$AF(c) : C \rightarrow A$$

As in a function which takes in a concrete rep of an object (c), and maps that concrete rep C to an abstract object A.

In actual code this is done by the *toString();* method.

The **rep invariant** as a function is a function defined as:

$$RI(c) : C \rightarrow \{True, False\}$$

As in a function which takes in a concrete rep of an object (c), and maps that concrete rep C to a boolean value depending on whether the rep satisfies the rep invariant.

In actual code this is done by the *repOk();* method. Given some rep invariant, we can implement a method called *repOk()* which checks if the rep invariant is being honoured in the class.

Data Abstraction

A data abstraction as a whole can then be thought of as the following collection:

- A rep.
- Override equals();
- Override toString();
- Override clone();
- Implementation of repOk();

clone() Method

The signature of the clone method is:

```
protected Object clone() throws CloneNotSupportedException;
```

This method creates a new object which is a copy of the object which called it.

There are two types of copies with different use cases:

- **Shallow copy** for primitive or immutable instance variables.
- **Deep copy** for mutable instance variables.

To override the clone method our class needs to implement the *Cloneable* interface.

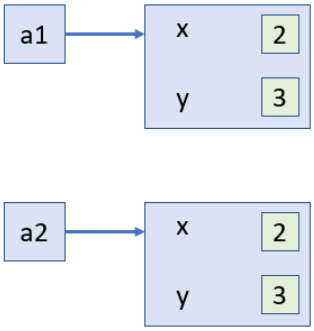
Following this, I will include 3 examples:

1. Shallow copy for a class Point.
2. Shallow copy for a class Circle (which is incorrect).
3. Deep copy for a class Circle (which is correct).

Example 1: Shallow Copy

```
class Point implement Cloneable {
private int x;
private int y;

//constructor here

// clone implemented as a shallow copy
@Override
public Point clone() throws CloneNotSupportedException{
return (Point)super.clone;
}

public static void main(String[] args) {
Point a1 = new Point(2,3);
Point a2 = a1.clone();
}
}
```

In this code we just tell the super (the Object class) to run its clone operation which allocates a new memory space for the instance variables **but** we want it to be downcast to a Point object. You can visualize the main as:



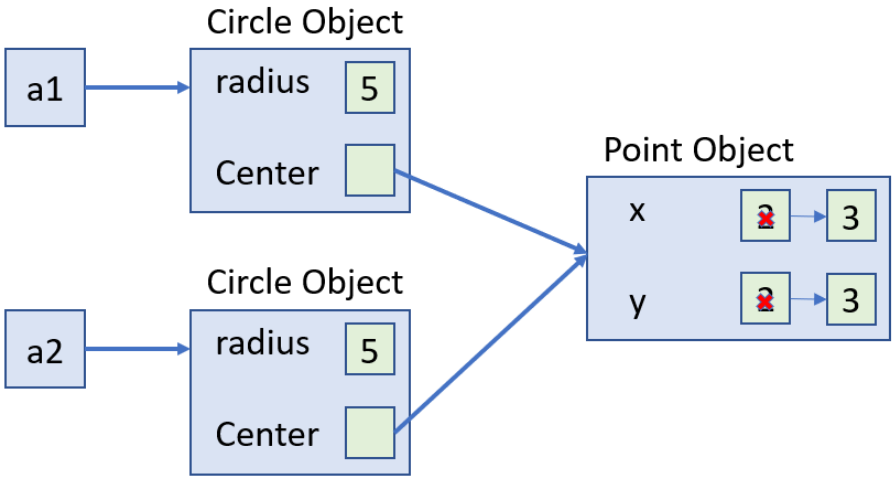
If you were to edit the x value of either point it would not affect the other point. This is because all of the instance variables of the class are either immutable or primitive (in this case all primitive).

Example 2: Why we need Deep Copy

If we try to do a shallow copy of a class with mutable reference instance variables, then we have an issue. Consider the point class from the previous example, and now look at the circle class:

```
public class Circle implements Cloneable {
private int radius;
private Point center;

// constructor here

// setters and getters here

// bad shallow copy
@Override
public Circle clone() throws CloneNotSupportedException{
return (Circle) super.clone();
}

public static void main(String[] args){
Circle a1 = new Circle(5, new Point(2,3));
Circle a2 = a1.clone();
a1.setCenter(3,3);
}
}
```

You can visualize what's happening in the main function as:



But notice that in changing the center point of the circle $c1$ we also changed the center point of circle $c2$! That's no good because they should not depend on each other like that. This happened because one of the instance variables of Circle was a mutable class. What we want is for the circle $c2$ to have it's own instance of a point object which can be changed independently. Notice that if Point were an immutable class then this would matter, as issues only come when you want to change the value of the Point object.

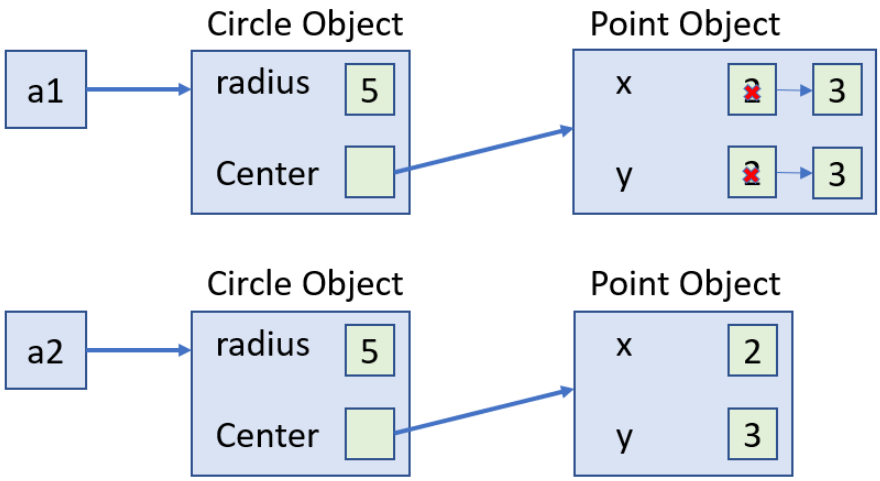
Example 3: Deep Copy

Let's instead to a **deep copy** of the instance of the Circle class. Once again recall the Point class from the first example, then:

```
public class Circle implements Cloneable {
private int radius;
private Point center;

// constructor here

// setters and getters here

// bad shallow copy
@Override
public Circle clone() throws CloneNotSupportedException{
Circle c = (Circle) super.clone();
c.center = (Point) center.clone();
return c;
}

public static void main(String[] args){
```

```
Circle a1 = new Circle(5, new Point(2,3));
Circle a2 = a1.clone();
a1.setCenter(3,3);
}
}
```

You can now visualize what's happening in the main function as:



Properties of Data Abstraction Implementations

A *benevolent side effect* is when an implementation modifies the rep without affecting the abstract object. This is only possible if the abstraction function is not one-to-one meaning many reps map to the same abstract object (which is very commonly true).

Exposing the rep is when the implementation provides users with access to mutable components of the rep. In actual code this is done by having non-private instance variables.

Design considerations for data abstractions:

1. **Mutability** is when a data abstraction has mutator methods. Immutable abstractions are safer than mutable ones.

2. The are four kinds of operations of data abstractions:

   - **Creator** operations produce new objects from scratch. (constructors)
   - **Producer** operations produce new objects given an existing object as an argument. (clone)
   - **Mutator** operations modify the state of their object. (setters)

- **Observer** operations provide information about the state of their object. (getters)

3. A data type is **adequate** if it provides sufficient methods for the client to use conveniently and efficiently.

5 Week 5: Modelling with UML

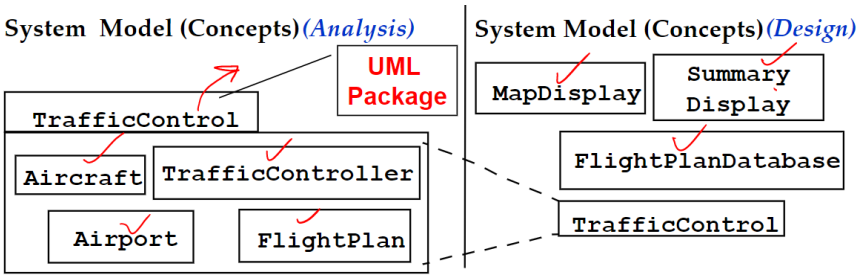
A model is an abstract representation of a system that enables us to analyse the system. A diagram of a model helps us visualize complicated systems.

