Software Engineering

System Analysis and Design

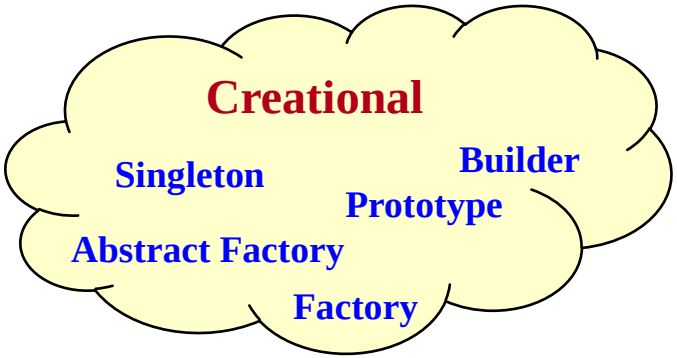
SYSTEM DESIGN OUTLINE

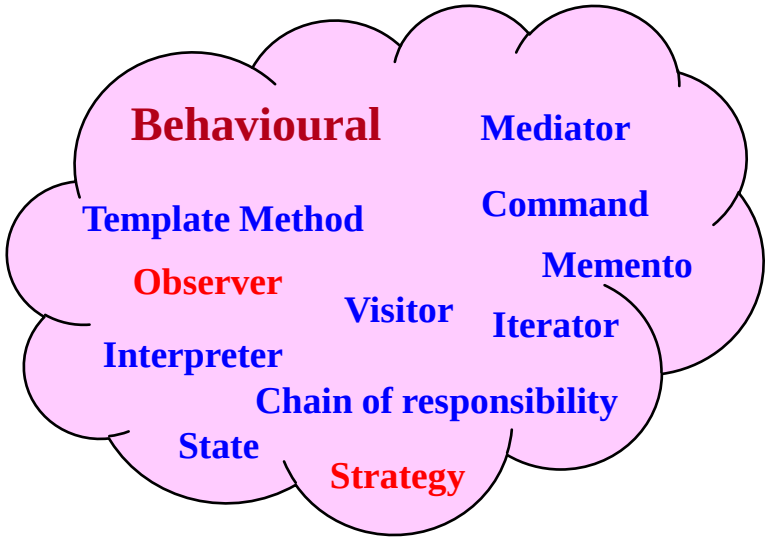
- ✓ System Design Overview
  - – Life Cycle Role
  - – The Purpose and Importance of System Design
  - – Realizing Design Goals
  - – Dealing with the Implementation Environment

- ➔ **System Analysis and Design Activities**
  - ✓ Architectural Analysis and Design
  - ✓ Use-case Analysis
  - ✓ Class Design
  - ✓ Object Behaviour Analysis: State Machine Diagrams
  - ☞ **Design Patterns**
  - – Anti Patterns

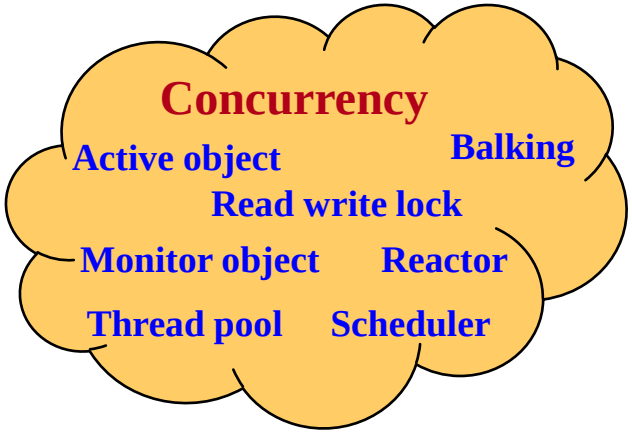
DESIGN PATTERNS

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

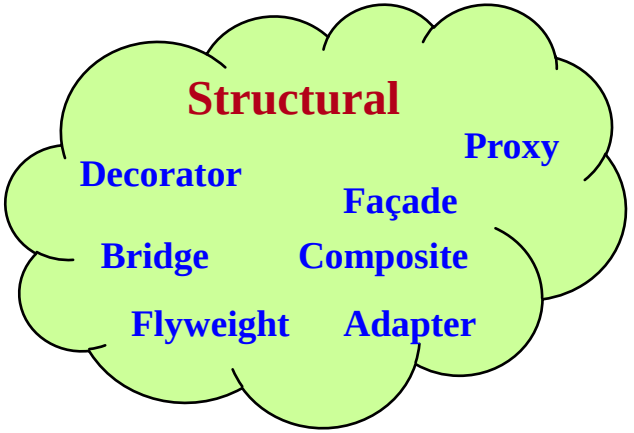




Behavioural patterns are concerned with how classes and objects interact and distribute responsibility.



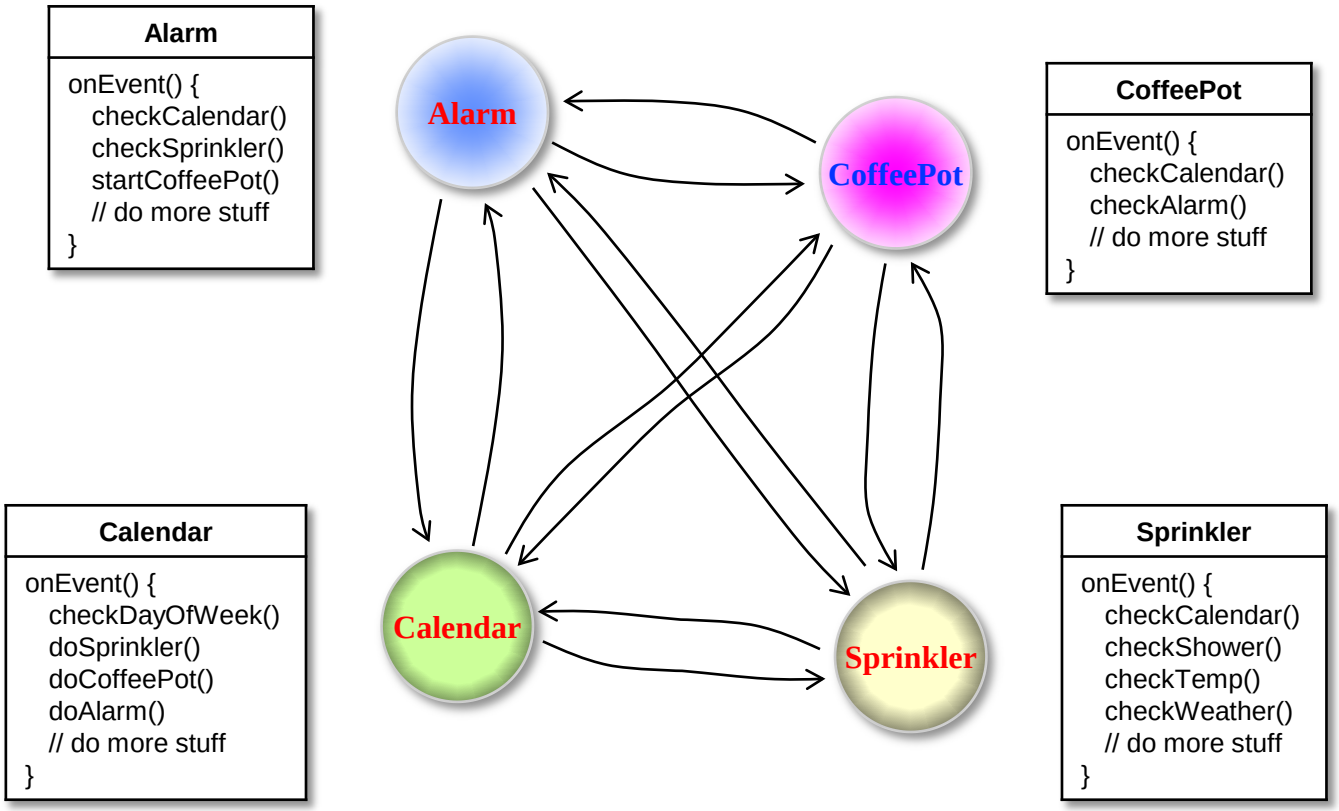
Concurrency patterns manage concurrent interactions among multiple objects.



Structural patterns compose classes or objects into larger structures.

