# Exam Revision

## U1 Systems, Emergent Properties, Modelling, UML Component diagrams

- Systems thinking is about how components work together to achieve a goal
- Systems have the following characteristics:
  - Central objective
    - All components work together to achieve a common goal
  - Integration
    - How the components are  /can be put together to complete a system
  - Interaction
    - The System works as a result of components working together
  - Interdependence
    - A change to one component will affect other components
    - High Cohesion - all systems need to be in a working state
    - Low Cohesion - the input of an object is more important than its ability to interact
  - Organisation

We expect Systems to have the following components:

- input , process output
- Environment - circumstances under which the system works
  - internal - what the system has control over

    - external - system has no control over
- Boundary - dotted line separating internal and external environment
- Interface - How the user interacts with the system
- Feedback & Control - regulate and maintain systems behaviour

**Emergent Properties**

---

# U2 Software Lifecycle, Development processes, agile and UML Activity Diagrams

---

# U3 Requirements engineering, use cases, use case descriptions, UML use case diagrams

---

# U4 Business / Domain analysis, UML object and class diagrams, state machine diagrams

---

# U5 Systems Analysis

---

# U6 Software Testing

# U7 Software Design and Trustworthiness

# U8 Software Architecture

# U9 Detailed Design

## Coupling and Cohesion

### Coupling

Coupling is the degree to which a class knows about another class.

A lowly coupled class may access information through interfaces or other class.

A highly coupled class would access information almost directly.

Tight coupling might make other classes not work after changes

> You have a class that adds two numbers, but the same class creates a window displaying the result. This is a low cohesive class

> because the window and the adding operation don't have much in common. The window is the visual part of the program and the adding function is the logic behind it.
>
> To create a high cohesive solution, you would have to create a class Window and a class Sum. The window will call Sum's method to get the result and display it. This way you will develop separately the logic and the GUI of your application.

Some guy on stack overflow

**Interaction coupling**

The number of messages sent between objects and parameters passed in those messages:

Like talking on the phone to a friend and telling them about your life:

- Low interaction coupling - you tell them about your life details and just the important stuff
- High interaction coupling - You tell your friends about everything and everyone including other friends which affects everyone's understanding.

So, low interaction coupling is like having private conversations without everyone knowing everything, while high interaction coupling is like a big group chat where everyone knows everything about everyone. Lower coupling often makes your "communication" (code) more flexible and easier to manage

**Inheritance Coupling**

The degree to which a class needs its inherited features / methods.

Low inheritance coupling means a class uses a small specific set of features and is less dependent on on its own implementation details. ( even if there are a lot of methds it only uses a small amount of them like stakc and vector ) , changes to superclass less likely to affect subclass

High inheritance coupling - subclass has a strong dependency on the implementation details of superclass, changes in superclass are likely to affect subclass.

Low inheritance coupling is generally favorable

## Cohesion

The degree to which a class has a single and well focused purpose.

The more focused a class is on a single purpose the higher its cohesion.

High cohesion classes are better as they’re easier to maintain and more reusable.

### Operation Cohesion

The degree to which an operation focuses on a single functional requirement.

A method should perform a single atomic task.

### Class Cohesion

The number of things a class represents / focuses on a single requirement.

each class should only represent a single entity

### Specialisation Cohesion

Measures the semantic cohesion of inheritance hierarchies

class A should extend B because B is a specialised version of A , not because it needs its attributes and operations.

### Temporal Cohesion

if a component within a program performs multiple tasks that are interconnected by their timing it exhibits temporal cohesion.

- Imagine a set of tasks that needs to be done at specific times.
- If you put all the tasks in one component like a method:
    - they are interconnected by when they need to be done

- We can manipulate this by putting them in a specific order.

We can group tasks like fetch data , process data and upload data into one method because they all form part of a larger process. We put them in that specific order because thats the order they need to go in.

Temporal cohesion focuses on organising tasks based on when they should happen even if they do different things.

---

# U10 Design Patterns

## Creational

How objects are created

### Singleton

An objects that can only be instantiated once.

Can be used to implement a global app settings.

```
class Settings {
    static instance: Settings;
    public readonly mode = 'dark';

  //prevent new instances with private constructor
    private constructor(){

    }

    //create a get method that checks if an instance already exi
    static getInstance() : Settings{
```

```
        if(!Settings.instance){
            //if not make a new instance
            Settings.instance = new Settings();
        }
        return Settings.instance;
    }

}

const settings = Settings.getInstance();
```

We can achieve the same effect by declaring it as a global variable.

## Prototype

also known as clone.

Inheritance can lead to a complex hierarchy of code.

Prototype solves this by replacing inheritance form a class by inheriting from an object that has already been created.

We can do this by creating a new instance of an object , say lion and then create another object, say cat , and passing in the lion as a prototype.

## Builder

Objects are created step by step using methods as opposed all at once using the constructor.

We can accomplish this by returning `this` which is a reference to the object itself in the methods.

## Factory

Instead of using the new keyword to create objects we use a function or method.

# Structural

How objects relate to each other

## Adapter

Allows objects with incompatible interfaces to collaborate.

It works by converting the interface of an object so that the other can understand it.

It works by hiding the complexity of conversion happening behind the scenes.

Like an object that operated in km can be wrapped to operate in miles.

Neither object is aware or the adapter.

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects

## Composite

**Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

or example, imagine that you have two types of objects: `Products` and `Boxes`. A `Box` can contain several `Products` as well as a number of smaller `Boxes`. These little `Boxes` can also hold some `Products` or even smaller `Boxes`, and so on.