System Domains

The **application domain** is the environment in which the system is operating in. These are the objects, devices, vehicles, people, and any other entity which will be applying the system.

The **solution domain** is the set of technologies used to build the system. These are the computers, sensors, helped methods and other technologies used to implement the model.

For example, consider the following set of classes:



Object-oriented analysis is concerned with modelling the application domain, while **object-oriented design** is concerned with modelling the solution domain.

What is UML?

UML (Unified Modelling Language) is a standardized notation for modelling software systems.

There are three components of the model of a system, the sum of which is the entire system model:

1. **Functional model** describes the functionality of the system from the user's perspective. (Use case diagrams)

2. **Object model** describes the structure of the system in terms of objects, attributes, and operations. (Class diagrams)

3. **Dynamic model** describes the internal behaviour of the system. (Sequence diagrams, statechart diagrams, activity diagrams)

There are 5 types of UML diagrams:

1. **Use case diagrams** describe the functional behaviour of the system as seen by the user.
2. **Class diagrams** describe the static structure of the system in terms of objects, attributes, and associations.
3. **Sequence diagrams** describe the dynamic behaviour between objects of the system.
4. **Statechart diagrams** describe the dynamic behaviour of an individual object.
5. **Activity diagrams** describe the dynamic behaviour of the workflow of a system.

We are concerned with the first 2 in this course.

UML Class Diagrams

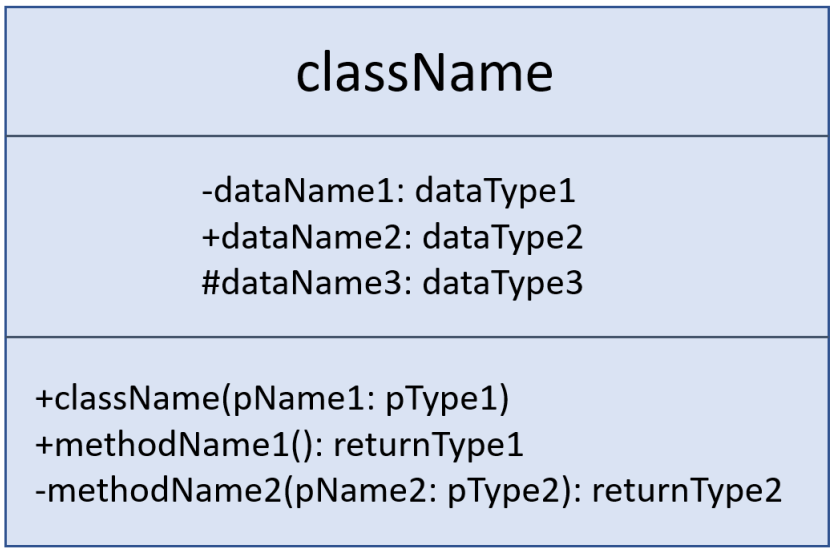
UML class diagrams are used to represent the structure of the system. This type of diagram shows each class, its instance variables, methods, and relationships among the classes with their multiplicities.

This type of diagram is used during requirements analysis to model the application domain. Also used during system design to model subsystems.

Let's use the following general code structure of a class:

```
public class className {
        private dataType1 dataName1;
        public dataType2 dataName2;
        protected dataType3 dataName3;

        public className(pType1 pName1) {}
        public returnType1 methodName1() {}
        private returnType2 methodName2(pType2 pName2) {}
}
```

We would draw this class as a UML class diagram as the following:



Note that we use − to denote private, + to denote public, and # to denote protected. Also, if the class is abstract, then we put the title in italics.

Go back to page 6 of this document to show how we denote relationships, and refer to the following table to see how we denote multiplicity of aggregation relationships:

Multiplicity	Option	Cardinality
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1 ✓	1 ✓	Exactly one instance
0..* ✓	* ✓	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

The only additional note needed is if you have an array declared as:

```
public dataType arr[] = new dataType[mult];
```

Then it would be denoted as the following in the UML class diagram (note the multiplicity follows the same rules as the table above):

```
+arr: dataType[mult]
```

6 Week 6: Design Patterns

We now begin our study of **design patterns**. Design patterns provide a vocabulary for understanding and discussing designs.

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without even doing it the same way twice.

A design pattern is a structure of a problem which has some solution which can be implemented in context. The are four essential elements to a design pattern:

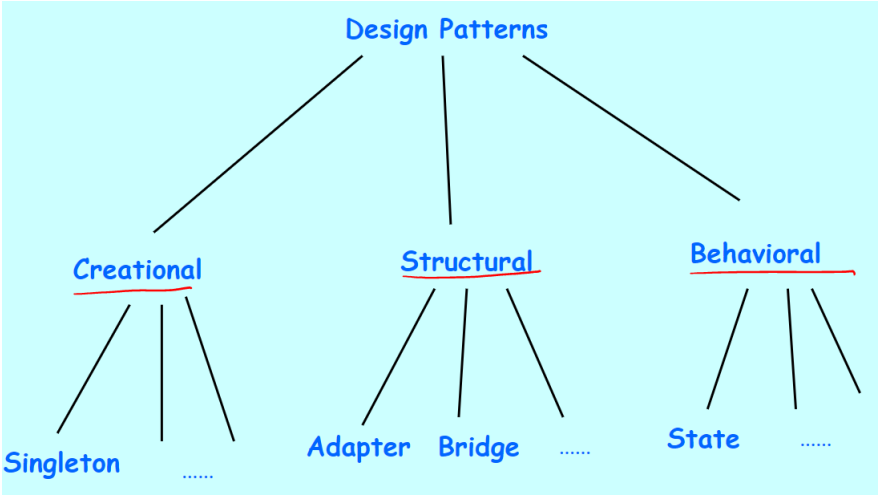
1. **Pattern name** is a handle we can use to identify a design problem.
2. **Problem** is what is being solved by the design pattern.
3. **Solution** is how the pattern solves the problem in context.
4. **Consequences** are the results and trade-offs of applying the pattern.

In order to describe a design pattern we need to know the following information:

1. Pattern name.
2. Intent (purpose of the pattern).
3. Motivating scenario in which the design pattern solves some problem.
4. Applicability of the design pattern.
5. Generic structure in the form of a UML class diagram.
6. The passes and/or objects participating in the design pattern, their responsibilities and collaborations.
7. The trade-offs and results of using the pattern.
8. How the pattern can be implemented.

We use design patterns to identify solutions to problems by looking at the abstract problem, instead of the problem in context. This lets use reuse code from other solutions to the same abstract problem. This lets us get a higher level perspective on the problem and on the process of design, as well as improving modifiability and maintainability of code.

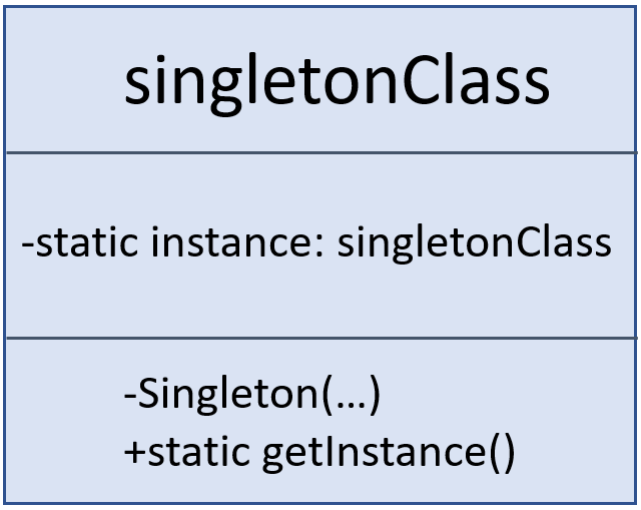
There is the following classification of design patterns, along with a few specific design patters we will soon study:



Design Pattern: Singleton

The singleton design pattern is classified as *creational*.

