

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory Builder <b>Factory Method</b> Prototype <b>Singleton</b>	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

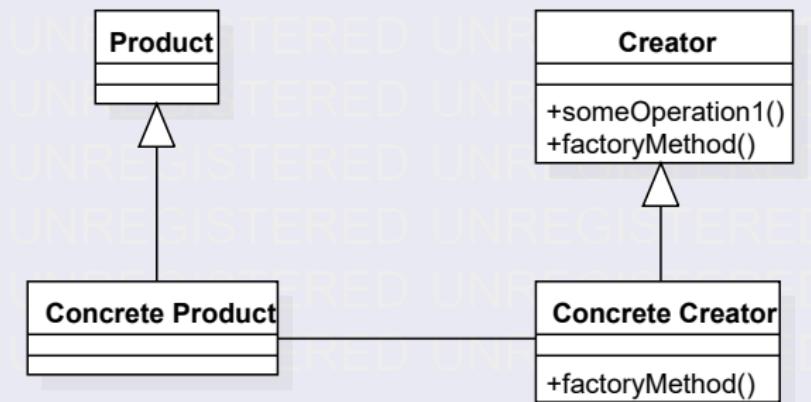
## Factory Method: Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Also known as Virtual Constructor

## Factory Method: Applicability

- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

## Factory Method: Structure



## Factory Objects

- In order to understand Factory Method patterns, we must first understand factory objects.
- A **factory object** operates like a factory in the real world, and creates objects.
- Factory objects **makes software easier to maintain and change**, because object creation happens in the factories. The methods that use these Factories can then focus on other behavior.

## Factory Objects: Example

- Imagine a situation where you have a software that implements an *online store that sells knives*.
- Perhaps when the store first opens, you only produce `steak knives` and `chef' knives`. You would have a superclass with those two subclasses.
- In your system, first, a knife object is created.
- Conditionals then determine which subclass of `Knife` is actually instantiated. This act of instantiating a class to create an object of a specific type is known as **concrete instantiation**.
- In Java, concrete instantiation is indicated with the operator “`new`”.

## Factory Objects: Example (contd.)

- However, imagine that the store is successful and adds more knife types to sell. New subclasses will need to be added, such as BreadKnife, ParingKnife, or FilletKnife. The list of conditionals would need to grow and grow as new knife types are added.
- However, the methods of the knife after it is created, such as sharpening, polishing, and packaging, would likely stay the same, no matter the type of knife. This creates a complicated situation.
- Instead of making Knives in the store, it may be better to create them in a Factory object.

## Factory Objects: Example (contd.)

- A factory object is an object whose role is to create “product” objects of particular types.
- **In this example**, the methods of sharpening, polishing, and packaging would remain in the `orderKnife` method. However, the responsibility of creating the product will be delegated to another object: a “Knife Factory”.
- The code for deciding which knife to create, and the code for deciding which subclass of `Knife` to instantiate, are moved into the class for this factory.

## Factory Objects: Example (contd.)

```
public class KnifeFactory extends KnifeStore{
{
    public Knife createKnife(String knifeType)
    {
        Knife knife = null;
        // create Knife object
        If (knifeType.equalsIgnoreCase("steak"))
        {
            knife = new SteakKnife();
        }
        else if (knifeType.equalsIgnoreCase("chefs"))
        {
            knife = new ChefsKnife();
        }
        return knife;
    }
}
```

## Factory Objects: Example (contd.)

- The code to create a `Knife` object has been moved into a method of a new class called `KnifeFactory`. This allows the `KnifeStore` class to be a client for the Knife Factory.
- The `KnifeFactory` object to use is passed into the constructor for the `KnifeStore` class.
- Instead of performing concrete instantiation itself, the `orderKnife` method delegates the task to the factory object.

## Factory Objects (contd.)

- Concrete instantiation is the primary purpose of Factories.
- In general, a factory object is an instance of a factory class, which has a method to create product objects.

## Factory Objects (contd.)

```
public abstract class KnifeStore
{
    public Knife orderKnife(String knifeType)
    {
        Knife knife;
        knife = createKnife(knifeType);

        knife.sharpening();
        knife.polishing();
        knife.packaging();
        return knife;
    }
    abstract Knife createKnife(String knifeType);
}
```

## Benefits of Factory Objects

There are numerous benefits to using factory objects.

- One of these benefits is that it is **much simpler to add new types of an object to the object factory** without modifying the client code.
- Factories allow client code to operate on generalizations. This is known as coding to an interface, not an implementation.
- The use of the word “factory” serves a good metaphor here. Using Factory objects is similar to using a factory in real-life to create knives—the stores do not usually make the names themselves, but get them from a factory.
- Essentially, using factory objects means that you have **cut out redundant code and made the software easier to modify**, particularly if there are multiple clients that want to instantiate the same set of classes.

## Factory Method

- The Factory Method pattern does not use a Factory object to create the objects, instead, the Factory Method uses a separate “method” in the same class to create objects.
- The power of Factory Methods come in particular from how they create specialized product objects.
- Generally, in order to create a “specialized” product object, a Factory Object approach would subclass the factory class.

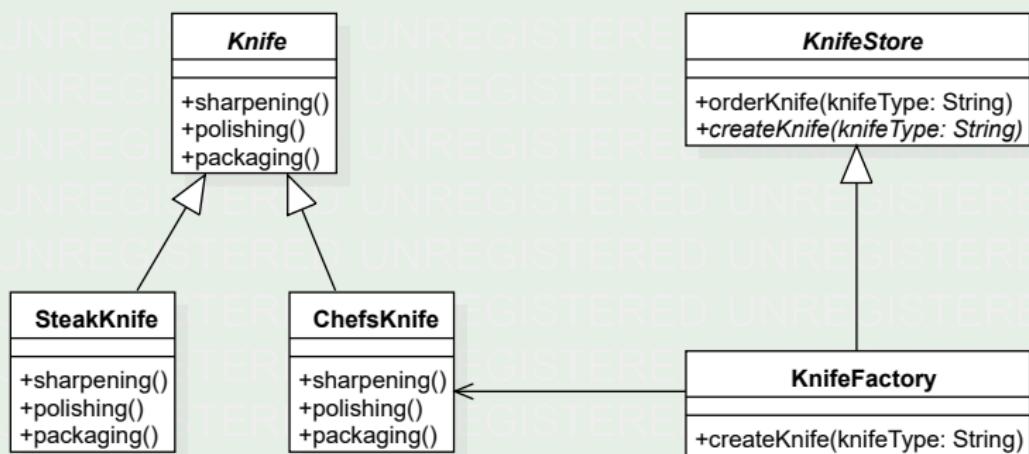
For example,

a subclass of the `KnifeStore` Factory called `KnifeFactory` would make `ChefsKnife` and `SteakKnife` product objects.

## Factory Method (contd.)

- A Factory Method approach, however, would use a KnifeFactory subclass of KnifeStore. The KnifeFactory has a method—the “Factory Method”—that is responsible for creating ChefsKnife and SteakKnife product objects instead.
- This design pattern’s intent is to define an interface for creating objects, but let the subclasses decide which class to instantiate.
- So, instead of working with a factory object, we specialize or subclass the class that uses the Factory Method.
- Each subclass must define its own Factory Method. This is known as **letting the subclasses decide** how objects are made.