BOB'S HOUSE OF THE FUTURE

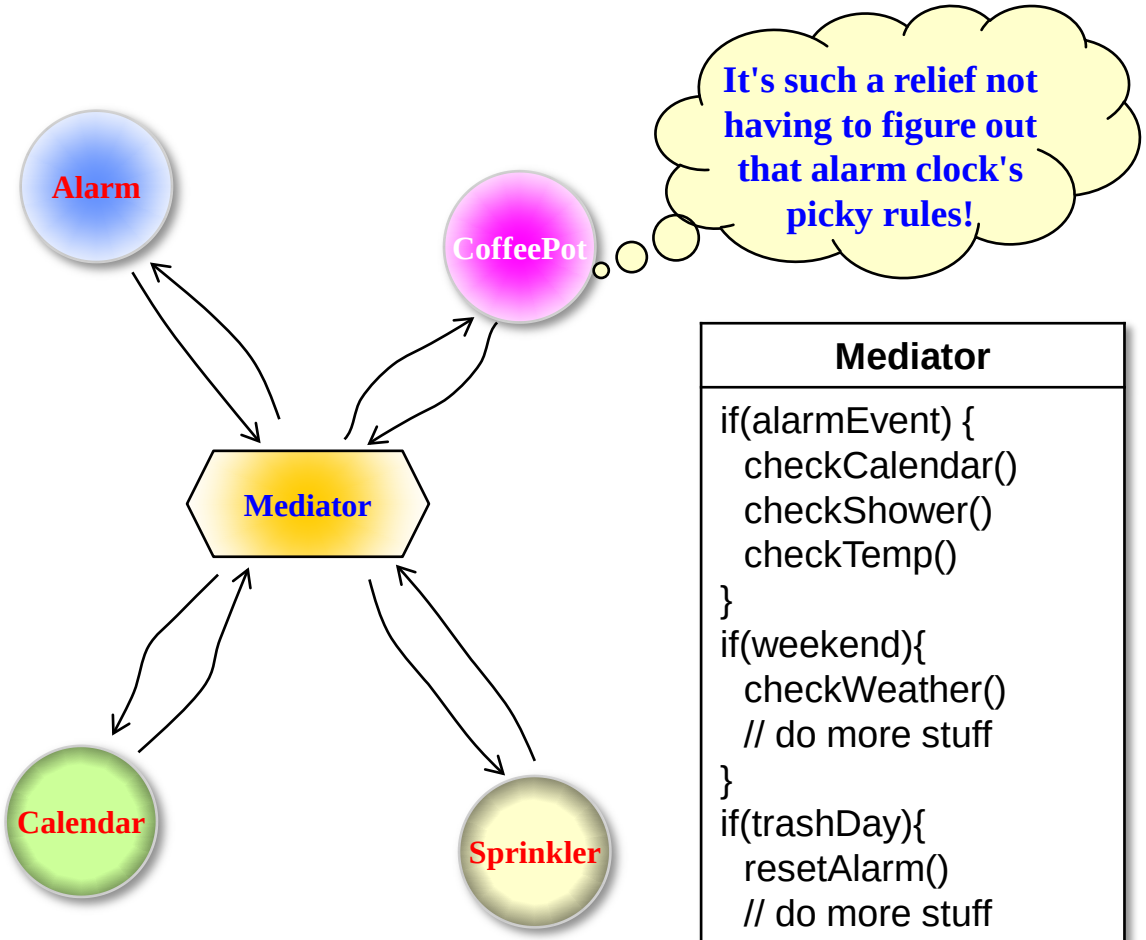
Bob's house is Java-enabled!



HouseOfTheFuture's dilemma???

MEDIATOR PATTERN

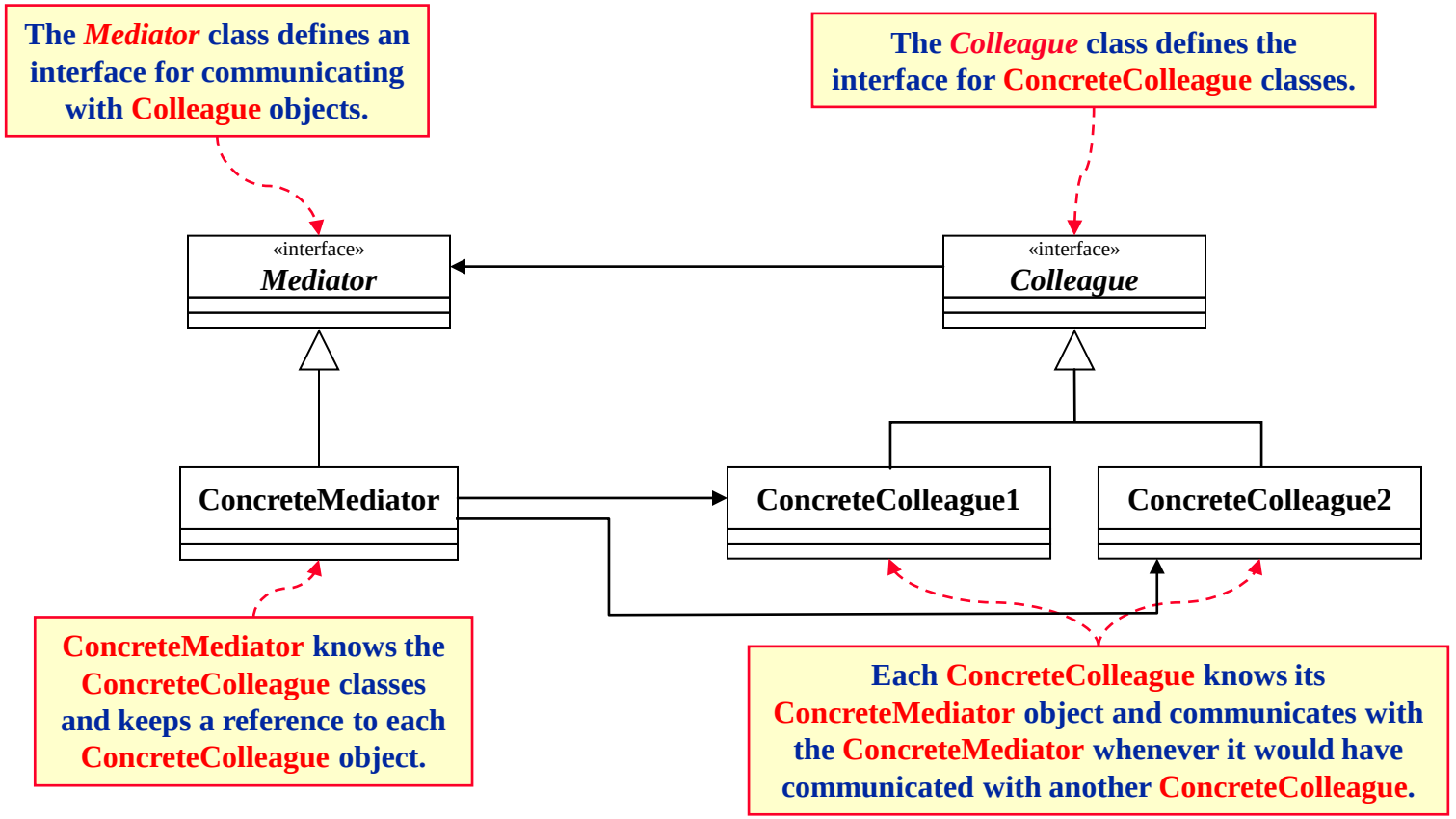
- All appliances now only need to interact with the Mediator:
  - tell the Mediator when their state changes.
  - respond to requests from the Mediator.



Mediator

```
if(alarmEvent) {
  checkCalendar()
  checkShower()
  checkTemp()
}
if(weekend){
  checkWeather()
  // do more stuff
}
if(trashDay){
  resetAlarm()
  // do more stuff
  checkAlarm()
  // do more stuff
}
```

MEDIATOR PATTERN: CLASS DIAGRAM



The MEDIATOR pattern is commonly used to coordinate related GUI components.