- **Intent:** The intent is to ensure a class has only one instance and provide a global point of access to it.
- **Motivation:** It is important for some classes to only have one instance. For example if you have a class responsible for handling the window of a game, you certainly don't want multiple windows popping up.
- **Applicability:** anywhere where we want to enforce only once instance of a class can exist.
- **Generic Structure**:



- **Participants and Collaborations**: client classes create an instance of the singleton through the static getInstance method.
- **Consequences**: results include controlled access to the sole instance, and the pattern can be extended to a variable number of permitted instances easily.
- **Implementation:** There are three important points to implementing the Singleton design pattern:
  1. A private static member of the class that refers to the desired object.
  2. A public static method that instantiates this class if this member is null and then returns the value of this member.
  3. A private or protected constructor so that no other class can directly instantiate the class.

You can implement the *getInstance()* method in the following way (pseudocode):

```
if instance == null
        instance = new Singleton(...)
return instance;
```

What this does it to create the first and only instance of the class, you call the getInstance method, but then to access the class you use the same method but instead it will just return to you the already created instance. This makes it impossible to create another instance of the class. The following is actual java code implementation of a singleton class:

```
public class Singleton {
        private static Singleton instance;
```

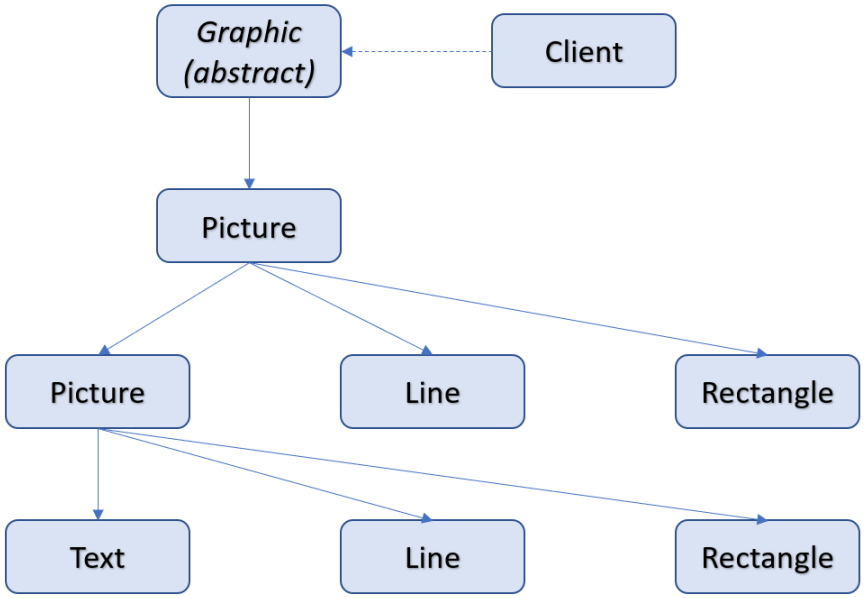
```
        private Singleton () {}
        public static Singleton getInstance() {
                if (instance == null) {
                        instance = new Singleton();
                }
                return instance;
        }

        // after this you would put the instance data and methods.
}
```

Design Pattern: Composite

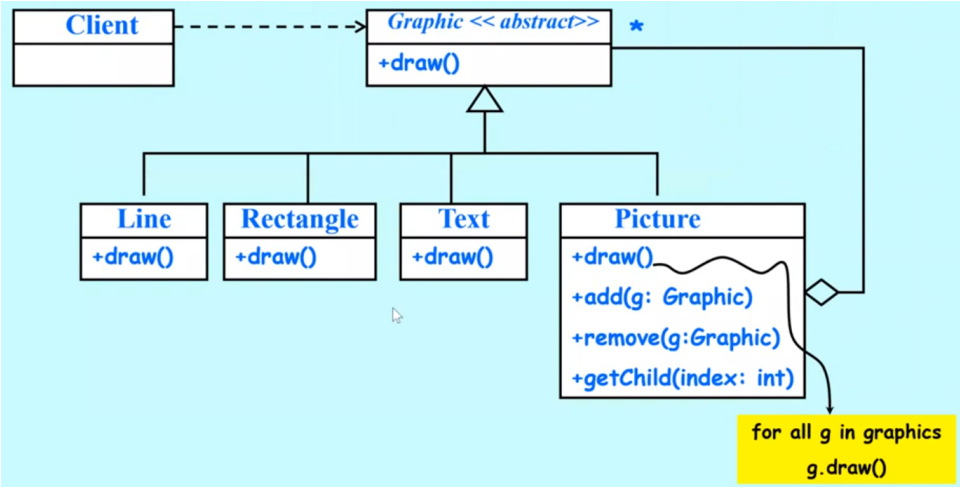
Constructional design pattern.

- **Intent:** The intent is to compose objects into a tree structure to represents part-whole relationships. The composite pattern lets clients treat individual objects and compositions of objects uniformly.
- **Motivation:** For example if you have a folder which can contain files and other folders, you want to be able to rename folders just like you can rename files. Additionally you want to be able to ask the size of a file the same way you can ask the size of a folder.
- **Applicability:** The applicability of this pattern is when you want the client to be able to treat compositions and individual objects uniformly.
- **Participants and Collaborations**: These classes include the **Component, Leaf, Composite**, and of course **Client**. (Clients use the component class to interact with the structure. If the uniform method is called on the leaf then the concrete implementation is called, otherwise if the uniform method is called on a composite, then each element of that composite has its uniform method called).
- **Consequences:**
  - Easy to add new components.
  - Simple to use for the client.
  - Simple structure of hierarchies.
  - Can make design overly generalized.

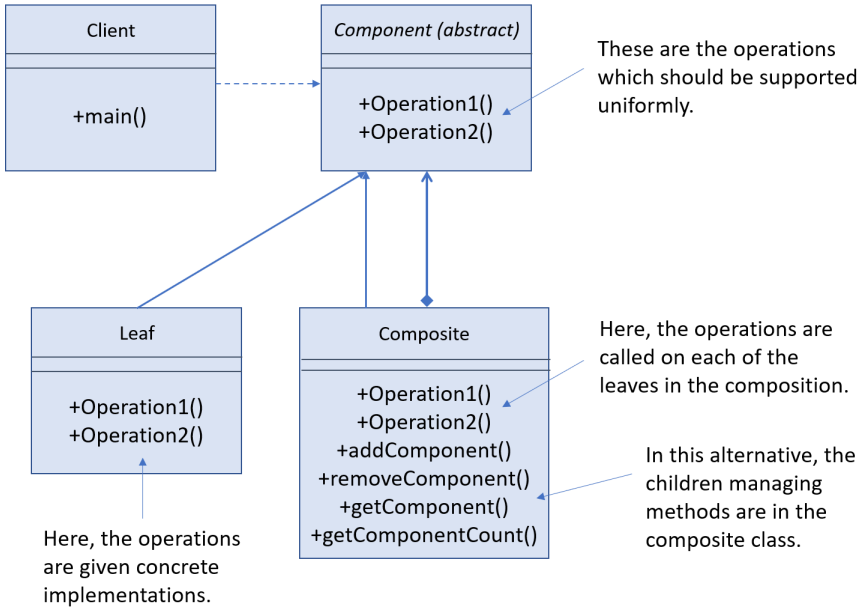
Let's say you are making a graphics editing software. You would define **primitive classes** which are actual objects which can be drawn like **Text, Line, Rectangle** and then you would define **composite classes** which may be called **Picture** which can contain multiple primitive classes, or other Pictures. You may then create a **Graphic** abstract class which would be what the client interacts with which controls the graphic being created as a whole, instead of individual pictures of primitives. The following image illustrates this:



From here, we can say that **draw** is the method which wants to be treated uniformly between composites (Picture class) and primitives (Text, Line, Rectangle classes). In the abstract class we have an abstract method draw() which must be overridden concretely by th primitives. If you call draw() on a composite, then draw() is called on all the elements within that composite.