## Factory Method: Example (contd.)



## Factory Method: Example (contd.)

- In the above diagram, the “Knife” and “KnifeStore” classes are italicized to indicate that they are abstract classes that cannot be instantiated.
- In summary, this diagram indicates to us that `KnifeFactory`, and any other `KnifeStore` subclass defined must have its own `createKnife()` method.
- This structure is the core of the Factory Method design pattern.

## Factory Method: Summary

- A simple factory like this **returns an instance of any one of several possible classes that have a common parent class.**
- The common parent class can be **an abstract class, or an interface.**
- The calling program typically has a way of telling the factory what it wants, and the factory makes the decision which subclass should be returned to the calling program. It then creates an instance of that subclass, and then returns it to the calling program.

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
<b>Abstract Factory</b> Builder <b>Factory Method</b> Prototype <b>Singleton</b>	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

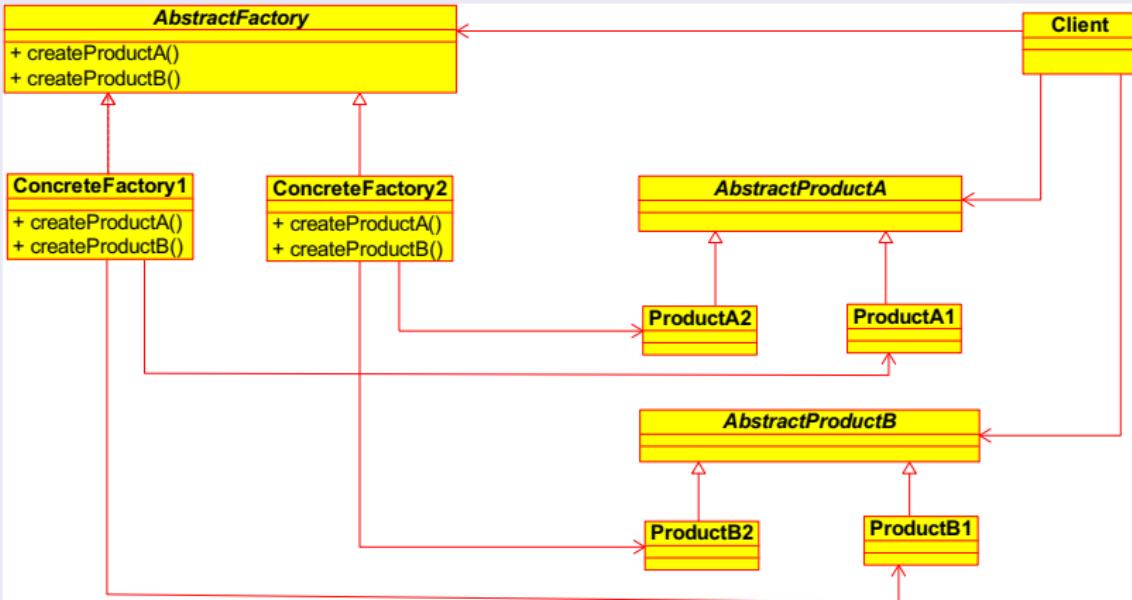
## Abstract Factory: Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Also known as **Kit**

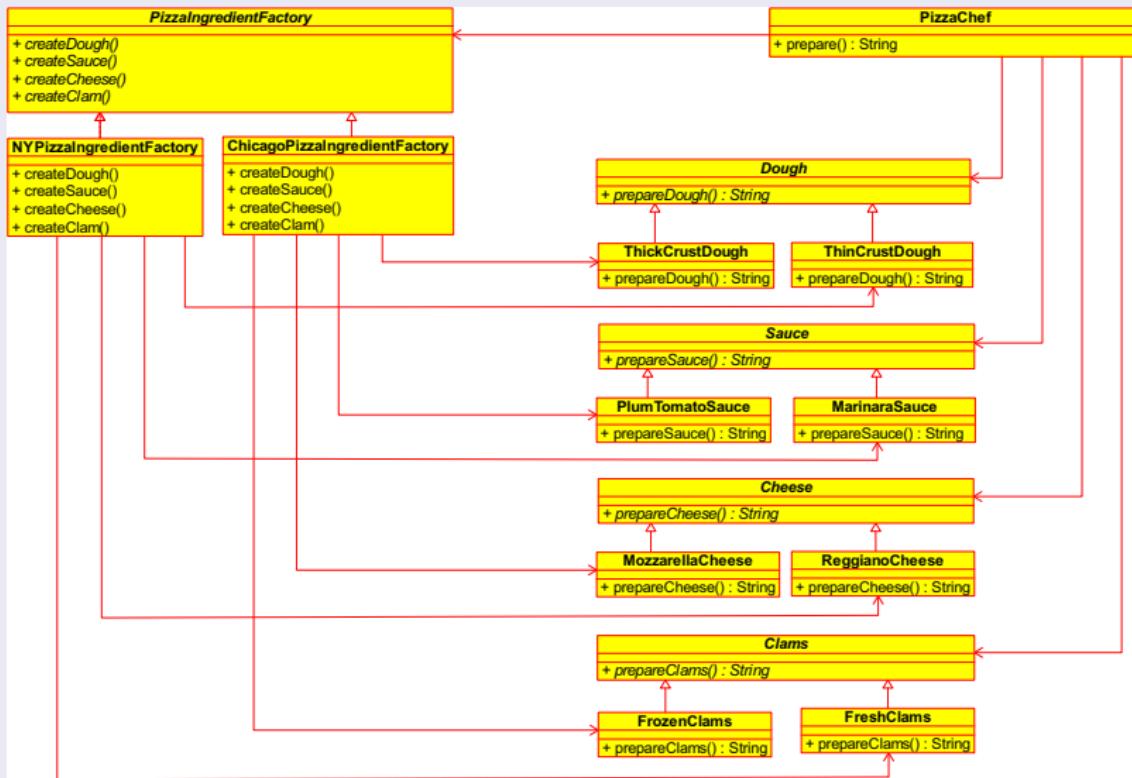
## Abstract Factory: Applicability

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

## Abstract Factory: Structure



## Abstract Factory: Example



## Abstract Factory: Example (contd.)

```
public abstract class PizzaIngredientFactory {  
    public abstract Dough createDough();  
    public abstract Sauce createSauce();  
    public abstract Cheese createCheese();  
    public abstract Clams createClams();  
}
```

## Abstract Factory: Example (contd.)

```
public class NYPizzaIngredientFactory extends PizzaIngredientFactory{  
  
    public ThinCrustDough createDough(){  
        return new ThinCrustDough();  
    }  
  
    public MarinaraSauce createSauce (){  
        return new MarinaraSauce();  
    }  
  
    public ReggianoCheese createCheese(){  
        return new ReggianoCheese();  
    }  
  
    public FreshClams createClams (){  
        return new FreshClams();  
    }  
}
```