MEDIATOR PATTERN

Advantages

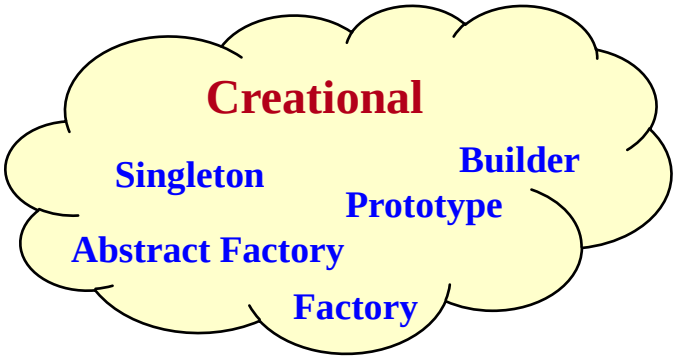
- **Decoupled Colleagues:** The colleague classes are totally decoupled. Adding a new or reusing a colleague class is very easy.
- **Comprehension:** Simplifies understanding and maintenance since the mediator centralizes and encapsulates the control logic.
- **Simplified Object Protocols:** The colleague objects need to communicate only with the mediator object (reduces communication from many-to-many to one-to-many).
- **Limits Subclassing:** Only need to extend the mediator class when the logic needs to be extended.

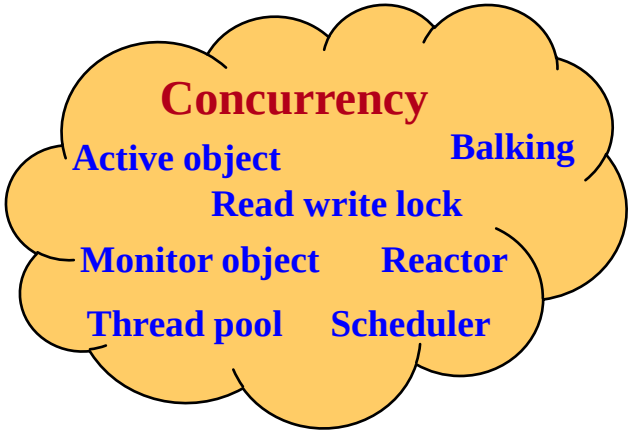
Disadvantages

- **Complexity:** The Mediator object can become overly complex.

DESIGN PATTERNS

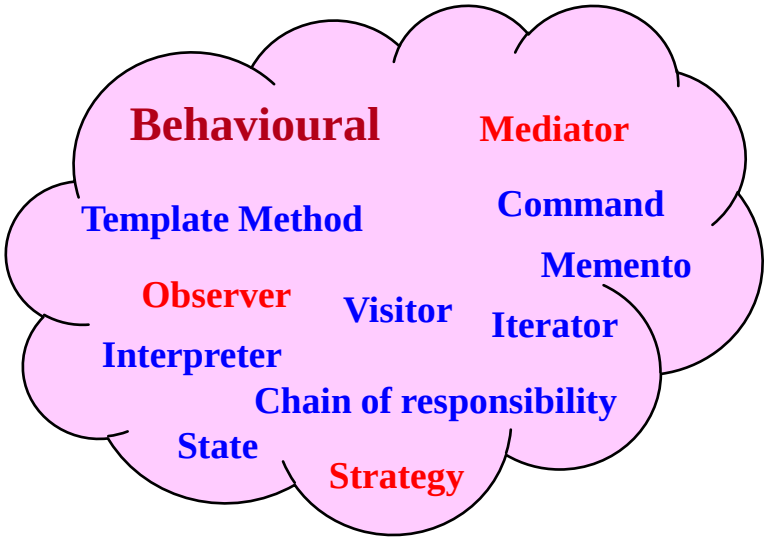
Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

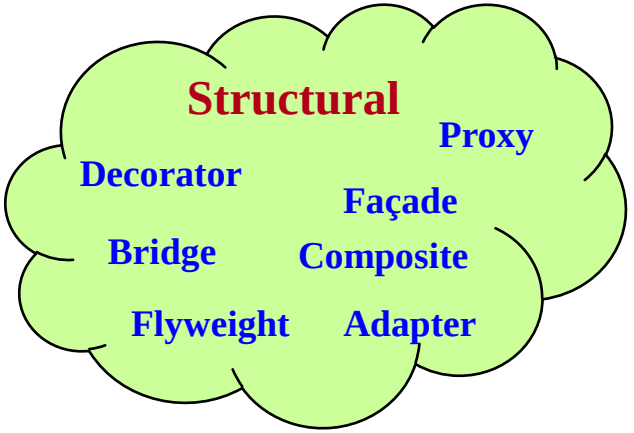




Concurrency patterns manage concurrent interactions among multiple objects.

Behavioural patterns are concerned with how classes and objects interact and distribute responsibility.





Structural patterns compose classes or objects into larger structures.

PROXY PATTERN

Problem

Sometimes we want to defer the full cost of creating and initializing an object until we actually need to use it.

Example

- In a document editor that can embed graphical objects in a document, objects like large raster images can be expensive to create.

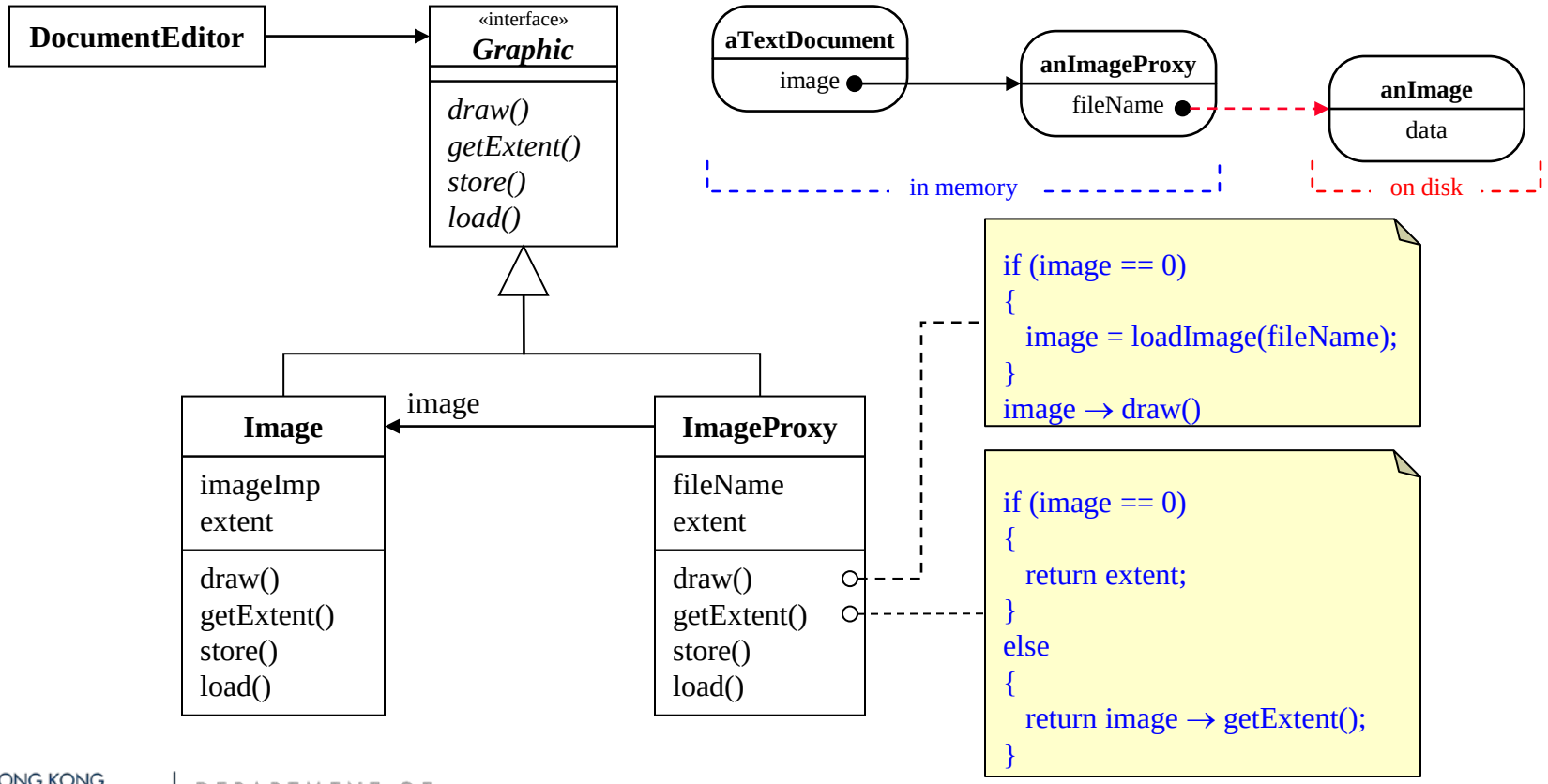
- However, opening a document should be fast. Thus, we should avoid creating expensive objects all at once when the document is opened. Moreover, usually not all objects are visible at the same time anyway.