The Composite pattern suggests that you work with `Products` and `Boxes` through a common interface which declares a method for calculating the total price.

How would this method work? For a product, it'd simply return the product's price. For a box, it'd go over each item the box contains, ask its price and then return a total for this

box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated. A box could even add some extra cost to the final price, such as packaging cost.

### Facade

A facade is an interface that hides low level implementation from the user.

It works like an API , the end user can use simple methods to interact with the complex code behind the facade.

### Proxy

A View can create data but the framework itself needs to intercept data and update the ui when that data changes.

We handle this by replacing the original object witha Proxy.

The proxy takes the original object as a parameter and a handler as the second,

we can then override methods like get and set within the code whenever a property of the object is accessed or changed.

Proxies are used when we have large objects that would require a lot of resources to duplicate, but we still need to work with them.

## Behavioral

How Objects communicate with each other.

### State

An object behaves differently based on a finite number of states.

The state pattern allows us to start with a base class and provide different functionality depending on its internal state.

We can create a separate class for each potential state with an identical method that behaves differently.

## Observer

Allows Objects to subscribe to events that are broadcast by another object.

its a one to many relation.

A radio tower can send out a transmission that many receivers can intercept.

Create a mediator that sits between the object and objects subscribed to provide communication

## Strategy

**Strategy** is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.

The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

The context isn’t responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn’t know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.

## Mediator

**Mediator** is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other. Instead, these components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components. As a result, the components depend only on a single mediator class instead of being coupled to dozens of their colleagues.

### Iterator

Allows you to traverse through a collection of objects.

Looping over an array of items using an enhanced for statement is using the iterator pattern.

We can create an Iterator pattern by defining an object that has a next method which returnd an object that has a value of the current object we’re looking at in the loop and a done property so it can finish iterating , like a count or step.

---

# U11 Implementation

---

# U12 Refactoring

### Extract hierarchy

Refine the current class by introducing a new class hierarchy or extending exiisting hierarchy.

Move methods and fields that are relevant to all instances to subclasses so they only inherit methods they need.

### Tease Apart inheritance

Identify common properties and model them as separate class hierarchies.

### Convert Procedural Design to Objects

Remove procedural components from classes

## Separate domain from presentation

Move display methods from a class into separate display / ui classes.

## Method level only:

## Inline method

Replace a call to a method by its statements / code

```
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ? 2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
//turns into
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

## Problem

When a method body is more obvious than the method itself, use this technique.

## Solution

Replace calls to the method with the method’s content and delete the method itself.

## Inline temp

Replace a local variable with the associated expression

```
boolean hasDiscount(Order order) {
  double basePrice = order.basePrice();
  return basePrice > 1000;
}
//turns into
boolean hasDiscount(Order order) {
  return order.basePrice() > 1000;
}
```

## Problem

You have a temporary variable that’s assigned the result of a simple expression and nothing more.

## Solution

Replace the references to the variable with the expression itself.

## Introduce explaining variable

use a local variable to replace a portion of a complex expression.

## Extract class

Classes always start out clear and easy to understand. They do their job and mind their own business as it were, without butting into the work of other classes. But as the program expands, a method is added and then a field... and eventually, some classes are performing more responsibilities than ever envisioned.

## Problem

When one class does the work of two, awkwardness results.

## Solution

Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.

## Move Method

1. You want to move a method to a class that contains most of the data used by the method. This makes **classes more internally coherent**.
2. You want to move a method in order to reduce or eliminate the dependency of the class calling the method on the class in which it's located. This can be useful if the calling class is already dependent on the class to which you're planning to move the method. This **reduces dependency between classes**

## Problem

A method is used more in another class than in its own class.

## Solution

Create a new method in the class that uses the method the most, then move code from the old method to there. Turn the code of the original method into a reference to the new method in the other class or else remove it entirely.

## Move Field

When a new class is introduced, existing fields may need to be moved to the new class

## Problem

A field is used more in another class than in its own class.

## Solution

Create a field in a new class and redirect all users of the old field to it.

## Problem

A field is used more in another class than in its own class.

## Solution

Create a field in a new class and redirect all users of the old field to it.

## Pull up field

When subclasses have the same field move it to the superclass.

## Problem

Two classes have the same field.

## Solution

Remove the field from subclasses and move it to the superclass.

## Pull up method

When methods in different subclasses have the same behaviour move them to the

## Problem

Your subclasses have methods that perform similar work.

## Solution

Make the methods identical and then move them to the relevant superclass.

## Push down field

When a field is in a superclass but only used by a few subclasses.

## Problem

Is a field used only in a few subclasses?

## Solution

Move the field to these subclasses.

## Push down method

When a method in a superclass is used by only a few subclasses move it to the subclasses

## Problem

Is behavior implemented in a superclass used by only one (or a few) subclasses?

## Solution

Move this behavior to the subclasses.

## Extract Subclasses

When there is a class which has methods and implementation for rare use cases, create a subclass of it and move the feature(s) there.

## Problem

A class has features that are used only in certain cases.

## Solution

Create a subclass and use it in these cases.

## Extract Superclass

One type of code duplication occurs when two classes perform similar tasks in the same way, or perform similar tasks in different ways. Objects offer a built-in mechanism for simplifying such situations via inheritance. But oftentimes this similarity remains unnoticed until classes are created, necessitating that an inheritance structure be created later.

## Problem

You have two classes with common fields and methods.

## Solution

Create a shared superclass for them and move all the identical fields and methods to it.

## Extract interface

Make a new interface from an existing class

## Problem

Multiple clients are using the same part of a class interface. Another case: part of the interface in two classes is the same.

## Solution

Move this identical portion to its own interface.

---