Generic Structure 1

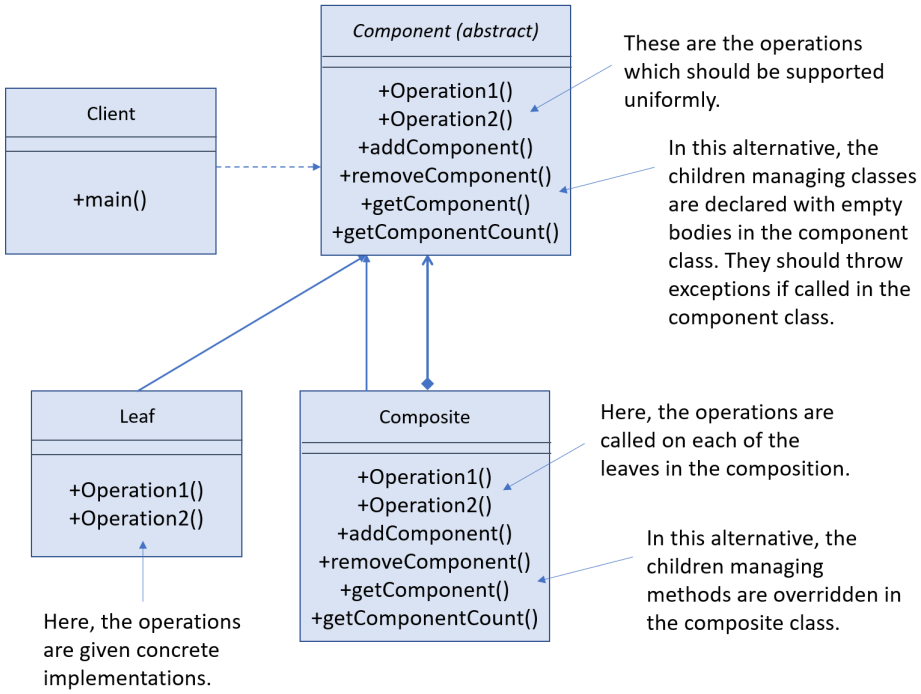


- This is a safe alternative.
- Does not exactly treat the leaf and the composite exactly the same.

Here is an example of the main file:

```
Component c1 = new Leaf();
Component c2 = new Leaf();
Component c3 = new Composite();
((Composite)c3).addComponent(c2);
((Composite)c3).addComponent(c1);
```

Generic Structure 2



- This is an unsafe alternative. If the client were to try and run the leaf managing method on a leaf an exception would be thrown.
- Does treat the leaf and the composite exactly the same.

Here is an example of the main file:

```
Component c1 = new Leaf();
Component c2 = new Leaf();
Component c3 = new Composite();
c3.addComponent(c2);
c3.addComponent(c1);
```

Notes on implementation:

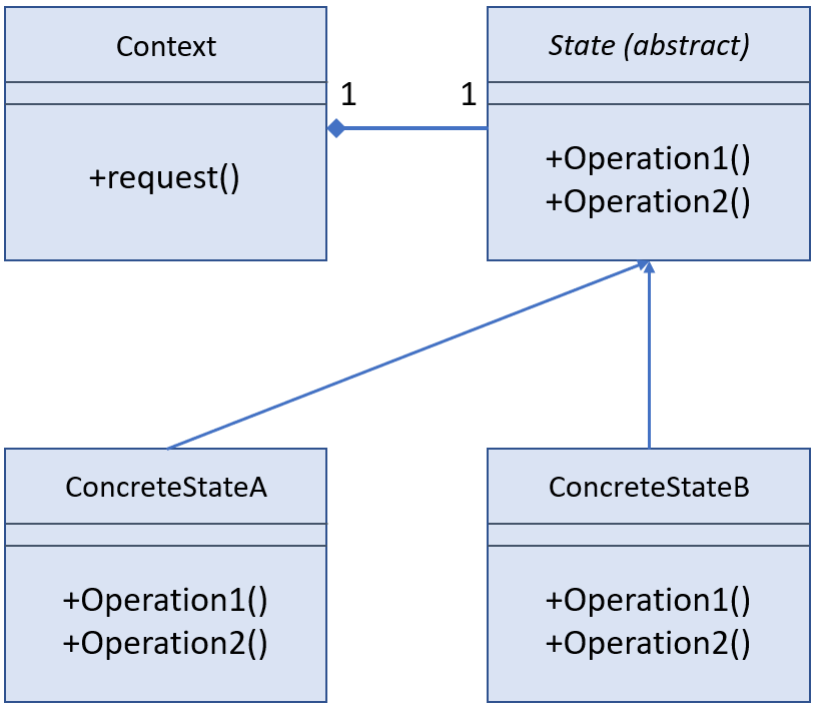
- Leaf extends component.
- Leaf gives concrete implementation of the uniform methods.
- Composite extends component.
- Composite contains an arrayList of Components.
- Composite's implementation of the uniform method is just to call the method for all the elements in the arrayList.

7 Week 7: Design Patterns

Design Pattern: State

- **Intent:** The intent is to allow an object to alter its behaviour (implementation of methods) when its internal state changes. The object will appear to change to a different class.
- **Motivation:** Sometimes there is an object that can be in one of several states, and long if-else chains get cumbersome. The state of an object is the exact condition of an object at a given time, based on instance variables.
- **Applicability:** This design pattern is used when an object behaviour depends on its state, and must change its behaviour at runtime. This design pattern is also useful when methods have large, multipart conditional statements.
- **Participants and Collaborators:** Context, State, some number of ConcreteState subclasses. Context delegates state-specific requests to the current concreteState. Context is the primary interfacer for clients.
- **Consequences:**
  - New states and transitions can be added easily by defining new subclasses.
  - Localizes behaviour of a state into an object.
  - Allows state transitional logic to be part of state object instead of conditional statements.
  - More objects used.

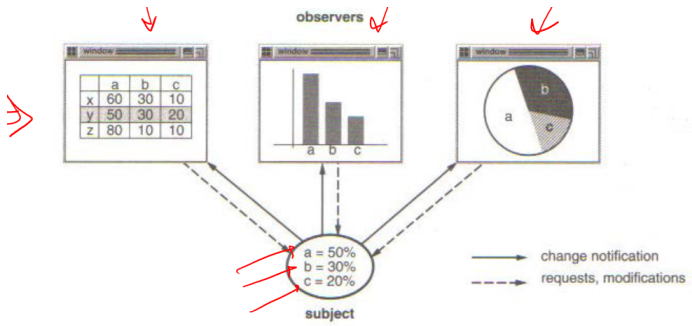
The following is the general structure of the pattern:



- The Context class contains a variable of type *State*. This variable gets assigned to different children of the *State* class depending on what state the object should be in.
- *State* is an abstract class, which has abstract methods.
- The operations in the abstract class need to be overridden in the concrete classes.

Design Pattern: Observer

- **Intent:** The intent is to define a one-to-many dependency between objects so that when the one object changes its state, all its dependants are notified and updated automatically.
- **Motivation:** An example could be an object containing data, which has many *observers* being objects which create graphs from that data.

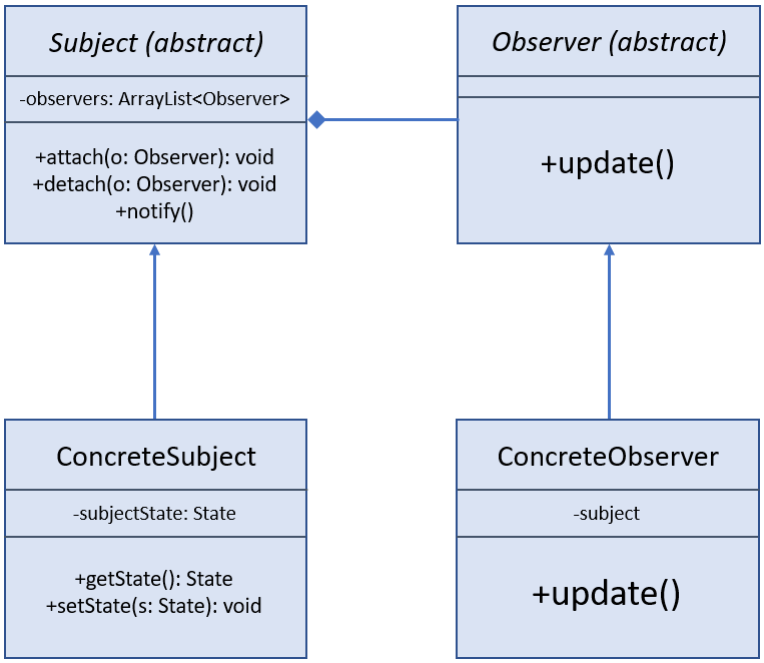


When the data is updated, we want the graph making objects to be notified, and updated.

