Programming Paradigms

Procedural: C, Assembly **Object-Oriented:** C#, C++, Python, Java, Scala **Functional:** Haskell, Elm, Scala **Logical:** Prolog

CS vs SE

CS: More theory focused, Algorithms

SE: Practical / Implementation, Design, Regulated by CEAB, not taken too seriously by PEO

PEO: Professional Engineers of Ontario, Licensing and accreditation for Engineering
4 years of practical experience (3 with an accredited degree), law / ethics exam, yearly fees
Regulation, enforcement, discipline
Has legal authority to fine companies misusing the term engineer
Iron Ring just means you graduated, pretty useless, get it at the Kipling ceremony

CEAB: Canadian Engineering Accreditation Board, determines if an engineering program gets approved.

SDLC

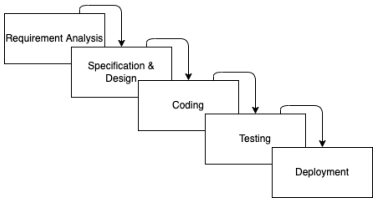
- Requirement Analysis
  - Getting input from stakeholders (customers, salespeople, industry experts, programmers.)
  - What the software does
  - Software Requirements Specification (SRS)
    - Document describing what the software is, what it does, etc.
  - Functional Requirements
    - What the software actually does
    - Must be:
      - Atomic: Each requirement should cover exactly one function of the software
      - Precise: Requirements should not be ambiguous
      - Verifiable: Must be testable to determine whether the requirement is actually met
    - Things you can **prove** using logic / discrete math
    - "The software shall..."
  - Non-functional Requirements
    - Usability, performance, security
- Specification and Design
  - Determining how the software will meet requirements specified in SRS document.
  - Modules, classes & objects, packages
  - Libraries & APIs
  - Class Diagrams
  - What the code does, what should functions output given specific inputs
- Coding
  - How specification is implemented
- Deployment and Maintenance

Contract between client and developers

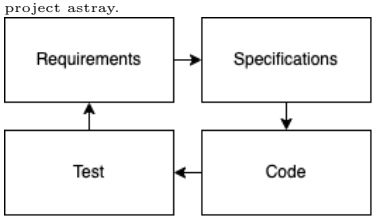
Scope: Important for clients, allows for specifying what should be included. **Out of scope:** Important for devs, allows specifying what the program will not be responsible for, preventing feature creep.

Methodologies

Waterfall: Finish one phase completely, then start the next. Each phase has mini-plan, and waterfalls into next. Drawback is that missing details in one phase end up causing issues in future phases.



Agile: Separate product into cycles and deliver working product quickly. Essentially small loops of waterfall, each only concerned about a small part of each step (eg requirements can be loose at the beginning, sprints may not produce working code). Constant feedback allows for issues to be spotted before they become massive problems. Lots of customer interaction can lead the project astray.



Spiral: Similar to agile, but a prototype is made each sprint cycle.

Good Software

Efficient, timely, performant. **Maintainable:** Salable, Extendable, Reusable, Readable High Usability. **Reliable:** The ability of a system or component to perform required functions under static conditions for a specific period **Correctness:** Meets (specified) requirements. Correctness is achieved if it behaves exactly as intended for all of its use-cases. (eg passes tests) **Robust:** Meets unspecified requirements. The ability of a system to cope with errors during execution, and cope with erroneous input (eg the program should fail gracefully) **Distribution of cost in development** 40% of cost on initial development. 60% on maintenance. Of that, 20% is on making sure things are correct, 20% is adaptive (correcting things the client wants to be fixed), 20% is improvement.

Object-Oriented Programming

Encapsulation

Safeguarding the content of a class from direct outside access. Make certain fields private, access them through public getters and setters. Separation of concerns. **Modularity:** Programs should be made up of independent, interchangeable parts Information Hiding Single responsibility Principle **Open-close principle:** objects should be open for extension, but closed for modification.

Relationships between Classes

Inheritance

Is-a relationship. Good for reusability, code does not need to be rewritten.

Aggregation

Has-a relationship: A has-a B, then A has an instance of B. Must be mandatory upon creation, otherwise it is simply association / dependence

Composition

Part-of relationship: A part-of B, A can't exist without B, and vice-versa. A subset of aggregation.

Dependence / Association

Uses-a relationship: A uses-a B, then A creates an object of B inside a method. Association is generally between unrelated classes.