☞ **Expensive objects should be *created on demand.***

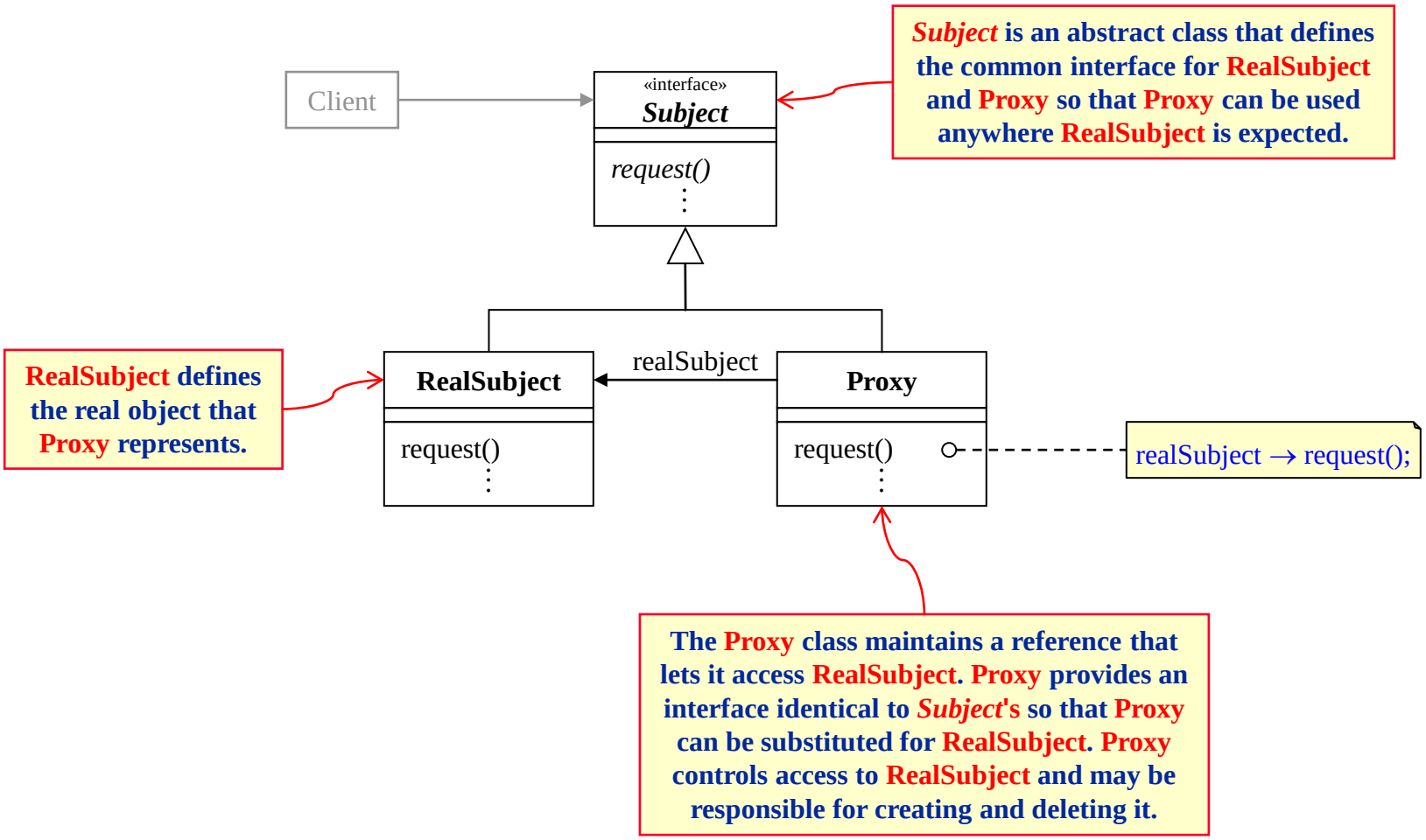
PROXY PATTERN: SOLUTION

Use an *image proxy* that acts as a stand-in (*surrogate*) for the real image (i.e., a *virtual* proxy).

The proxy acts just like the image, instantiates it when needed and forwards request to it after creating it.

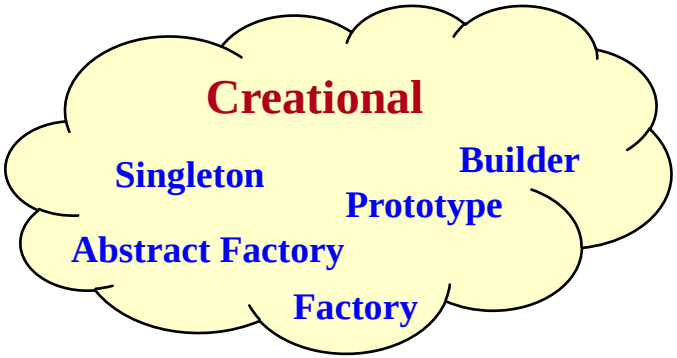


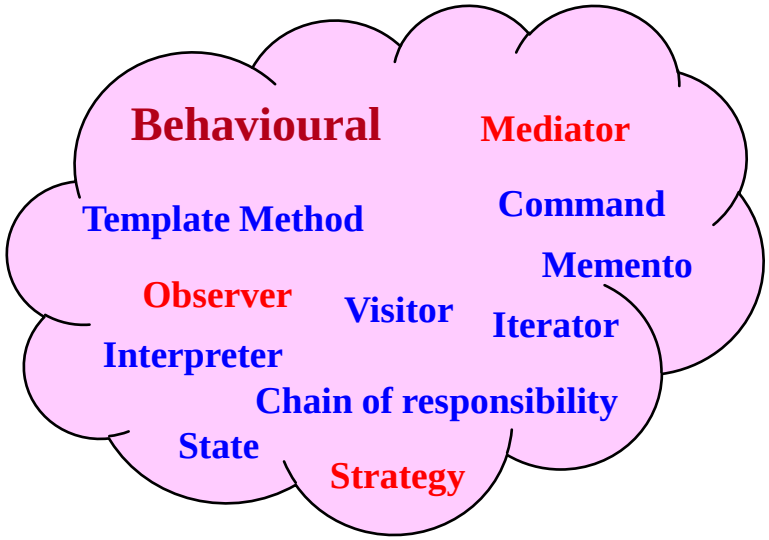
PROXY PATTERN: CLASS DIAGRAM



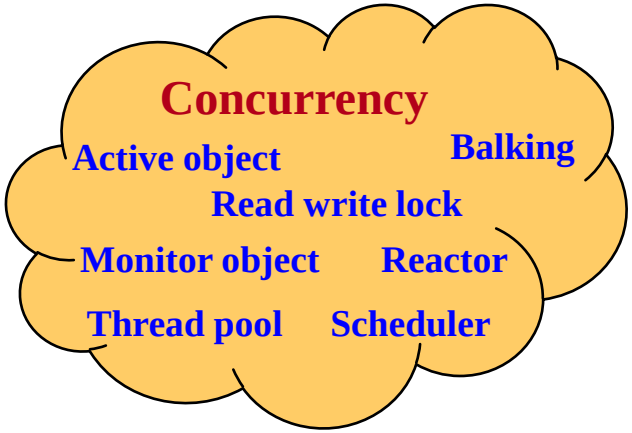
DESIGN PATTERNS

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

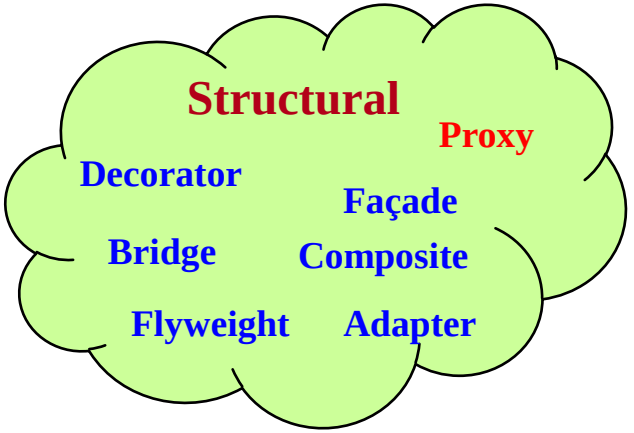




Behavioural patterns are concerned with how classes and objects interact and distribute responsibility.