## Abstract Factory: Example (contd.)

```
public class ChicagoPizzaIngredientFactory extends  
    PizzaIngredientFactory{  
  
    public ThickCrustDough createDough(){  
        return new ThickCrustDough();  
    }  
  
    public PlumTomatoSauce createSauce (){  
        return new PlumTomatoSauce();  
    }  
  
    public MozzarellaCheese createCheese(){  
        return new MozzarellaCheese();  
    }  
  
    public FrozenClams createClams (){  
        return new FrozenClams();  
    }  
}
```

## Abstract Factory: Example (contd.)

```
public abstract class Dough {  
    abstract String prepareDough();  
}  
  
public class ThickCrustDough extends Dough{  
  
    public String prepareDough()  
    {  
        return "ThickCrust Dough";  
    }  
}  
  
public class ThinCrustDough extends Dough{  
  
    public String prepareDough()  
    {  
        return "ThinCrust Dough";  
    }  
}
```

## Abstract Factory: Example (contd.)

```
public abstract class Sauce {  
    abstract String prepareSauce();  
}  
  
public class PlumTomatoSauce extends Sauce{  
  
    public String prepareSauce()  
    {  
        return "PlumTomato Sauce";  
    }  
}  
  
public class MarinaraSauce extends Sauce{  
  
    public String prepareSauce()  
    {  
        return "Marinara Sauce";  
    }  
}
```

## Abstract Factory: Example (contd.)

```
public abstract class Cheese {  
    abstract String prepareCheese();  
}  
  
public class MozzarellaCheese extends Cheese{  
  
    public String prepareCheese()  
    {  
        return "Mozzarella Cheese";  
    }  
}  
  
public class ReggianoCheese extends Cheese{  
  
    public String prepareCheese()  
    {  
        return "Reggiano Cheese";  
    }  
}
```

## Abstract Factory: Example (contd.)

```
public abstract class Clams {  
    abstract String prepareClams();  
}  
  
public class FrozenClams extends Clams{  
  
    public String prepareClams()  
    {  
        return "Frozen Clams";  
    }  
}  
  
public class FreshClams extends Clams{  
  
    public String prepareClams()  
    {  
        return "Fresh Clams";  
    }  
}
```

## Abstract Factory: Example (contd.)

```
public class PizzaChef {  
  
    private Dough doughType;  
    private Sauce sauceType;  
    private Cheese cheeseType;  
    private Clams clamsType;  
  
    public PizzaChef(PizzaIngredientFactory pizzaIngFac)  
    {  
        doughType = pizzaIngFac.createDough();  
        sauceType = pizzaIngFac.createSauce();  
        cheeseType = pizzaIngFac.createCheese();  
        clamsType = pizzaIngFac.createClams();  
    }  
}
```

## Abstract Factory: Example (contd.)

```
// PizzaChef (Contd.)
public String prepare()
{
    String myDough;
    String mySauce;
    String myCheese;
    String myClams;
    String outputPizza;

    myDough= doughType.prepareDough();
    mySauce = sauceType.prepareSauce();
    myCheese = cheeseType.prepareCheese();
    myClams = clamsType.prepareClams();

    outputPizza = myDough+", "+mySauce+", "+myCheese+", "+myClams;
    return outputPizza;
}
```

## Abstract Factory: Example (contd.)

```
import java.util.Scanner;

public class PizzaCustomer {

    private static PizzaChef myPizzaClient;
    private static PizzaIngredientFactory myPizza;

    public static void main(String a[]){

        System.out.println("What pizza you would like today?: ");
        Scanner in = new Scanner(System.in);
        String pizzaType = in.nextLine();
        String outputPizza;
```

## Abstract Factory: Example (contd.)

```
//PizzaCustomer (contd.)  
  
if(pizzaType.equalsIgnoreCase("NY")){  
    myPizza = new NYPizzaIngredientFactory();  
}  
else if(pizzaType.equalsIgnoreCase("Chicago")){  
    myPizza = new ChicagoPizzaIngredientFactory();  
}  
else{  
    System.out.println("Not a valid pizza type!");  
    return;  
}  
myPizzaClient = new PizzaChef(myPizza);  
outputPizza = myPizzaClient.prepare();  
System.out.println(pizzaType+" is made with "+outputPizza);  
}  
}
```

## Abstract Factory

- Is that a Factory Method lurking inside the Abstract Factory?
  - The job of an Abstract Factory is to define an interface for creating a set of products.
  - Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations.
  - So, factory methods are a natural way to implement your product methods in your abstract factories.

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
<b>Abstract Factory</b> <b>Builder</b> <b>Factory Method</b> <b>Prototype</b> <b>Singleton</b>	Adapter Bridge <b>Composite</b> Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

## Composite: Intent

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

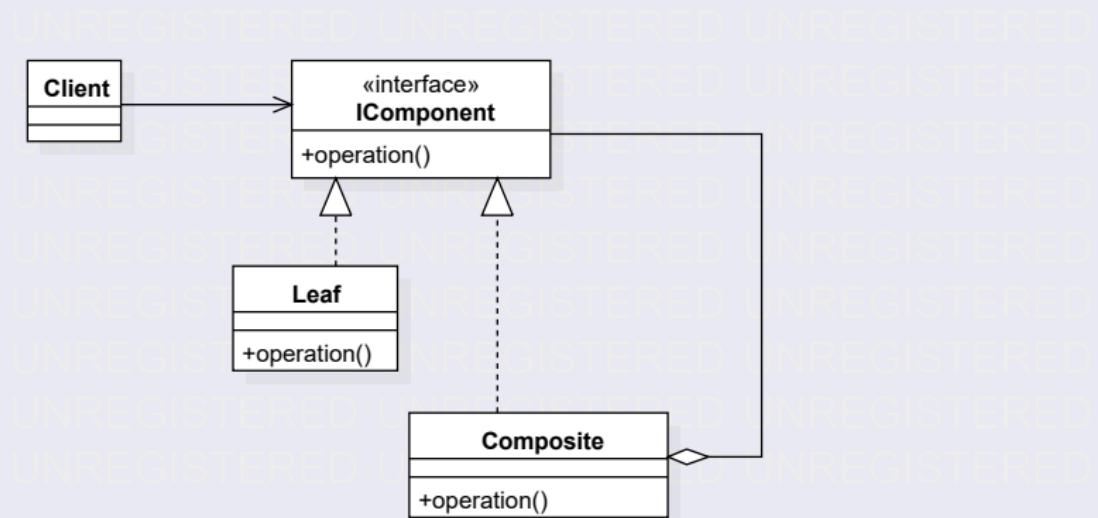
## Composite: Applicability

- Use the Composite pattern when
  - you want to represent part-whole hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

## Composite Pattern

- A composite design pattern is meant to achieve two goals:
  - To compose nested structures of objects, and
  - To deal with the classes for these objects uniformly.

## Composite Pattern: Structure



## Composite Pattern (contd.)

- In this design, a component interface serves as the supertype for a set of classes.
- Using polymorphism, all implementing classes conform to the same interface, allowing them to be dealt with uniformly.

## The Leaf Class and the Composite Class

- A **composite class** is used **to aggregate any class** that implements the component interface.
- The composite class allows you to “traverse through” and “potentially manipulate” the component objects that the composite object contains.
- A **leaf class** represents a non-composite type. It is not composed of other components.