- **Applicability:**
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others and you don't know how many objects need to be changed.
  - When an object should be able to notify the objects without making assumptions about who these objects are.
- **Participants:**
  - **Subject** provides an interface for attaching and detaching Observer objects. This makes it so the concrete object doesn't have to deal with implementation of this.
  - **Observer** defines an updating interface/abstract class for objects that should be notified (which will inherit).
- **Consequences:**
  - Can reuse subjects without reusing their observes, and vice-versa.
  - Lets you add observers without modifying the subject or other observers.
  - Minimal coupling between subject and observers.
  - Support for broadcast communication.
  - Unexpected updates.
- **Implementation:** To implement the design pattern, have Observers that attach themselves to a Subject that watches for a change in state to occur. Subject tells the observers the the change occurs.

The general outline of how this works is based off of the **Subject** class and the **Observer** class. A subject class may have any number of dependent observers, all of which are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state data.

The following is the general structure of the design pattern:



The pseudocode for the *notify()* method is:

```
for all o in observers
        0.update();
```

- notify() is called whenever the state of the concrete subject is changed.
- update() is overridden in the concrete observer classes.

8 Week 8: UML Use-Case Diagrams

Recall that there are three components to the system model of a software system:

- Functional model (Use-case diagrams)
- Object model (Class diagrams)
- Dynamic model (Sequence diagrams)

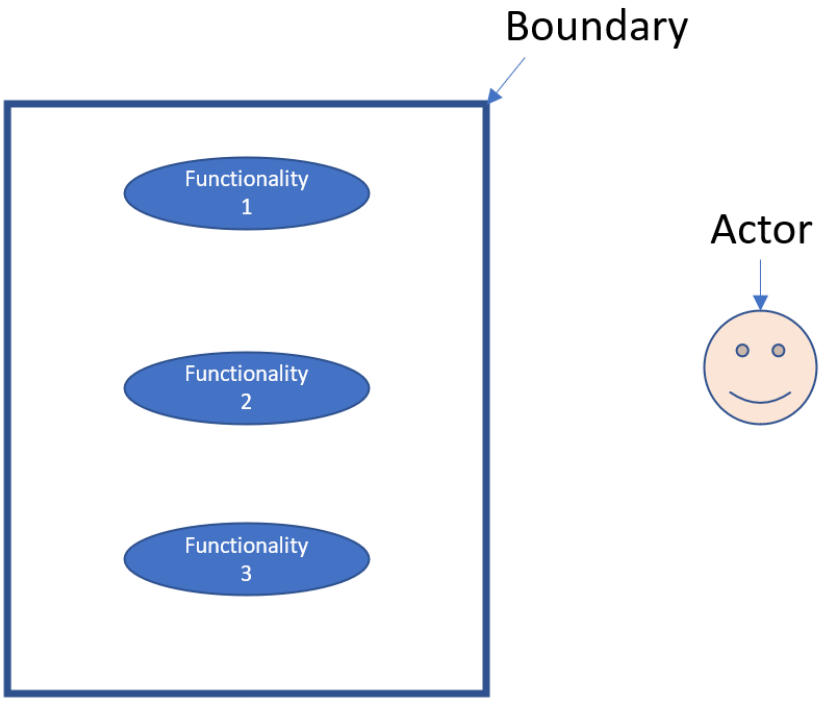
In this section we are concerned with building the functional model of a software system, defined by a use case diagram. Use case diagrams describe the functional behavior of the system as seen by the user. Use case diagrams are typically used in the requirements elicitation stage of development to communicate how different users will interact with the software, and what they can each have access to. These diagrams define the boundary of the system, as in what operations are the systems own responsibility to conduct.

Typical elements of a UML use case diagram include:

- **Actors:** the different generalizations of types of users which will use the software. Actors represent **roles**, as in a type of user of the system. Actors can be users, an external system, or sensors from the physical environment. An actor is given a role name, and an optional description.
- **Use Case:** the different operations or uses that the software system has. A use case represents a class of functionality or behavior as seen from an actor.
- **Boundary:** a literal box which encompasses all of the functionality of the system.
- **Relationships:** arrows and lines which indicate the relationships between actors, and functionality.

The use case model of the software system is the set of all use cases that completely describe the functionality of the system.

Actors are drawn outside of the boundary of the system, and functionality is drawn as circles inside the boundary of the system. For example the following image shows the general layout of a UML use case diagram, without any relationships:



This software described above is useless! The actor has no way of interacting with the software. This is why we need to define relationships between the actors and one or more functionalities.

Use Cases

As described above, use cases are a way to communicate the functionality and behavior of a software system. A use case is an abstraction that describes all possible scenarios involving the described functionality. Use cases consist of 7 descriptive elements:

1. A **unique name**.
2. **Participating actors** which is a list of the actors which interact with the use case.
3. **Flow of events** which shows which use cases are related to this use case (more on these relationships later). You should use the active voic while describing, saying what the actor or the system will do. The causal relationships should be clear, and the boundaries of the system should be stated.
4. **Entry conditions**. Conditions which initiate the use case.
5. **Exit conditions**. Conditions which end the use case.

6. **Exceptions**. Special cases of the use case which only happen given certain situations.

7. **Special requirements**. Extra requirements not directly stated by the functionality, for example a time limit on the response time.

Scenario-Based Requirements Elicitation

If you want to understand what a software system does, before it is developed, you can use a scenario based requirements elicitation approach. This approach has five components:

1. Identify actors.

2. Identify scenarios.

3. Identify use cases.

4. Refine use cases and identify relationships among use cases.

5. Identify non-functional requirements.

A **scenario** is an example of a real life story which would use the software system. In these scenarios we give the actors real names, and imagine a real situation. The focus of scenarios are understand ability. A use case is a generalization of a class of scenarios.

The following are some common questions you may ask a client to better understand the user case of the software system:

- What are the primary tasks that the system needs to perform?

- What data will the actor create, store, change, remove, or add in the system?

- What external changes does the system need to react to?

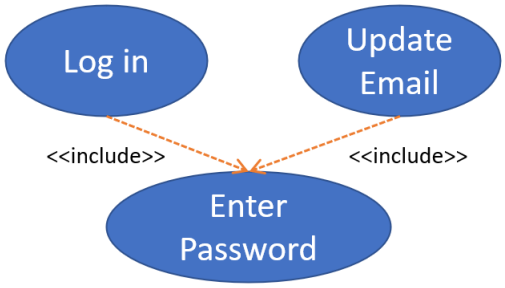
- What changes or events will the actor of the system need to be informed about?

After you have identified the scenarios, identify the formal use cases based on those scenarios. For example if many scenarios require a user to "Report emergency" then that should be a formal use case of the system. Describe all use cases using the seven elements we listed earlier.

Use Case Relationships

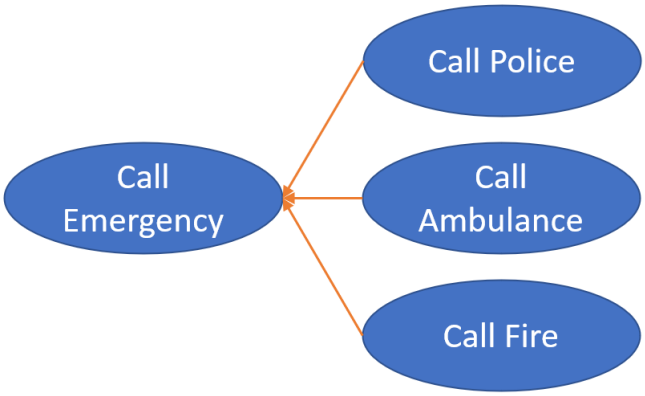
Dependencies between use cases are represented by use case relationships. There are three types of use case relationships:

1. Includes. If different use cases execute the same sub-use case, then we can say that the use case *includes* the sub-use case. For example, if you have three use cases: log in, update email, enter password, then the log in use case, and update email use case both include the enter password use case. We use a dotted open arrow pointing towards the common use case with the relationship labeled.