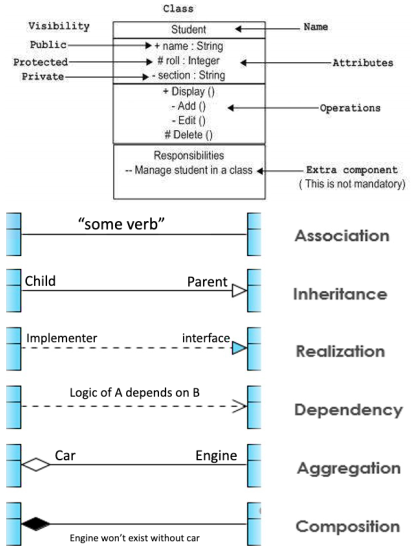
Abstraction

Hides complexity from users, showing only relevant info. Implementation hidden using abstract (partially abstract) classes, or interfaces (fully abstract).

Polymorphism

Performing a certain action in different ways (eg animals can make different noises). **Method overloading:** various methods with the same name but different parameters. **Method overriding:** child class overrides a method of its parent.

UML Diagrams



Java

Abstract Class

Can have abstract and non-abstract method. Can have non-final variables Can have final, non-final, static, and non-static variables. Can provide implementation of abstract class. Uses "extends" Can only extend one other class, but can implement multiple interfaces. Does not support multiple inheritance. Can have private members.

Interface

Can only have abstract methods. Variables are final by default. Can only have static and final variables. Can't provide implementation of abstract class. Can extend one or more Java interfaces. Supports multiple inheritance. Members are public by default.

Multiple Inheritance

When a class inherits from more than one class. Constructors are called in the order they are inherited.

SOLID

Single responsibility principle: Each class should have one and only one responsibility.
Open/Closed Principle: Classes should be open for extension but closed for modification
Liskov's Substitution Principle: Parent classes should be easily substituted with child classes without the application malfunctioning. **Interface Segregation Principle:** Many client-specific interfaces are better than one general interface. **Dependency Inversion Principle:** Classes should depend on abstraction, but not on concretion. Aka, have an interface which allows for communication with concrete classes. If class A changes, class B should not be affected.

Design Principles

Information Hiding

AKA Single Responsibility Principle AKA Encapsulation

Changes to a class should have **minimal** impact on other code: The API for a class should be completely independent of the implementation.

Open-Closed

Entities should be open for extension, but closed for modification.

In other words, the original functionality should remain static, while allowing new functionality to be added on, without the original source needing to be modified.

Design for Interfaces

AKA Dependency Inversion Don't depend on concrete classes, use interfaces instead.

Creational Patterns

Factory

Provides an interface for creating objects in a superclass, allows subclasses to alter type of objects that will be created.

1. Have an interface A

```
package factory;
public interface Factory {
  public Enemy spawn();
}
```

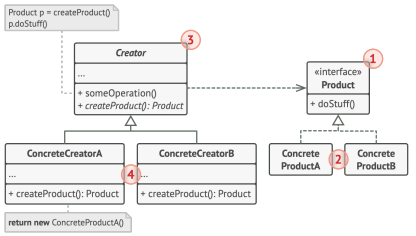
2. Have factory "products" implement A

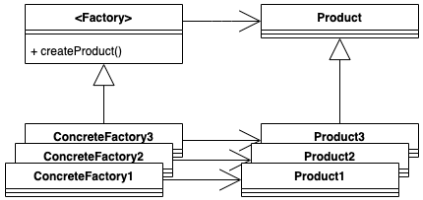
```
package factory;
public abstract class Enemy {
  protected int health;
  protected int strength;
  public abstract void
    takeDamage(int damage);
  public abstract void attack();
}
```

3. Have the factory function return an object of type A

```
package factory;
import java.util.Random;
public class AverageSpawner
    implements Factory{
  @Override
  public Enemy spawn() {
    Random r = new Random();
    double k;
    k = r.nextDouble();
    if(k < 0.65) {
      return new Minion();
    }
    else if(k < 0.9) {
      return new Elite();
    }
    else {
      return new Boss();
    }
  }
}
```

4. Let clients get "products" through the factory function





Advantages: Can switch out factories at runtime to change what's being produced. Object instantiation is encapsulated. Single Responsibility Principle - can move product creation into a separate area, making code easier to support. Open Closed Principle - can introduce new types of products without affecting existing code.
Disadvantages: Code can become complex due to subclasses.

Design Patterns

Decorator

Lets you attach new behaviors to objects by putting them in special wrappers that contain the behaviors.