## The Leaf Class and the Composite Class (contd.)

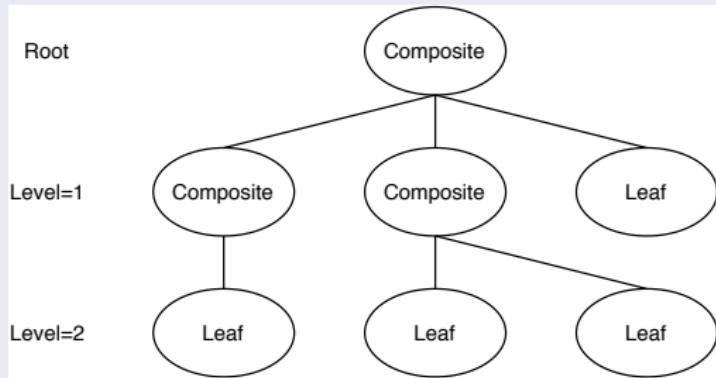
- The Leaf class and the Composite class implement the Component interface, unifying them with a single type.
- This allows us to deal with non-composite and composite objects **uniformly**—the Leaf class and the Composite class are now considered subtypes of Component.
- You may have other composite or leaf classes in practice, but there will only be one overall component **interface or abstract superclass**.

## Recursive Composition

- A composite object can contain other composite object, since the composite class is a subtype of component. This is known as **recursive composition**.
- This term is also a synonym for composite design patterns.

## Composite Design Patterns as Trees

- This design pattern has a composite class with a cyclical nature, which may make it difficult to visualize.
- Instead, it is easier to think of composite design patterns as trees:



## Composite Design Patterns as Trees (contd.)

- The main composite object, which is made up of other component objects, is at the root level of the tree.
- At each level, more components can be added below each composite object, like another composite or a leaf.
- Leaf objects cannot have components added to them.
- Composites therefore have the ability to “grow” a tree, while a leaf ends the tree where it is.

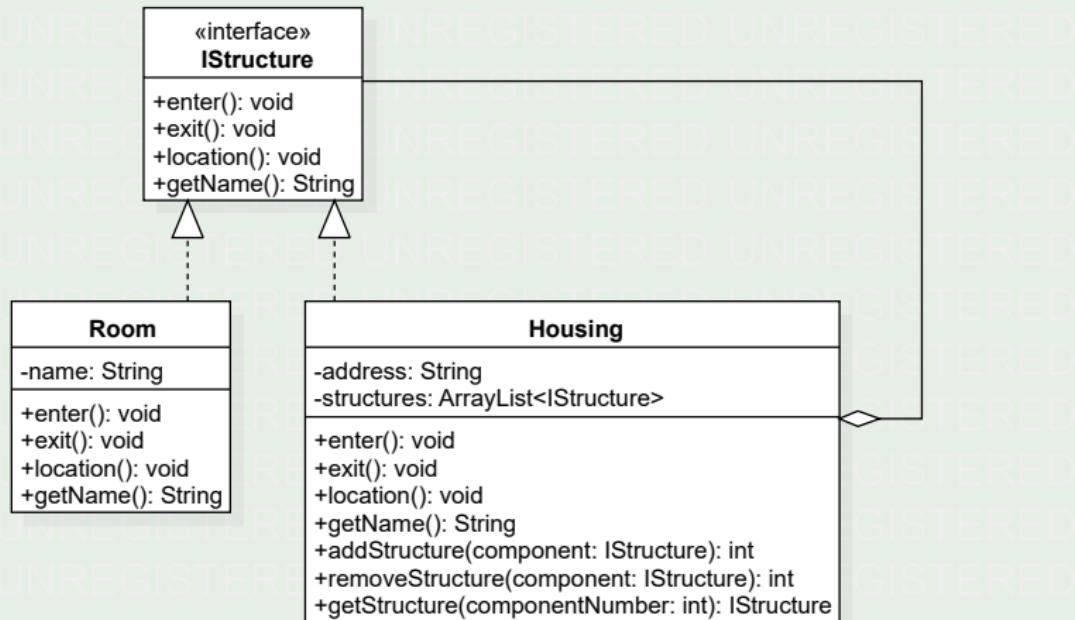
## Composite Design Patterns as Trees (contd.)

- At the beginning of this lecture, we mentioned two issues that composite design patterns are meant to address.
- This tree helps us understand how those goals are met. Individual types of objects, in particular, composite class objects, can aggregate component classes, which creates a tree-like structure.
- As well, each individual class is a subtype of an interface or superclass, and will be able to conform to a set of shared behaviors.

## Composite Pattern (contd.)

- Expressing this in Java can be broken down into steps.
  - ① Design the interface that defines the overall type.
  - ② Implement the composite class.
  - ③ Implement the leaf class
- Let us examine each of these steps using a specific example.

## Composite Pattern: Example



## Composite Pattern: Example (contd.)

- Step 1: Design the interface that defines the overall type

```
public interface IStructure {  
    public void enter();  
    public void exit();  
    public void location();  
    public String getName();  
}
```

## Composite Pattern: Example (contd.)

- Step 2: Implement the composite class

```
public class Housing implements IStructure {  
    private ArrayList<IStructure> structures;  
    private String address;  
  
    public Housing (String address) {  
        this.structures = new ArrayList<IStructure>();  
        this.address = address;  
    }  
  
    public void enter() {  
        System.out.println("You have entered the " + this.address);  
    }  
    public void exit() {  
        System.out.println("You have left the " + this.address);  
    }  
}
```

## Composite Pattern: Example (contd.)

- Step 2: Implement the composite class (contd.)

//Housing (contd.)

```
public void location() {
    System.out.println("You are currently in " + this.getName()
        + ". It has ");
    for (IStructure struct : this.structures)
        System.out.println(struct.getName());
}

public String getName() {
    return this.address;
}

public int addStructure(IStructure component) {
    this.structures.add(component);
    return this.structures.size() - 1;
}
```

## Composite Pattern: Example (contd.)

- Step 2: Implement the composite class (contd.)

//Housing (contd.)

```
public int removeStructure(IStructure component) {  
    this.structures.remove(component);  
    return this.structures.size() - 1;  
}  
  
public IStructure getStructure(int componentNumber) {  
    return this.structures.get(componentNumber);  
}  
}
```

## Composite Pattern: Example (contd.)

- Step 3: Implement the leaf class

```
public class Room implements IStructure {  
    private String name;  
  
    Room(String name) {  
        this.name = name;  
    }  
    public void enter() {  
        System.out.println("You have entered the " + this.name);  
    }  
    public void exit() {  
        System.out.println("You have left the " + this.name);  
    }  
    public void location() {  
        System.out.println("You are currently in the " + this.name);  
    }  
    public String getName() {  
        return this.name;  
    }  
}
```

## Composite Pattern: Example (contd.)

```
public class CompositeMain {  
  
    public static void main(String[] args) {  
        Housing building = new Housing("123 Street");  
  
        Housing floor1 = new Housing("123 Street - First Floor");  
  
        int firstFloor = building.addStructure(floor1);  
  
        Room washroom1m = new Room("1F Men's Washroom");  
        Room washroom1w = new Room("1F Women's Washroom");  
        Room common1 = new Room("1F Common Area");  
  
        int firstMens = floor1.addStructure(washroom1m);  
        int firstWomans = floor1.addStructure(washroom1w);  
        int firstCommon = floor1.addStructure(common1);  
  
        building.enter(); // Enter the building  
        building.location();  
    }  
}
```