Concurrency patterns manage concurrent interactions among multiple objects.

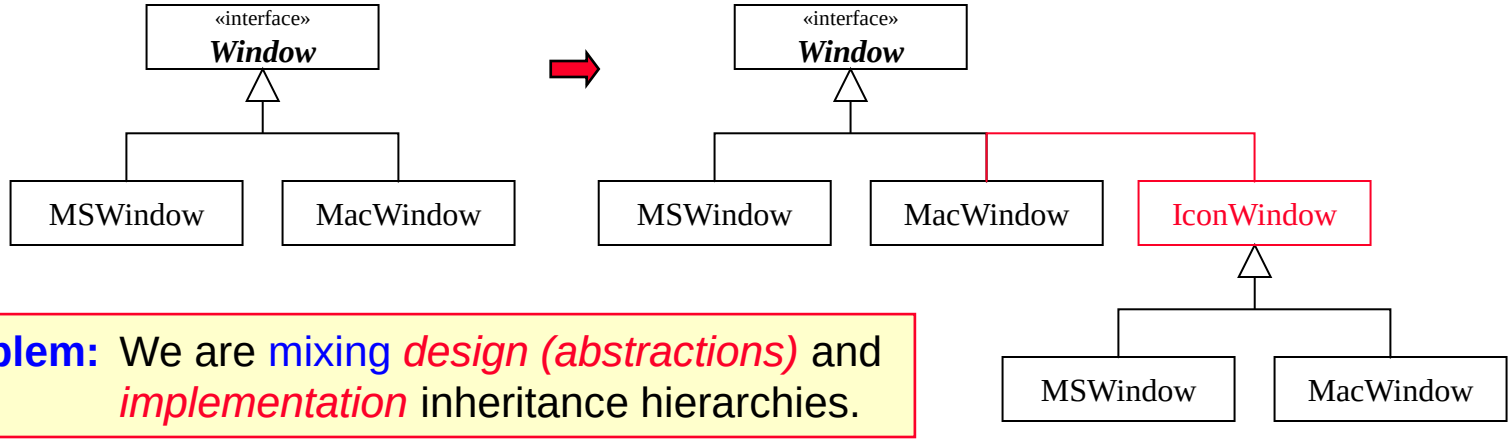


Structural patterns compose classes or objects into larger structures.

BRIDGE PATTERN

Problem: When an abstraction can have several possible implementations, inheritance is usually used to handle this. However, this binds an implementation to an abstraction permanently making it difficult to modify, extend and reuse abstractions and implementations *independently*.

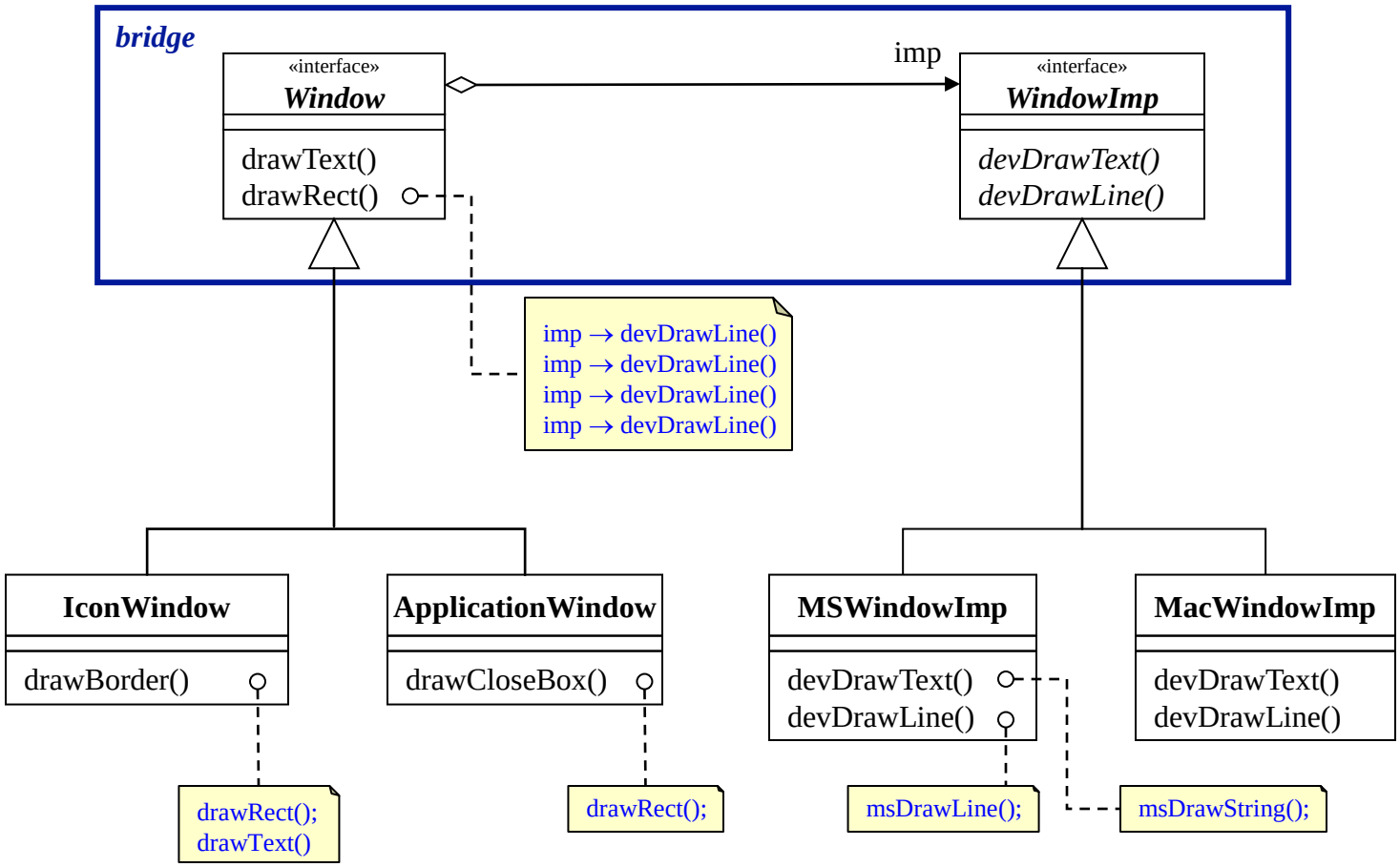
E.g., implementation of a portable Window abstraction in a user interface toolkit.



Problem: We are mixing *design (abstractions)* and *implementation* inheritance hierarchies.

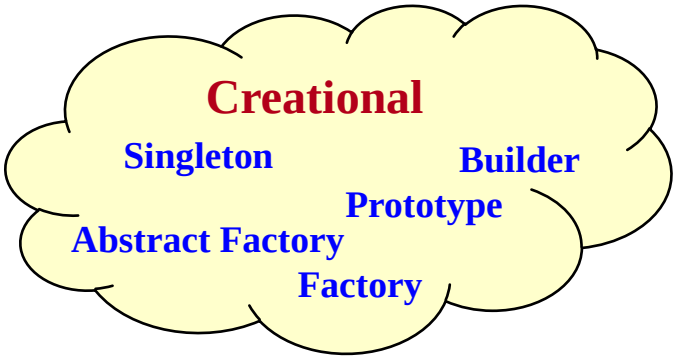
BRIDGE PATTERN: SOLUTION

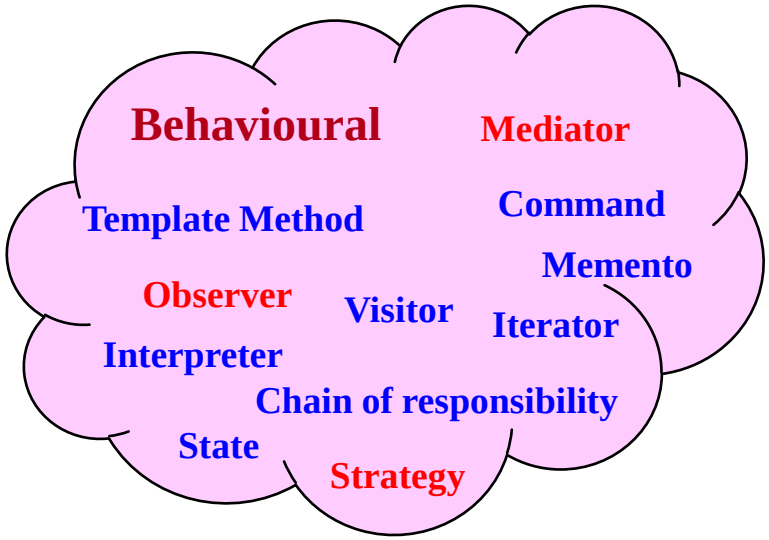
Separate the window abstraction and implementation class hierarchies and connect them using a *bridge*.