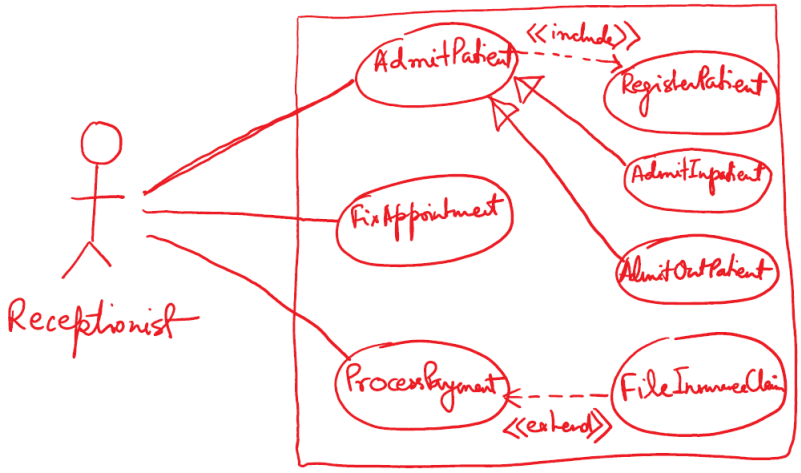


2. Extends. If a use case has to execute special actions in an exceptional situation, then we say those special actions are a use case that *extends* the initial use case. For example if you have a system that takes some physical input and then *Reports* it as a use case, you may have a *ConnectionDown* use case which extends *Report* with instructions on what to do if the connection is down. We use a dotted open arrow pointing towards the use case being extended with the relationship labeled.

3. Generalization (similar to inheritance). In this type of relationship, a *parent* use case can be achieved be completing one of its *child* use cases. These children use cases are also call possible *realizations*. For example, if you have a use case being *CallEmergency* then you can have three possible realizations being *CallPolice*, *CallAmbulance*, and *CallFire*. We use a solid arrow with a solid head pointing towards the parent without the relationship labeled.



The following is a full example of a use case diagram of a receptionist:



9 Week 9: Design Patterns

This week we exclusively learned about three design patterns:

- Adapter.
- Strategy.
- Facade.

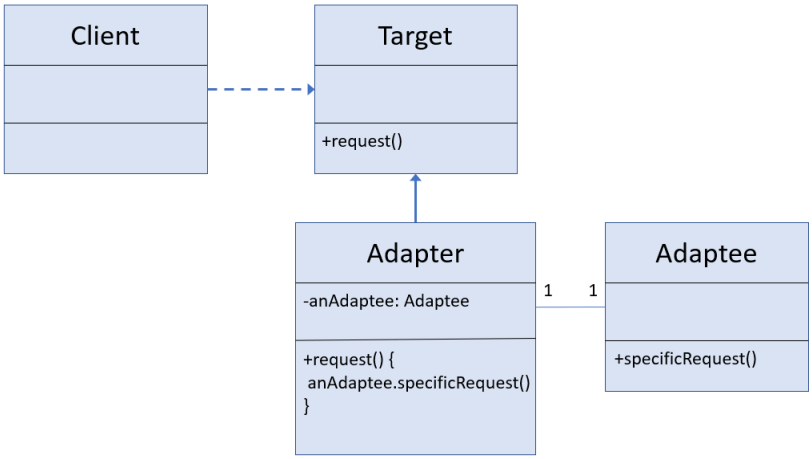
Design Pattern: Adapter

Adapter is a structural design pattern.

The main idea of this pattern is if you want to reuse some code that someone else made, but the names of the API (method names, instance variable names) do not match, even if the function of the method is the same (for example in your software you may call "addUp()" while the API uses "sum()"). You need a class in between your code and the outsourced code to call the function in the right way, in a way we need an *Adapter* between our software and the outsourced ones.

- **Intent:** The intent of this design pattern is to convert the interface of a class into another interface clients expect. Adapters let classes work together that couldn't otherwise because of incompatible APIs.
- **Motivation:** sometimes a class that's designed for reuse is not reusable only because of differences in the interface naming.
- **Applicability:** You can use this design pattern when you want to use an existing class, and its interface does not match with the one you need. You can also do this when you want to create a reusable class that cooperates with unrelated classes with other APIs.
- **Participants:**
  - **Target** is what runs the main software, and uses the adapter to interact with outsourced code.
  - **Client** interacts with the target.
  - **Adaptee** is the outsourced code which needs to be adapted before use.
  - **Adapter** is the class which redirects the function calls from *Target* to *Adaptee* using the correct API.
- **Consequences:** the only consequence is that preexisting code can fit into new classes without being limited by their interfaces.
- **Implementation:**
  - Contain the existing class in another class.
  - Have the containing class match the required interface and call the methods of the contained class.

The following is the generic structure of the *Adapter* design pattern:



Design Pattern: Strategy

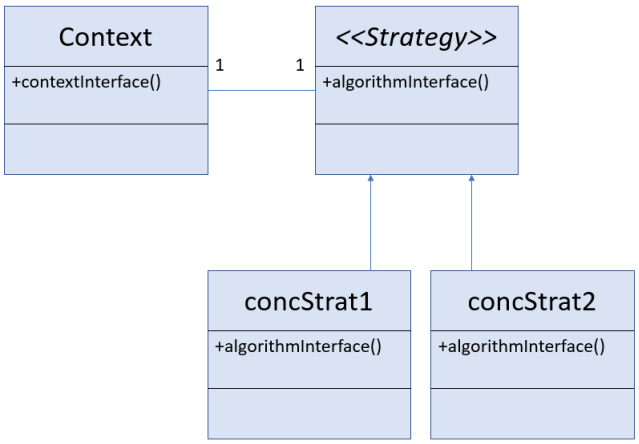
The strategy is categorized as a behavioral pattern.

The idea of this design pattern is **very** similar to the state design pattern. In this case we call them **strategies** instead of *states* and they describe **ways to implement a method** instead of **behaviors**.

- **Intent:** Define a family of algorithms, encapsulate each in a class and make them interchangeable. Strategy design pattern lets the algorithm vary independently from the clients that use it.
- **Motivation:** We could use this for example in the use of a sort() method. The different sort algorithms can be encapsulated into each their own classes, then the main class can can switch between which sort algorithm actually is being used to implement the sort() method.
- **Applicability:**
  - many related classes differ only by their behavior.
  - You need different variants of an algorithm to be used within a class.
  - A class defines many behaviors and otherwise you would use many conditional statements.
- **Participants:**
  - Context is the class which contains the main file. It contains some function call to the abstract Strategy class. Using polymorphism we can make it implement the correct algorithm.
  - The Strategy class is an abstract class which the concrete strategies inherit from.
  - Any number of concrete strategies which implement the algorithm.

- **Consequences:**
  - Provides an alternative to subclassing the Context to get a variety of behaviors.
  - Eliminates large conditional statements.
  - Provides a choice of implementations for the same behavior.
  - A disadvantage is that it has an increased number of objects.

The following is the general structure for the *Strategy* design pattern:



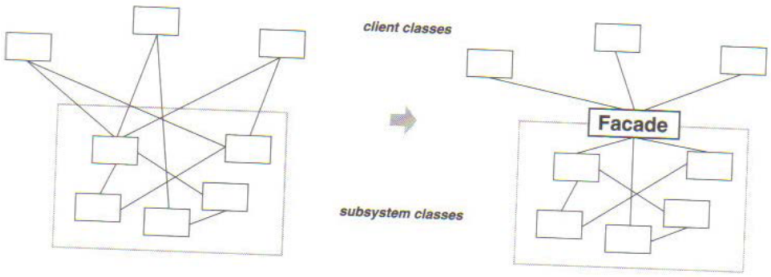
In the context class you have an instance variable of type Strategy which can be assigned to concrete strategies at runtime. The contextInterface() call call the algorithmInterface() method of the strategy object.

Design Pattern: Facade

This design pattern is classified as *Structural*.