## Composite Pattern: Example (contd.)

```
//CompositeMain (contd.)  
  
Housing currentfloor =  
    (Housing)building.getStructure(firstFloor);  
currentfloor.enter(); // Walk into the first floor  
currentfloor.location();  
  
Room currentRoom = (Room) currentfloor.getStructure(firstMens);  
currentRoom.enter(); // Walk into the men's room  
currentRoom.exit(); // Exit from the men's room  
  
currentRoom = (Room) currentfloor.getStructure(firstCommon);  
currentRoom.enter(); // Walk into the common area  
currentRoom.exit(); // Exit from the common area  
currentfloor.exit(); // Exit from the first floor  
  
floor1.removeStructure(common1);  
floor1.removeStructure(firstMens);  
currentfloor.enter(); // Walk into the first floor  
currentfloor.location();  
}
```

## Composite Pattern: Summary

- A composite design pattern allows you to build a tree-like structure of objects, and to treat individual types of those objects **uniformly**. This is achieved by:
  - Enforcing polymorphism across each class through implementing an interface (or inheriting from a superclass).
  - Using a technique called recursive composition which allows objects to be composed of other objects that are of a common type.
  - Composite design patterns apply the design principles of decomposition and generalization.
  - They break a whole into parts, but have the whole and parts both conform to a common type.
  - Complex structures can be built using composite objects and leaf objects which belong to a unified component type.

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
<b>Abstract Factory</b> <b>Builder</b> <b>Factory Method</b> <b>Prototype</b> <b>Singleton</b>	<b>Adapter</b> <b>Bridge</b> <b>Composite</b> <b>Decorator</b> <b>Facade</b> <b>Flyweight</b> <b>Proxy</b>	<b>Chain of Responsibility</b> <b>Command</b> <b>Interpreter</b> <b>Iterator</b> <b>Mediator</b> <b>Memento</b> <b>Observer</b> <b>State</b> <b>Strategy</b> <b>Template Method</b> <b>Visitor</b>

## Facade Pattern: Intent

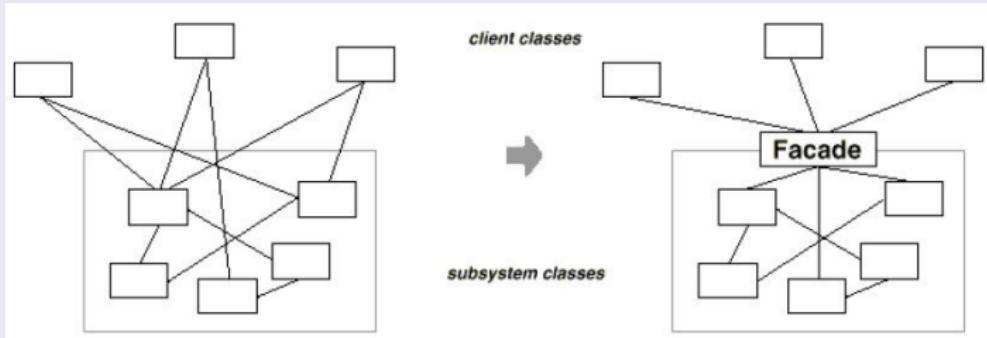
- Provide a unified interface to a set of interfaces in a subsystem.  
Facade defines a higher-level interface that makes the subsystem easier to use.

## Facade Pattern: Applicability

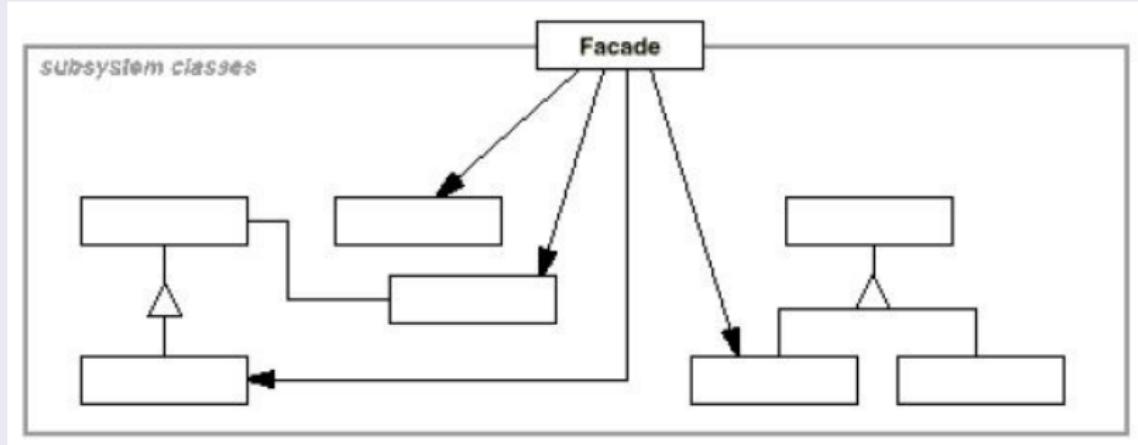
- Use the Facade pattern when
  - you want **to provide a simple interface to a complex subsystem.** Subsystems often get more complex as they evolve.
  - there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby **promoting subsystem independence and portability.**
  - you want **to layer your subsystems.** Use a facade **to define an entry point to each subsystem level.**

## Facade Pattern: Motivation

- Structuring a system into subsystems helps reduce complexity.
- A common **design goal** is to minimize the communication and dependencies between subsystems.
- One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.



## Facade Pattern: Structure



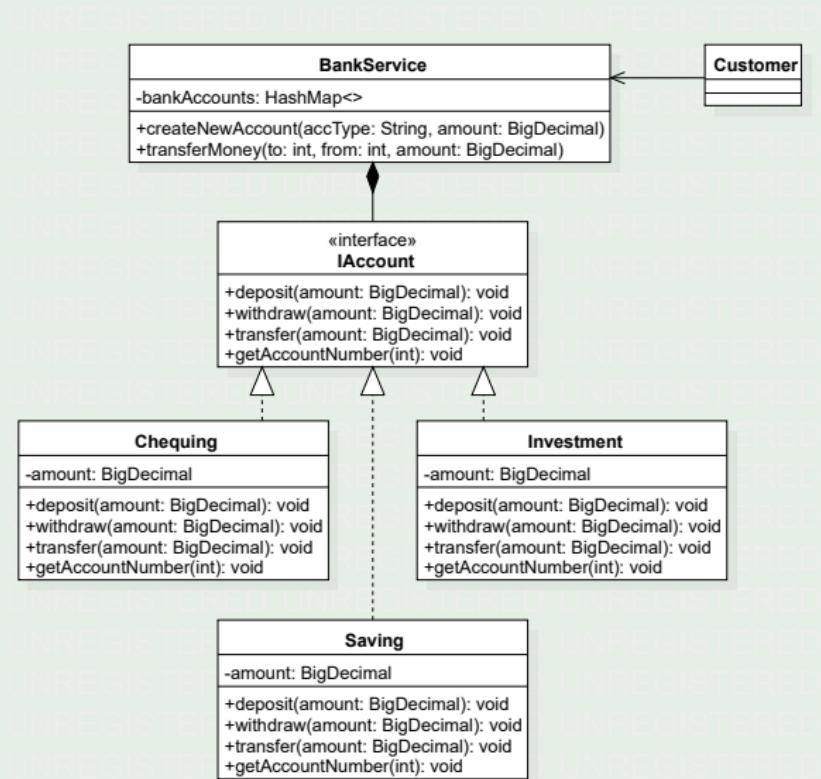
## Facade Pattern

- A facade is a **wrapper class** that encapsulates a subsystem in order to hide the subsystem's complexity, and acts as a point of entry into a subsystem without adding more functionality in itself.
- The wrapper class allows a client class to interact with the subsystem through the facade.
- A facade might be compared metaphorically to a waiter or salesperson, who hide all the extra work to be done in order to purchase a good or service.
- Often facade design patterns combine interface implementation by one or more classes, which then gets wrapped by the facade class.