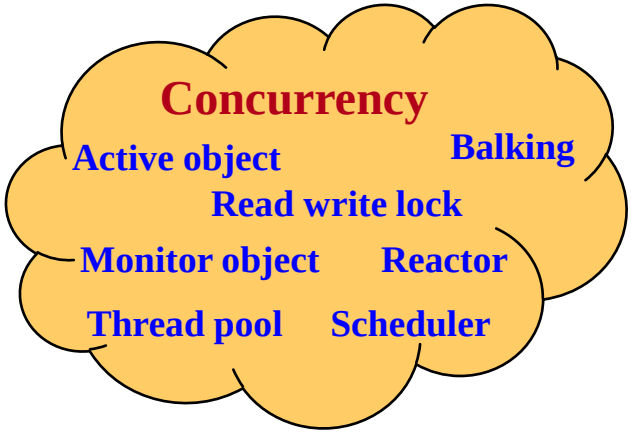
DESIGN PATTERNS

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

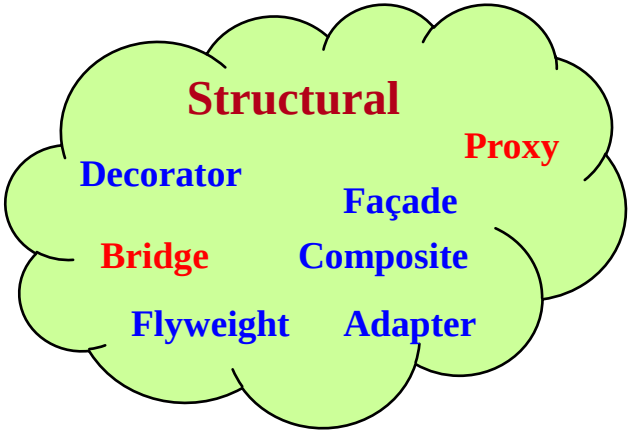




Behavioural patterns are concerned with how classes and objects interact and distribute responsibility.



Concurrency patterns manage concurrent interactions among multiple objects.



Structural patterns compose classes or objects into larger structures.

SINGLETON PATTERN

Problem: It is important for some classes to have ***exactly one instance***. How to ensure that a class has only one instance and that the instance is easily accessible?

A global variable can make an object accessible, but how to prohibit instantiation of multiple instances?

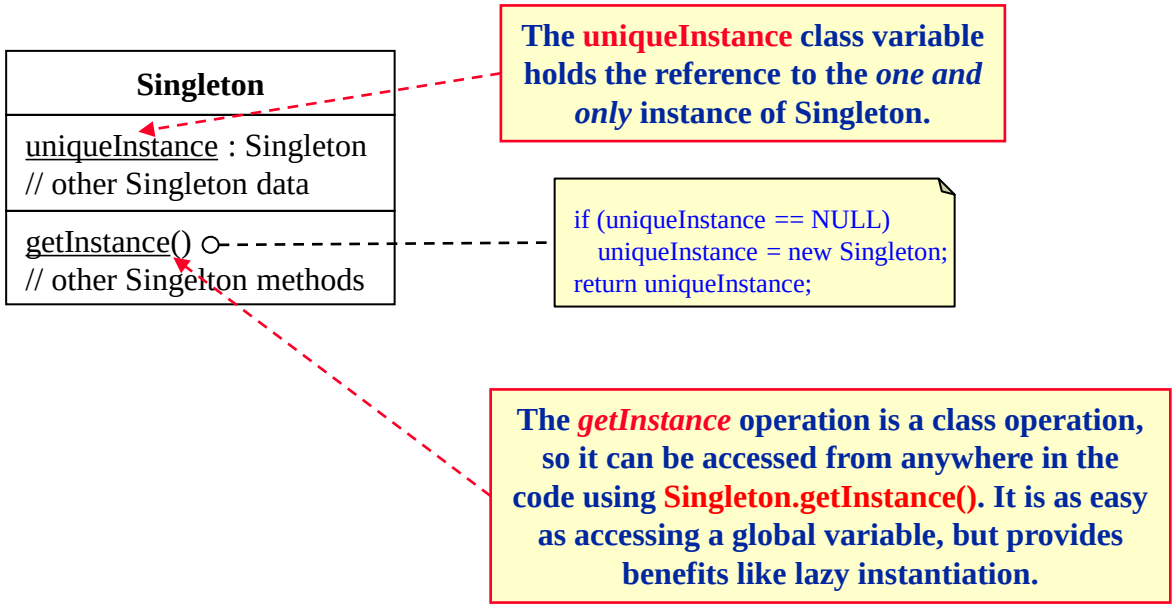
Why would we need only one instance of an object?

- Thread pools
- Caches
- Dialog boxes
- Objects used for logging
- Objects that handle graphics cards?
- System.out?

SINGLETON PATTERN: CLASS DIAGRAM

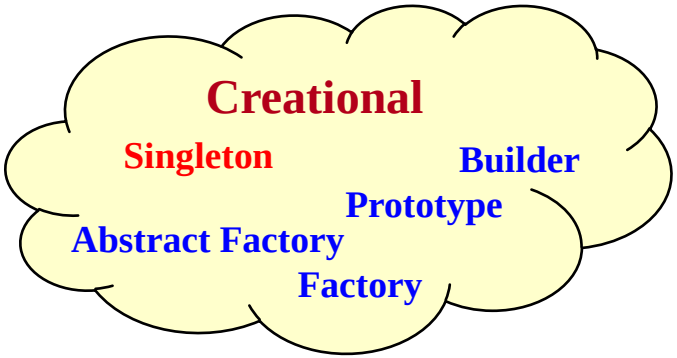
Make the singleton class itself responsible for

- keeping track of its sole instance,
- ensuring that no other instance can be created, and
- providing a way to access the instance.