1. Create interface A

```
public interface Burrito {
  public double getCost();
  public String getString();
}
```

2. Create base class B that implements A

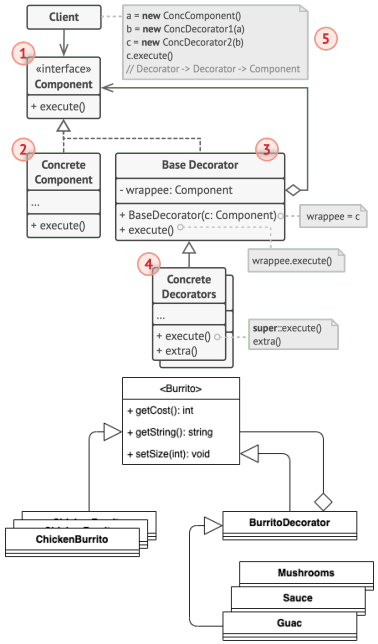
```
public class ChickenBurrito
    implements Burrito{
  public ChickenBurrito() {}
  public String getString() {
        return "Chicken" + "$12.00";
  }
  public double getCost() {
    return 12.00;
  }
}
```

3. Create decorator class C that implements A, that holds an instance of B, which is received through constructor

```
public abstract class
    BurritoDecorator
    implements Burrito{
  protected Burrito burrito;
  public BurritoDecorator
      (Burrito burrito) {
    this.burrito = burrito;
  }
  @Override
  public abstract double getCost();
  @Override
  public abstract String getString();
}
```

4. Create decorators by extending C and using super

```
public class Guac
  extends BurritoDecorator {
  public Guac(Burrito burrito) {
        super(burrito);
  }
  @Override
  public double getCost() {
        return 1.50
    + burrito.getCost();
  }
  @Override
  public String getString() {
        return burrito.getString()
    + "\n--Guac-+-"$1.50";
- -}
}
```



Decorators are both the original component, and also use the original component.
Advantages: Extensibility of code, new decor can just extend decorator interface. Greater flexibility, able to add or remove decorators at runtime. Simplifies coding, don't have to put all the functionality into the object. Single Responsibility Principle, larger classes can be broken down into several smaller ones.
Disadvantages: Code can be harder to maintain, decreases as number of decorator classes grows. Hard to remove wrappers. Hard to implement decorators in a way that doesn't depend on ordering.

Behavioral Patterns

Strategy

Define a family of algorithms, put them each in separate classes, and make their objects interchangeable.

1. Create a strategy interface

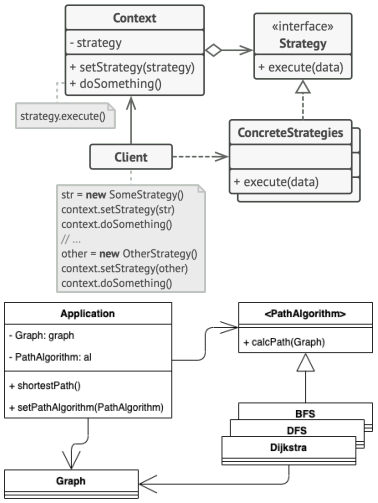
```
import java.util.ArrayList;
public interface Sorter {
  public void
    sort(ArrayList<Double> data);
}
```

2. Create concrete strategies that implement the interface

```
import java.util.ArrayList;
import java.util.Collections;
public class DefaultSorter
    implements Sorter{
  @Override
  public void
    sort(ArrayList<Double> data) {
    System.out
        .println("Default-Sort");
    Collections.sort(data);
  }
}
```

3. Create a context to store the strategy

```
import java.util.ArrayList;
import java.util.Collections;
public class SensorData {
  private ArrayList<Double> data
    = new ArrayList<Double>();
  private Sorter sort;
  public SensorData() {
        sort = new DefaultSorter();
  }
  public void
    addValue(double value) {
        data.add(value);
  }
  public void setSort(Sorter sort) {
        this.sort = sort;
  }
  public void print() {
        System.out
      .println(data.toString());
  }
  public void sort() {
        sort.sort(data);
  }
}
```



Advantages: Open-Closed Principle - Introducing new strategies without changing the encapsulation code, hiding of algorithm from application. Algorithms used by object can be changed at runtime. More maintainable, usable, extensible.
Disadvantages: Overly complex if there are only a few algorithms needed. Must be aware of the difference between algorithms to pick the right one.