## Facade Pattern (contd.)

- This can be explained through a number of steps.
  - ① Design the interface
  - ② Implement the interface with one or more classes
  - ③ Create the facade class and wrap the classes that implement the interface
  - ④ Use the facade class to access the subsystem
- Let us examine each of these steps with an example for a bank system.

## Facade Pattern: Example



## Facade Pattern: Example

- Step 1: Design the Interface

```
//IAccount
import java.math.BigDecimal;

public interface IAccount
{
    public void deposit(BigDecimal amount);
    public void withdraw(BigDecimal amount);
    public void transfer(BigDecimal amount);
    public int getAccountNumber();
}
```

## Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes

```
//Chequing
import java.math.BigDecimal;

public class Chequing implements IAccount
{
    private BigDecimal amount;
    public Chequing(BigDecimal initAmount) {
        this.amount = initAmount;
    }

    @Override
    public void deposit(BigDecimal amount) {
        System.out.println("Hi, This is DEPOSIT from CHEQUING
                           account!");
    }
}
```

## Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Chequing (contd.)  
    @Override  
    public void withdraw(BigDecimal amount) {  
        System.out.println("Hi, This is WITHDRAW from CHEQUING  
                           account!");  
    }  
  
    @Override  
    public void transfer(BigDecimal amount) {  
        System.out.println("Hi, This is TRANSFER from CHEQUING  
                           account!");  
    }  
  
    @Override  
    public int getAccountNumber() {  
        return 100001;  
    }  
}
```

## Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Saving
import java.math.BigDecimal;

public class Saving implements IAccount
{
    private BigDecimal amount;
    public Saving(BigDecimal initAmount) {
        this.amount = initAmount;
    }

    @Override
    public void deposit(BigDecimal amount) {
        System.out.println("Hi, This is DEPOSIT from SAVING
                           account!");
    }
}
```

## Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Saving (contd.)  
    @Override  
    public void withdraw(BigDecimal amount) {  
        System.out.println("Hi, This is WITHDRAW from SAVING  
                           account!");  
    }  
  
    @Override  
    public void transfer(BigDecimal amount) {  
        System.out.println("Hi, This is TRANSFER from SAVING  
                           account!");  
    }  
  
    @Override  
    public int getAccountNumber() {  
        return 200001;  
    }  
}
```

## Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Investment
import java.math.BigDecimal;

public class Investment implements IAccount
{
    private BigDecimal amount;
    public Investment(BigDecimal initAmount) {
        this.amount = initAmount;
    }

    @Override
    public void deposit(BigDecimal amount) {
        System.out.println("Hi, This is DEPOSIT from INVESTMENT!");
    }
}
```

## Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Investment (contd.)  
    @Override  
    public void withdraw(BigDecimal amount) {  
        System.out.println("Hi, This is WITHDRAW from INVESTMENT!");  
    }  
  
    @Override  
    public void transfer(BigDecimal amount) {  
        System.out.println("Hi, This is TRANSFER from INVESTMENT!");  
    }  
  
    @Override  
    public int getAccountNumber() {  
        return 300001;  
    }  
}
```

## Facade Pattern: Example (contd.)

- Step 3: Create the facade class and wrap the classes that implement the interface

```
import java.math.BigDecimal;
import java.util.HashMap;
//BankService
public class BankService
{
    private HashMap<Integer, IAccount> bankAccounts;

    public BankService()
    {
        this.bankAccounts = new HashMap<>();
    }
}
```

## Facade Pattern: Example (contd.)

- Step 3: Create the facade class and wrap the classes that implement the interface (contd.)

```
//BankService (contd)
public int createNewAccount(String type, BigDecimal initAmount)
{
    IAccount newAccount = null;
    switch (type) {
        case "chequing":
            newAccount = new Chequing(initAmount);
            break;
        case "saving":
            newAccount = new Saving(initAmount);
            break;
        case "investment":
            newAccount = new Investment(initAmount);
            break;
        default:
            System.out.println("Invalid account type");
            break;
    }
}
```

## Facade Pattern: Example (contd.)

- Step 3: Create the facade class and wrap the classes that implement the interface (contd.)

```
//BankService (contd)
if (newAccount != null)
{
    this.bankAccounts.put(newAccount.getAccountNumber(),
        newAccount);
    return newAccount.getAccountNumber();
}
return -1;
}

public void transferMoney(int to, int from, BigDecimal amount) {
    IAccount toAccount = this.bankAccounts.get(to);
    IAccount fromAccount = this.bankAccounts.get(from);
}
}
```

## Facade Pattern: Example (contd.)

- Step 4: Use the facade class to access the subsystem

```
//Customer
import java.math.BigDecimal;

public class Customer
{
    public static void main(String[] args)
    {
        BankService myBankService = new BankService();
        int mySaving = myBankService.createNewAccount("saving", new
            BigDecimal(500.00));
        System.out.println("New saving account created with account
            number: "+mySaving);
        int myInvestment =
            myBankService.createNewAccount("investment", new
            BigDecimal(1000.00));
        System.out.println("New investment account created with
            account number: "+myInvestment);
    }
}
```

## Facade Pattern: Example (contd.)

- Step 4: Use the facade class to access the subsystem (contd.)