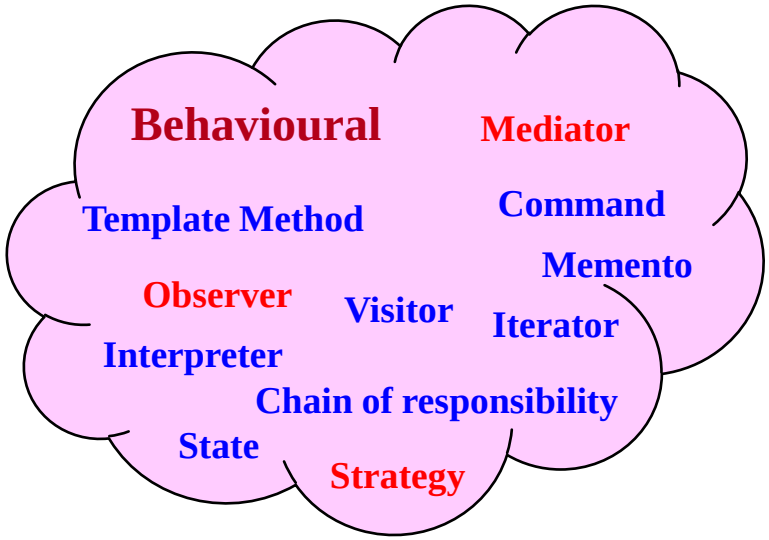


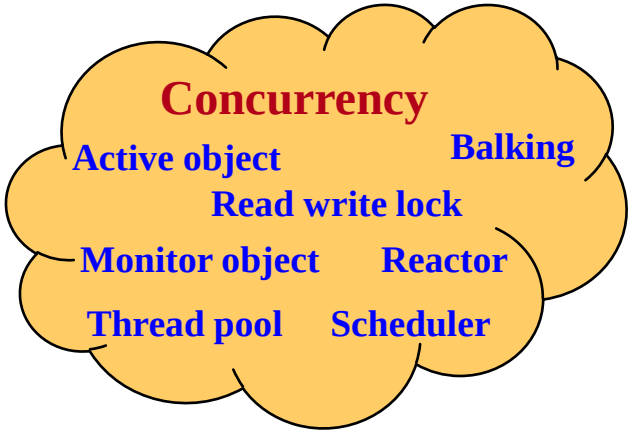
DESIGN PATTERNS

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

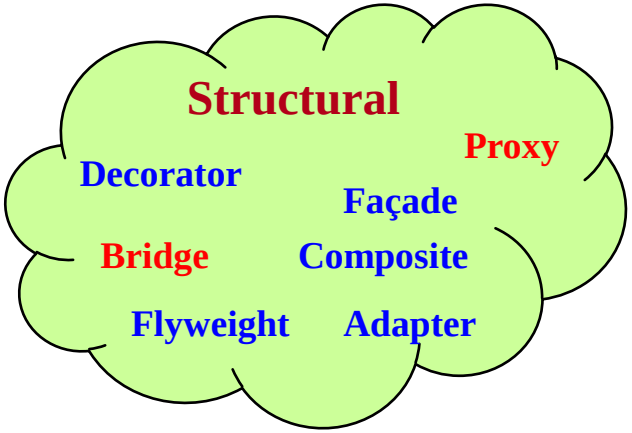


Behavioural patterns are concerned with how classes and objects interact and distribute responsibility.





Concurrency patterns manage concurrent interactions among multiple objects.



Structural patterns compose classes or objects into larger structures.

SIMPLE FACTORY PATTERN

- To use classes, they need to be instantiated (using new operator).

- Often, we need to decide at runtime what specific subclass to instantiate.

```
Duck duck;
if (picnic)          { duck = new MallardDuck(); }
else if (hunting)    { duck = new DecoyDuck; }
else if (inBathTub) { duck = new RubberDuck(); }
```

- This kind of code is not closed for modification.
  - To extend this code with new concrete types, it will have to be reopened.

THE OPEN-CLOSED PRINCIPLE

Design Principle

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

We should allow a design to incorporate new behaviour without modifying existing code.

How best to achieve it?

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

SIMPLE FACTORY PATTERN: MAKING PIZZA

```
Pizza orderPizza() {
      Pizza pizza = new Pizza();

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

We want to be able to make different types of pizza.

So, for flexibility, we really want this to be an abstract class or interface, but we cannot directly instantiate either of those.

SIMPLE FACTORY PATTERN: MAKING PIZZA

```
Pizza orderPizza(string type) {
      Pizza pizza;

      if (type.equals("cheese")) {
        pizza = new CheesePizza();
      } else if (type.equals("greek") {
        pizza = new GreekPizza();
      } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
      }

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

We are now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings & cheese), then we bake it, cut it and box it!

Each Pizza subtype (CheesePizza, GreekPizza, etc. knows how to prepare itself.

SIMPLE FACTORY PATTERN:
ADDING MORE PIZZA TYPES

This code is NOT closed for modification. If the Pizza Shop always changes its pizza offering, we have to get into this code and modify it.

```
Pizza orderPizza(string type) {
    Pizza pizza;

    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek") {
      pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
    } else if (type.equals("clam") {
      pizza = new ClamPizza();
    } else if (type.equals("veggie") {
      pizza = new VeggiePizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

This is what varies. As the pizza selection changes over time, you will have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, baking and packaging a pizza has remained the same for years and years. So, we do not expect this code to change, just the pizza it operates on.

SIMPLE FACTORY PATTERN:
ENCAPSULATING OBJECT CREATION

```
if (type.equals("cheese")) {
      pizza = new CheesePizza();
   } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
   } else if (type.equals("clam") {
      pizza = new ClamPizza();
   } else if (type.equals("veggie") {
      pizza = new VeggiePizza();
}
```

```
Pizza orderPizza(string type) {
      Pizza pizza;

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

First we pull the object creation code out of the orderPizza operation.

Then we place that code in an object that is only going to create pizzas. If any other object needs a pizza created, this is the object to come to.

What's going to go here?

SimplePizzaFactory

SIMPLE FACTORY PATTERN:
A SIMPLE PIZZA FACTORY

Here is our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

First we define a createPizza() operation in the factory. This is the operation all clients will use to instantiate new objects.

```
public class SimplePizzaFactory {
    public Pizza createPizza(string type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
          pizza = new CheesePizza();
        } else if (type.equals("pepperoni") {
          pizza = new PepperoniPizza();
        } else if (type.equals("clam") {
          pizza = new ClamPizza();
        } else if (type.equals("veggie") {
          pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Here is the code we pulled out of the orderPizza() operation.

This code is still parameterized by the type of the pizza, just like our original orderPizza() operation was.

SIMPLE FACTORY PATTERN:
REWORKING THE PIZZA STORE

Now we give PizzaStore a reference to a SimplePizzaFactory.

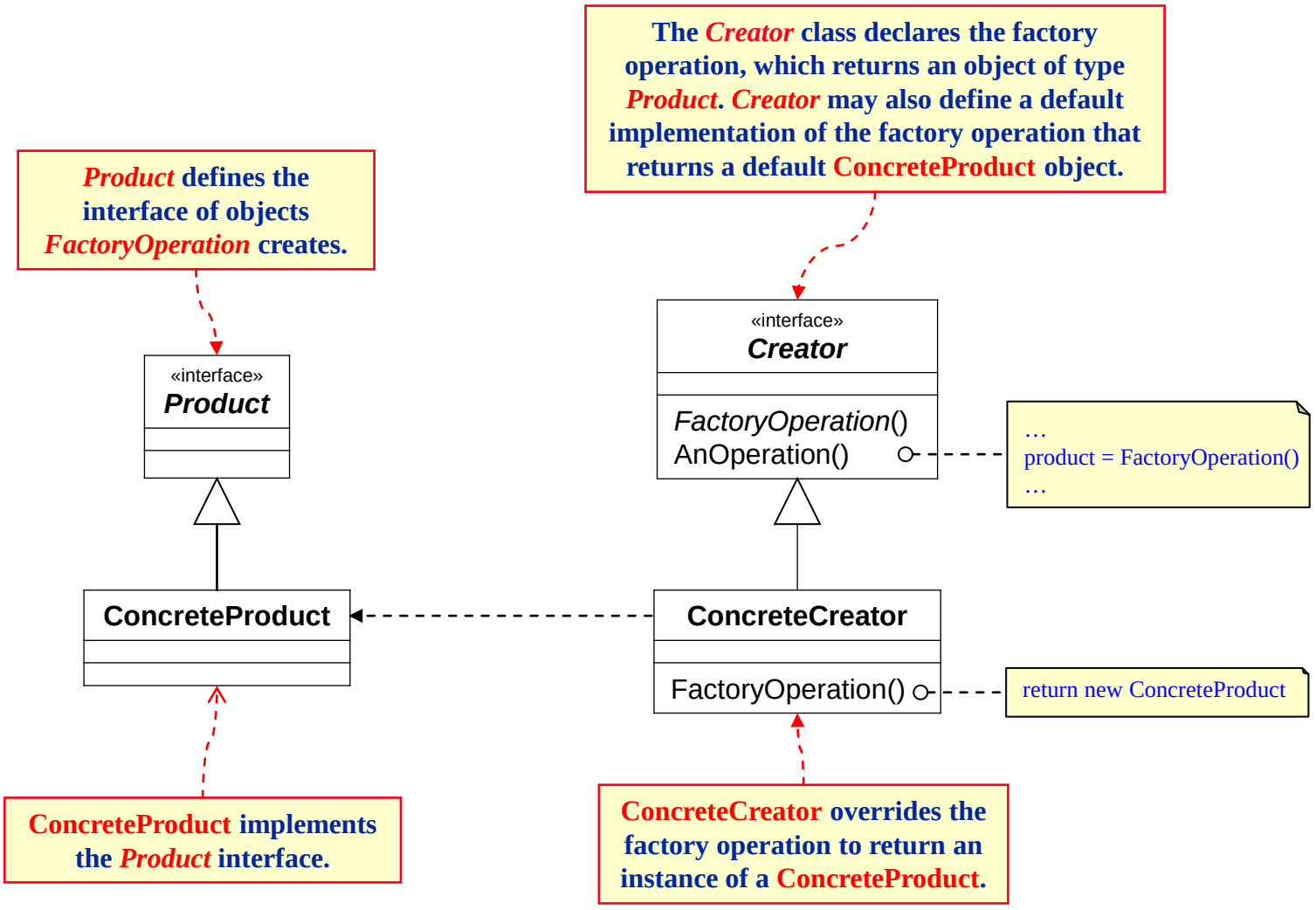
```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory
    }