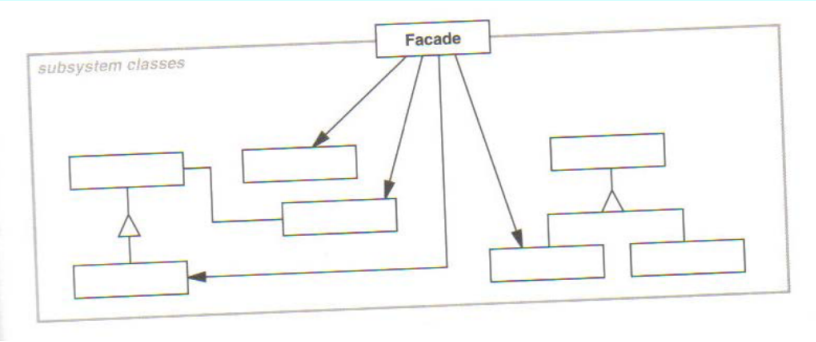
The idea of this pattern is if you have some complicated subsystem within your software system, then you may want to do some action which requires a bunch of function calls at once in a reliable pattern. In this case we can just make a new function which calls all of these methods so the user does not need to know the inner workings. We say that that function is a *Facade* to the complex subsystem.

- **Intent:** The intent of this design pattern is to provide a unified interface to a set of interfaces in a subsystem. The facade defines a higher-level interface that makes the subsystem easier to use.
- **Motivation:** we want to simplify how we use an existing system.



- **Applicability:**
  - You want to provide a simple interface to a complex subsystem.
  - You want to layer your subsystems. The facade defines an entry point to each subsystem level.
  - You want to decouple classes of a subsystem from clients and other subsystems, promoting system independence and portability.
- **Participants:**
  - The facade class is what is communicated with by the client or other subsystems, and which calls the functions within the complex subsystem. The facade may have to do computation on its own with private helper methods to properly call the functions within the system.
  - The subsystem classes.
- **Consequences:**
  - Shields client from subsystem complexities.
  - Promotes weak coupling, allowing you to change classes that comprise the subsystem without affecting clients.
  - It does not prevent sophisticated clients from accessing the underlying subsystem.

The general structure of the *Facade* design pattern is below:



10 Week 10: Design Patterns

This week we learned about three new design patterns:

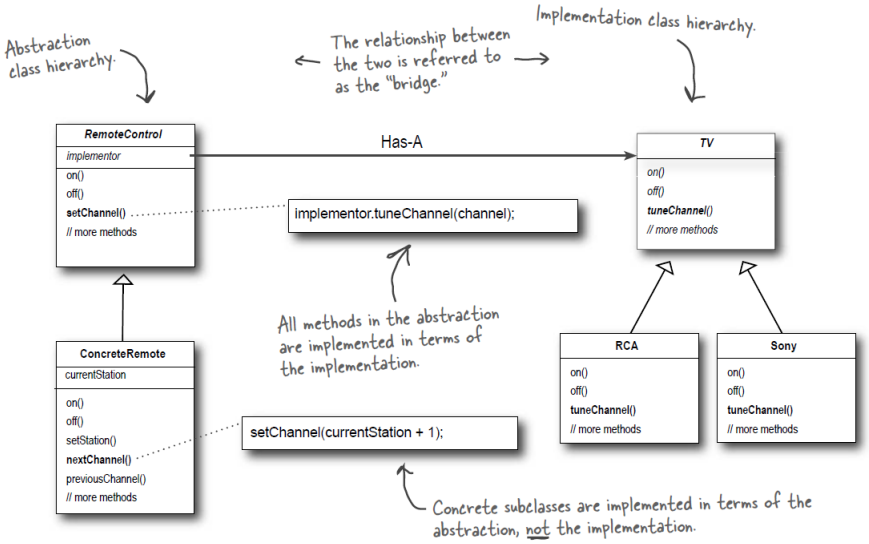
- Bridge Pattern.
- Factory Method Pattern.
- Abstract Factory Pattern.

Design Pattern: Bridge

This design pattern is classified as *structural*.

Given the following scenario: You are the developer for a TV company who is testing many different remote controls with different TVs, and they should all work together. You may create an abstract class *Remote Control* which would then have subclasses for all the types of remotes you would need, but then you **also** need an abstract class *TVSet* which would each have a slightly different implementation of *setChannel(channel)* for example. You want to make a piece of software that supports all of the remotes and TVs, so you need to be able both the *abstraction* (different remotes) and *implementation* (different TVs) independently.

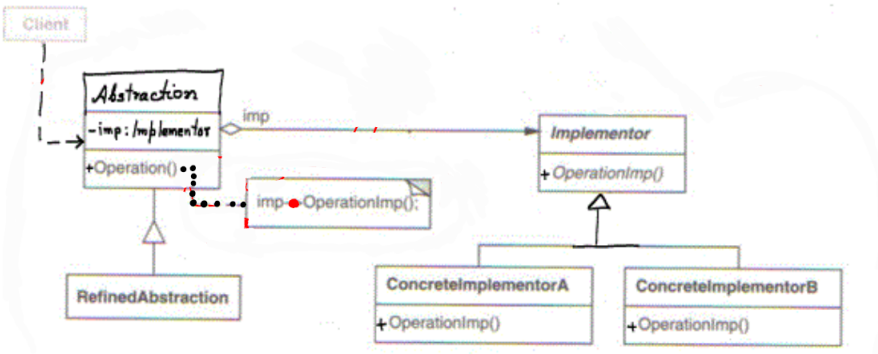
The following class diagram shows the TV problem solved using this pattern:



It's kind of like a state design pattern and a strategy design pattern in one, and both ways.

- **Intent:** The intent of this design pattern is to decouple an abstraction from its implementation so that the two can very independently.

- **Motivation:** Some intuitive solutions to this problem require 3+ levels of abstraction (inheritance levels) which against OOP design principles.
- **Applicability:**
  - Use when you want to avoid permanent binding between abstraction and implementation. This means when you don't want a specific "remote" bound to a specific "TV".
  - Both abstractions and implementations should be extensible by subclassing.
- **General Structure:**



- **Participants:**
  - **Abstraction** maintains a reference to an object of type Implementor. This class forwards client requests to Implementor.
  - **ConcreteAbstraction** is an actual version of the abstraction, and is a subclass.
  - **Implementor** defines the methods for implementation of what the abstraction wants to do.
  - **ConcreteImplementor** is an actual version of the implementation, and is a subclass.
- **Benefits:** Abstraction and implementation can be extended independently as they are decoupled.
- **Implementation:** Encapsulate the implementations into an abstract class. Contain a reference to the implementation in the base class of the abstraction.

Design Pattern: Factory Method

This method is categorised as *creational*.

The idea of this pattern is if you are in a class and want to create an instance of another class, but you **don't** want to specify which one, in fact you want to leave it up to your own subclasses to "decide" which new class to make.

To be clear, the subclasses do no "deciding" (the subclass is chosen by the Client) but the phrasing is used because the "Creator" class has no knowledge of which other class is being created (this will make more sense once you see some examples).

- **Intent**: Define an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Motivation:** This design pattern is useful when you are making an *interface* (not literally a Java Interface, but a set of code which allows other code to interace). For example if you have an *Application* class which is meant to create *Document* classes. The *Document* that *Application* creates depends on what concrete application is trying to make a document (Word would make a text file, Audio editors would make an .mp3 file, etc.) The *Application* class is responsible for managing and creating documents at the request of one of it's child classes. *Application* should be made in such a way that it does not need to know which exact concrete application is calling to make a new *Document*.
- **Applicability:**
  - Use when a class cannot anticipate the class of objects it must create.
  - A class wants its subclass to specify the objects it creates.
- **General Structure:**

Diagram 1:

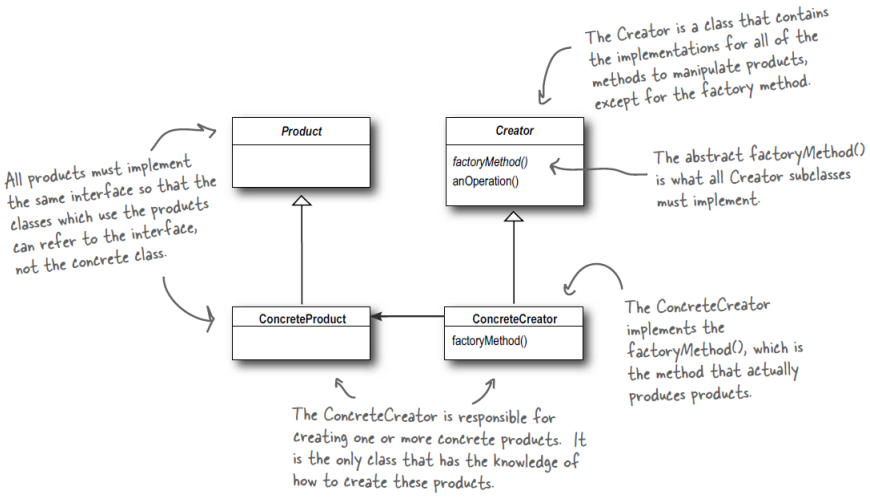
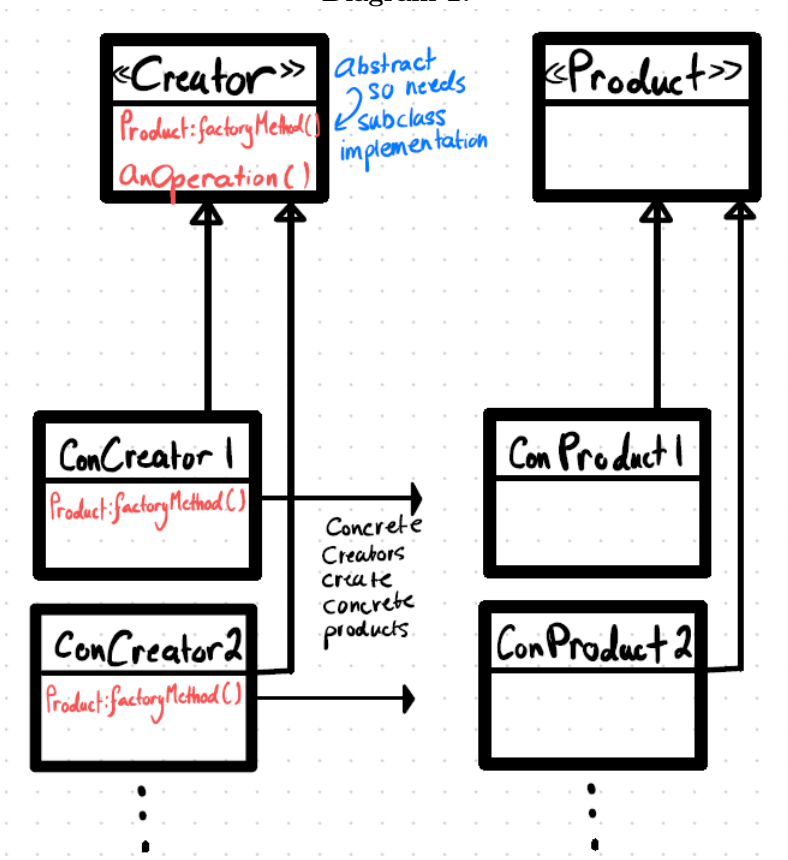

The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.
Product
Creator
factoryMethod()
anOperation()
The abstract factoryMethod() is what all Creator subclasses must implement.
All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.
ConcreteProduct
ConcreteCreator
factoryMethod()
The ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.
The ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

Diagram 2:


«Creator»
Product:factoryMethod()
anOperation()
abstract so needs subclass implementation
«Product»
ConCreator 1
Product:factoryMethod()
ConCreator2
Product:factoryMethod()
Concrete Creators create concrete products
ConProduct1
ConProduct2

A few notes:

  - The *FactoryMethod* is the method which handles the creation of new Products.
  - The factory method is abstract in the Creator class.
  - AnOperation() can have contextual meaning in the implementation, however it must at some point call FactoryMethod.
  - The concrete factory method implementations return the correct type of *ConcreteProduct*.

- **Participants:**
  - **Creator** declares the factory method. Relies on its subclasses to define the factory method so that it returns an instance of the appropriate *ConcreteProduct*.
  - **ConcreteCreator** overrides the factory method.
  - **Product** is the interface of the objects of the factory.
  - **ConcreteProduct** implements the Product interface.

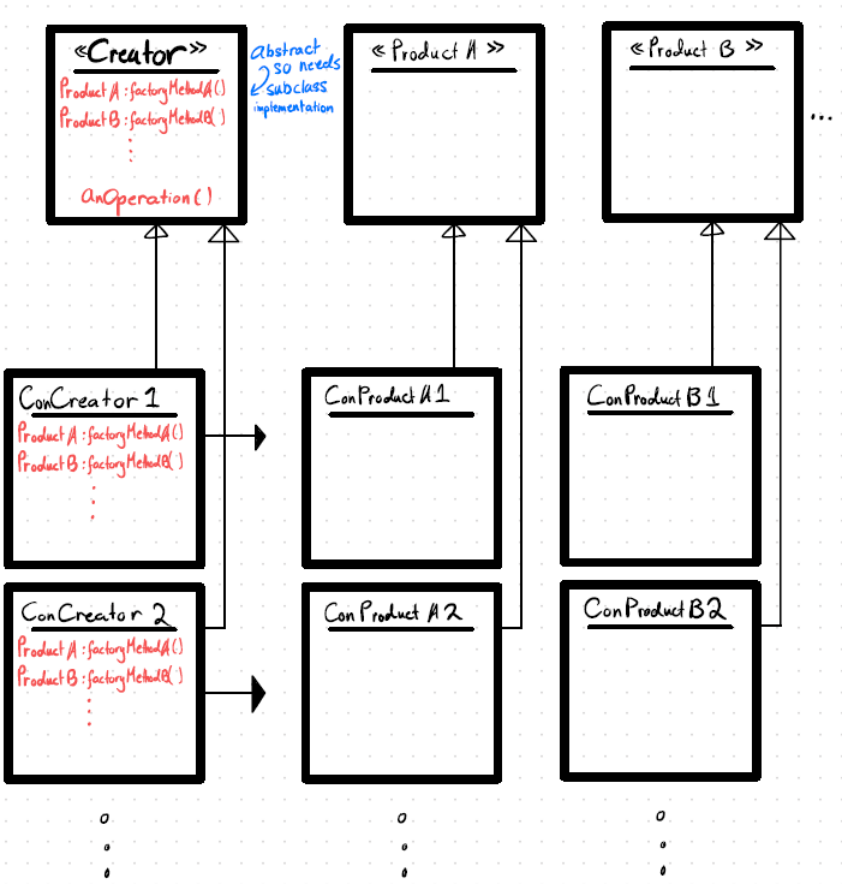
Design Pattern: Abstract Factory

This method is categorised as *creational*.

The idea of the abstract factory pattern is you take the Factory Method pattern, but now your *Creator* class does not just make a single *Product* at a time, but instead it creates *families of related classes*. A family of related classes are classes that contextually work together. For example let's say you are making a button (which contains Functionality and Icon) with text, on three operating systems: Windows, Mac, and Linux. You would want the Creator to create WindowsFunctionality and WindowsIcon when the ConcreteCreator is Windows as opposed to come combination of Functionality and Icon for Windows, Mac, and Linux.

Your creator class now can create multiple products potentially of different types.

- **Intent:** Provide an interface for creating families of related or independent objects without specifying their concrete classes.
- **General Structure:**



A few notes:

- The abstract Creator class is sometimes called *AbstractFactory*.
- The abstract Creator has different factory methods for the different Products in the family of classes being created.
- Each factory method can be overridden by any of the concrete factory classes.
- An abstract factory can have any number of product classes.
- You can think of Product A, B, and so on as categories of products.

- **Applicability:**
  - When a system should be independent of how its products are created.
  - When a systems should be configured with one of multiple families of objects.

  - When a family of related products is designated to be used together.
  - When you want to provide a class library of products, and you want o just reveal their interfaces.

- **Participants:**
  - Same as **Factory Method** with few exceptions.
  - **AbstractFactory** is the *Creator* class for this design pattern. It defines the interface for how to create each member of the family of objects. Each family member is creates by having a unique **ConcreteFactory**.

- **Benefits:**
  - Isolates concrete classes.
  - Exchanging product families easy.
  - One downside is supporting new products could require editing tested code, but adding new families of Products is easy.

- **Implementation:**
  - Define an abstract factory class that specifies which objects are to be created.
  - Implement one concrete class for each family.

11 Week 11: Requirement Elicitation + Analysis, System Design and Object Design

12 Week 12: Testing and Debugging