```
//Customer (contd)
myBankService.transferMoney(mySaving, myInvestment, new
    BigDecimal(300.00));
System.out.println("Money transferred from saving account
    "+mySaving+" to investment account: "+ myInvestment);
}
}
```

## Facade Pattern: Summary

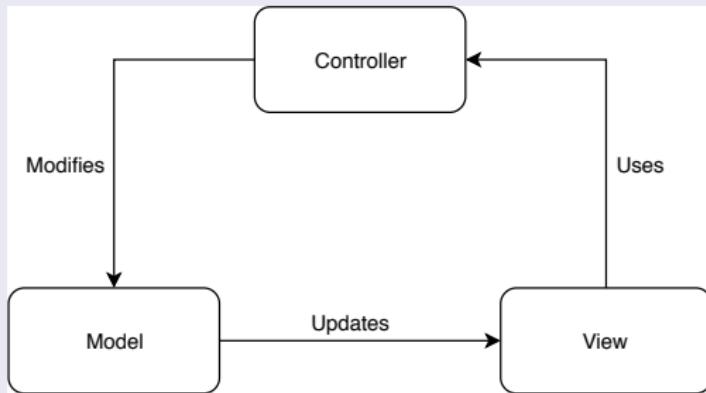
- Is a means to hide the complexity of a subsystem by encapsulating it behind a unifying wrapper called a facade class.
- Removes the need for client classes to manage a subsystem on their own, resulting in less coupling between the subsystem and the client classes.
- Handles instantiation and redirection of tasks to the appropriate class within the subsystem.
- Provides client classes with a simplified interface for the subsystem.
- Acts simply as a point of entry to a subsystem and does not add more functional the subsystem.

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## MVC Pattern

- MVC stands for Model View Controller.
- Model, View, Controller (MVC) patterns are a pattern that should be **consider for use with user interfaces**.
- MVC patterns **divides the responsibilities** of a system that offers a user interface into three parts: model, view, and controller.



## MVC Pattern (contd.)

- The **Model** is going to contain the underlying data, state, and logic that users want to see and manipulate through an interface. It is the “back end”, or the underlying software.
- A **key aspect** of the MVC pattern is that the model is self-contained. It has all of the state, methods, and other data needed to exist on its own.
- The **View** gives a user the way to see the model in the way they expect, and allows them to interact with it, or at least parts of it. It is the “front end” or the presentation layer of the software.
- A **model** could have several views that present different parts of the model, or present the model in different ways.

## MVC Pattern (contd.)

- When a value changes in the model, the view needs to be notified so it can update itself accordingly.
- The observer design pattern allows this to happen. In an observer pattern, observers are notified when the state of the subject changes.
- In this case, the view is an observer. When the model changes, it notifies all of the views that are subscribed to it.

## MVC Pattern (contd.)

- The **view** may also present users with ways to make changes to the data in the underlying model. It does not directly send requests to the model, however. Instead, information about the user interaction is passed to a Controller.
- The **controller** is responsible for interpreting requests and interaction with elements in the view, and changing the model.
- The **view** is therefore only responsible for the visual appearance of the system.
- The **model** focuses only on managing the information for the system.

## MVC Pattern (contd.)

- The MVC pattern uses the **separation of concerns** design principle to divide up the main responsibilities in an interactive system. The controller ensures that the views and the model are loosely coupled.
- The **model** corresponds to entity objects, which are derived from analyzing the problem space for the system.
- The **view** corresponds to a boundary object, which is at the edge of your system that deals with users.
- The **controller** corresponds to a control object, which receives events and coordinates actions.

## MVC Pattern (contd.)

For example,

- Imagine you are creating an interface for a grocery store, where cashiers can enter orders, and they are displayed.
- Customers and cashiers should be able to see the list of items entered into the order with a barcode scanner, and see the total bill amount.
- Cashiers should also be able to make corrections if necessary.

## MVC Pattern (contd.)

- Controllers make the code better in the following ways:
  - The view can **focus on its main purpose**: presenting the user interface, as the controller takes the responsibility of interpreting the input from the user and working with the model based on that input.
  - The view and the model **are not “tightly coupled”** when a controller is between them. Features can be added to the model and tested long before they are added to the view.
- Controllers make the code **cleaner and easier to modify**.

## MVC Pattern: Summary

- The MVC pattern is particularly important because of its use of the **separation of concerns**.
- It allows the program to be **modular and loosely coupled**.
- The view and the model can change, without needing knowledge of the other.

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## Anti-Patterns & Code Smells

- No matter how well you design your code, there will still be changes that need to be made.
- **Refactoring** helps manage this.
  - It is the process of making changes to your code so that the external behaviors of the code are not changed, but the internal structure is improved.
  - This is done by making small, incremental changes to the code structure and testing frequently to make sure these changes have not altered the behavior of the code.

## Anti-Patterns & Code Smells

- Ideally, refactoring changes are made when features are added, and not when the code is complete. This saves time, and makes adding features easier.
- Changes are needed in code when bad code emerges. Just like patterns emerge in design, bad code can emerge as patterns as well. These are known as **anti-patterns** or **code smells**.
- Code smells help “sniff out” what is bad in the code.

## Anti-Patterns & Code Smells (contd.)

- Comments
- Duplicate Code
- Long Method
- Large Class
- Data Class
- Data Clumps
- Long Parameter List
- Divergent Class
- Shotgun Surgery
- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Primitive Obsession
- Switch Statements
- Speculative Generality
- Refused Request

## Comments

- One of the most common examples of bad code is comments. This code smell can occur between two extremes.
  - If **no comments** are provided in the code, it can be hard for someone else, or even the original developer returning to the code after some time away, to understand what the code is doing or should be doing.
  - On the other hand, if there are **too many comments**, they might get out of sync as the code changes. Comments can be a “deodorant” for bad smelling code. The use of a lot of comments to explain complicated design can indicate that bad design is being covered up.
- There are other ways comments can indicate bad code, though. If the comments take on a “**reminder**” nature, so they indicate something that needs to be done, or that if a change is made to one section in the code, it needs to be updated in another method, this indicates bad code.

## Comments (contd.)

- Comments might reveal that the programming language selected for the software is not appropriate. This could happen if the programming language does not support the design principles being applied.
- Comments are very useful for documenting application programmer interfaces (APIs) in the system, for documenting the rationale for a particular choice of data structure or algorithm, and make it easier for others to understand and use the code.

## Duplicate Code

- Duplicated code occurs **when blocks of code exist in the design that are similar, but have slight differences**. These blocks of code appear in multiple places in the software.
- This can be a problem, because if something needs to change, then the code needs to be updated in multiple places. This applies to adding functionalities, updating an algorithm, or fixing a bug.
- Instead, if the code only needed to be updated in one location, it is easier to implement the change. It also reduces the chance that a block of code was missed in an update or change.
- This anti-pattern relates to the D.R.Y. principle, or “Don’t Repeat Yourself”, which suggests that programs should be written so that they can perform the same tasks but with less code.

## Long Method

- The long method anti-pattern suggests that code should not have long methods. Long methods can indicate that the method is more complex or has more occurring within it than it should.
- Determining if a code is too long can be difficult. There is even some debate if length of code is even a good measure of code complexity.
- Some methods, such as setting up a user interface, can be naturally long, even if focused on a specific task. Sometimes a long method is appropriate.

## Long Method (contd.)

- This anti-pattern may also depend on the programming language for the system.
- Some developers suggest that having an entire method visible at once on the screen is a good guideline, with no more than around 50 lines of code.
- However, some studies show that programmers can handle methods of a couple hundred lines before they introduce bugs! Determining “how long is too long” for a method isn’t always a straightforward process!

## Large Class

- The large class anti-pattern suggests that classes should not be too large.
- Large classes are commonly referred to as **God classes**, **Blob classes**, or **Black Hole classes**. They are classes that continue to grow and grow, although they typically start out as regular-sized.
- Large classes occur when more responsibilities are needed, and these classes seem like the appropriate place to put the responsibilities.
- This growth will require extensive comments to document where in the code of the class certain functionalities exist.
- Classes should have an explicit purpose to keep the class cohesive, so it does one thing well. If a functionality is not specific to the class' responsibility, it may be better to place it elsewhere.

## Data Class

- The data class anti-pattern is on the opposite end of the spectrum of the large class. It occurs when there is too small of a class. These are referred to as **data classes**.
- Data classes are classes that **contain only data and no real functionality**. These classes usually have only getter and setter methods, but not much else. This indicates that it may not be a good abstraction or a necessary class.

## Data Clumps

- **Data clumps** are groups of data appearing together in the instance variables of a class, or parameters to methods.
- Consider this code smell through an example. Imagine a method with integer variables  $x$ ,  $y$ , and  $z$ .