    public Pizza orderPizza(string type) {
        Pizza pizza

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    // other operations here
}
```

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() operation uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we have replaced the new operator with a create operation on the factory object. No more concrete instantiation here!

SIMPLE FACTORY PATTERN: CLASS DIAGRAM



WHEN TO USE DESIGN PATTERNS

1. **Solutions to *problems that recur* with variations**
   - No need for reuse if the problem only arises in one context!

2. **Solutions that *require several steps***
   - Not all problems need all steps.
   - Design patterns can be overkill if the solution is a simple linear set of instructions.

3. **Solutions where the solver is *more interested in the existence of the solution* than its complete derivation**
   - Design patterns leave out too much to be useful to someone who really wants to understand.
   - **BUT**, they can be a temporary bridge …

BENEFITS OF DESIGN PATTERNS

- Design patterns *enable large-scale reuse* of software designs.
  - They also help document systems to enhance understanding.
- Design patterns *explicitly capture expert knowledge and design tradeoffs* and make this expertise more widely available.
- Design patterns *help improve developer communication*.
  - Design pattern names form a vocabulary among the developers.
- Design patterns *help ease the transition to object-oriented technology*.

DRAWBACKS OF DESIGN PATTERNS

- Design patterns *do not lead to direct code reuse*.
- Design patterns are *deceptively simple*.
- Development teams *may suffer from design pattern overload*.
- Design patterns are *validated by experience and discussion* rather than by automated testing.
- Integrating design patterns into a software development process is a *human-intensive activity*.

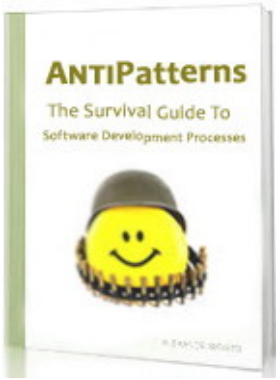
SYSTEM DESIGN OUTLINE

✓ System Design Overview
- Life Cycle Role
- The Purpose and Importance of System Design
- Realizing Design Goals
- Dealing with the Implementation Environment

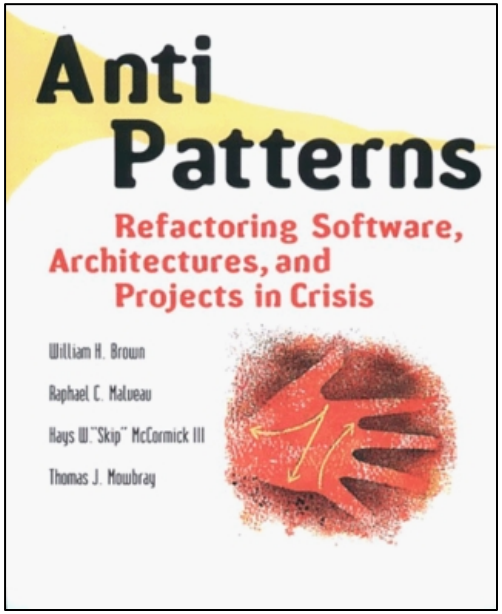
➔ **System Analysis and Design Activities**
- Architectural Analysis and Design
- Use-case Analysis
- Class Design
- Object Behaviour Analysis: State Machine Diagrams
- Design Patterns
- ☞**Anti Patterns**

ANTI PATTERNS

- An anti pattern tells you what is a **bad solution**.
  - Why a bad solution is attractive.
  - Why the solution is bad (in the long term).
  - What alternatives (good solutions) you have.



- There are many types of anti patterns
  - development
  - object-oriented
  - organizational
  - domain specific



WELL-KNOWN ANTI PATTERNS

- Anti patterns are all around us. They are often used as tools for social control.

Social Anti Patterns

- Criminal
- Terrorist
- Drug Addict/Dealer
- Heretic/Witch
- Dilbert's Pointy-haired Manager

Software Anti Patterns

- Spaghetti Code
- Stovepipe System
- Analysis Paralysis
- Design By Committee
- God Class
- Mythical Man Month
- Death March Project

DEVELOPMENT ANTI PATTERN: SPAGHETTI CODE

- **spa·ghet·ti code** [Slang] *an undocumented piece of software code that cannot be extended or modified without extreme difficulty due to its convoluted structure.*

Un-structured code is a liability

Well structured code is an investment

DEVELOPMENT ANTI PATTERN: SPAGHETTI CODE

Symptoms

- Quick demonstration code (prototype) that becomes operational.
- "Lone Ranger" programmer (*who was that masked man*)?
- Obsolete or scanty documentation.
- **50%** of maintenance is spent on system rediscovery.
- Hesitant Programmer Syndrome
  - More likely to break it than extend it.
  - Easier to just rewrite it.
- Cannot be reused
  - System software and COTS packages can't be upgraded.
  - Performance cannot be optimized.
- User work arounds.

DEVELOPMENT ANTI PATTERN: SPAGHETTI CODE

Object-Oriented Spaghetti Code

- Many object methods with no parameters.
- Suspicious class or global variables.
- Intertwined and unforeseen relationships between objects.
- Procedure-oriented methods, objects with procedure-oriented names.
- OO advantage lost — inheritance cannot be used to extend the system, polymorphism is not effective either.

Bottom Line

The software has reached a point of diminishing returns where the effort involved to maintain existing code exceeds the cost of developing a new "ground up" solution.

DEVELOPMENT ANTI PATTERN: THE BLOB/GOD CLASS

Symptoms

- A single class with many attributes and/or operations (i.e., a class that knows too much or does too much).
- Violates the cohesion (single responsibility) property of OO design.
- A nightmare to maintain (changes or bug fixes introduce new bugs).

DEVELOPMENT ANTI PATTERN:
THE BLOB/GOD CLASS

Typical Causes

- Inappropriate requirements specification
- Lack of an object-oriented architecture
- Lack of (any) architecture
- Lack of architecture enforcement
- Too limited intervention

DEVELOPMENT ANTI PATTERN: THE BLOB/GOD CLASS

Consequences

- Lost OO advantage.
- Too complex to reuse or test.
- Unnecessary code
- Expensive to load.

Refactor the code!

SYSTEM Analysis and DESIGN: RETROSPECTIVE

Architectural design

- Outlines the overall software structure of the system in terms of cohesive, loosely-coupled subsystems.
- Makes use of architectural patterns to organize subsystems.

Class Analysis

- Structures the system by partitioning use cases into boundary, control and entity classes.
- Assigns responsibilities to classes.

Class Design

- Completes the specification of each class (attributes, operations).
- Restructures and optimizes classes.
- Makes use of design patterns to handle commonly occurring design requirements, particularly nonfunctional requirements.

SYSTEM DESIGN: SUMMARY

The Analysis and Design Model contains:

Subsystems, their dependencies and their interfaces.

- ☞ **Implemented by implementation subsystems containing source code, scripts, binaries, executables, etc.**

Analysis classes and their responsibilities.

- ☞ **Localizes changes to boundary, entity or control classes.**

Design classes, their operations, attributes, relationships and implementation requirements.

- ☞ **Implemented by files containing source code.**

COMP 3111 SYLLABUS

- ✓ **1. Introduction**
- ✓ **2. Modeling Software Systems using UML**
- ✓ **3. Software Development**
- ✓ **4. System Requirements Capture**
- ✓ **5. Implementation**
- ✓ **6. Testing**
- ✓ **7. System Analysis and Design**
- 8. Software Quality Assurance
- 9. Managing Software Development

SYSTEM ANALYSIS & DESIGN

EXERCISE