```
public void doSomething (int x, int y, int z) {  
    //...  
}
```

- If there are many methods in the system that perform various manipulations on the variables, it is better to have an object as the parameter, instead of using the variables as parameters over and over again. That object can be used in their place as a parameter.

## Data Clumps (contd.)

```
public class Point3D {  
    private int x;  
    private int y;  
    private int z;  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    public int getZ() {  
        return z;  
    }  
}  
  
//contd.  
public void setX(int newX) {  
    x = newX;  
}  
public void setY(int newY) {  
    y = newY;  
}  
public void setZ(int newZ) {  
    z = newZ;  
}
```

- Be careful not to just create data classes, however. The classes should do more than just store data.
- The original `doSomething()` method, or a useful part it, might be added to the `Point3D` class to avoid this.

## Long Parameter List

- Another code smell is having **long parameter lists**. A method with a long parameter list can be difficult to use. They increase the chance of something going wrong.
- Methods with long parameter lists require extensive comments to explain what each of the parameters does and what it should be.
- If long parameter lists are not commented, however, then it may be necessary to look inside the implementation to see how they are used. This breaks encapsulation.
- The best solution for long parameter lists is to introduce parameter objects. A parameter object captures context.

## Divergent Class

- Some code smells occur when making changes to the code itself. A divergent change is one such code smell. It occurs when you have to change a class in many different ways, for many different reasons.
- This relates to the large class code smell, where a large class has many different responsibilities. **Poor separation of concerns** is therefore a common cause of divergent change.
- Classes should have only one specific purpose. This reduces the number of reasons the code would need to change, and reduce the variety of changes needed to be implemented.
- Separation of concerns resolves two code smells—large class and divergent change.

## Shotgun Surgery

- **Shotgun surgery** is a code smell that occurs when a change needs to be made to one requirement, and a numerous classes all over the design need to be touched to make that one change. In good design, a small change is ideally localized to one or two places (although this is not always possible).
- This is a commonly occurring code smell. It can happen if you are trying to add a feature, adjust code, fix bugs, or change algorithms.

## Shotgun Surgery (contd.)

- Modular code is not always an option, however. Some changes require shotgun surgery no matter how well designed the code.
- The shotgun surgery smell is normally resolved by moving methods around.
- If a change in one place leads to changes in other places, then this indicates that the methods are related in some way. Perhaps there is a better way to organize them. If not, then you may have to deal with it as it is.

## Feature Envy

- **Feature envy** is a code smell that occurs when there is a method that is more interested in the details of a class other than the one it is in.
- If two methods or classes are always talking to one another and seem as if they should be together, then chances are this is true.

## Inappropriate Intimacy

- **Inappropriate intimacy** is a code smell that occurs when two classes depend too much on one another through two-way communication.
- If two classes are closely coupled, so a method in one class calls methods of the other, and vice versa, then it is likely necessary to remove this cycle.
- Methods should be factored out so both classes use another class. At the very least, this makes communication one-way, and should create looser coupling.
- Cycles are not always necessarily a bad thing. Sometimes, they are necessary. But, if there's a way to make the design simpler and easier to understand, then it is a good solution.

## Message Chains

- A **message chain** occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.
- It also potentially violates the Law of Demeter (or Principle of Least Knowledge is a design guideline), which specify which methods are allowed to be called.
- Long chains of calls could be appropriate, however, if they return a limited set of objects that methods are allowed to be called on. If those objects follow the Law of Demeter, then the chain is appropriate.

## Primitive Obsession

- **Primitive obsession** is a code smell that occurs when you rely on the use of built-in types too much. Built-in types, or primitives, are things like ints, longs, floats, or strings. Although there will be need of them in the code, they should only exist at the lowest levels of the code.
- Overuse of primitive types occurs when abstractions are not identified, and suitable classes are not defined.
- If you are using primitive types often at a high-level, then it is a good indicator that suitable classes are not being declared, and there is a primitive obsession code smell in the system.

## Switch Statements

- **Switch statements** are a code smells that occur when switch statements are scattered throughout a program. If a switch is changed, then the others must be found and updated as well.
- Although there can be a need for long if/else statements in the code, sometimes switch statements may be handled better.

For example,

if conditionals are checking on type codes, or the types of something, then there is a better way of handling the switch statements.

- It may be possible to reduce conditionals down to a design that uses polymorphism.

## Speculative Generality

- The code smell **speculative generality** occurs when you make a superclass, interface, or code that is not needed at the time, but that may be useful someday. This practice introduces generality that may not actually help the code, but “over-engineers” it.
- In Agile development, it best to practice **Just in Time Design**. This means that there should be just enough design to take the requirements for a particular iteration to a working system. This means that all that needs to be designed for are the set of requirements chosen at the beginning of an iteration.

## Speculative Generality (contd.)

- Software changes frequently. Clients can change their mind at any time, and drop requirements. So your design should stay simple, and time should not be lost on writing code that may never been used.
- If generalization is necessary, then it should be done. This change may take longer at the time, compared to if it is set up for beforehand, but it is better than writing code that may not be needed down the line.

## Refused Request

- A **refused request** code smell occurs when a subclass inherits something but does not need it.
- If a superclass declares a common behavior across subclasses, and subclasses are inheriting things they do not need or use, then they may not be appropriate subclasses for the superclass.
- Instead, it may make more sense for a stand-alone class. Or perhaps the unwanted behaviors should not be defined in the superclass.
- Finally, if only some subclasses use them, then it may be better to define those behaviors in the subclasses only.

## Anti-Patterns & Code Smells: Summary

- Code smells help identify bad code and bad design. It is good practice to review your code frequently for code smells to ensure that the code remains reusable, flexible, and maintainable.

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

- Let's say you have a pizza shop, and as a cutting-edge pizza store owner in **Objectville** you might end up writing some code like this:

```
Pizza orderPizza() {  
    Pizza pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
  
    return pizza;  
}
```

- But you need more than one type of pizza.....

```
Pizza orderPizza() {  
    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;  
}
```

- But the pressure is on to add more pizza types .....

```
Pizza orderPizza() {  
    Pizza pizza;  
  
    if (type.equals("cheese")) {  
        pizza = new CheesePizza();  
    } else if (type.equals("greek")) { //this variety is no  
        longer available  
    } else if (type.equals("pepperoni")) {  
        pizza = new PepperoniPizza();  
    } else if (type.equals("clam")) { //two more varieties are  
        added namely- clam and veggie  
        pizza = new ClamPizza();  
    } else if (type.equals("veggie")) {  
        pizza = new VeggiePizza();  
    }  
  
    //code continued to next the slide
```

- But the pressure is on to add more pizza types (contd.) .....

```
//code continued from the previous slide
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

- Encapsulating object creation

```
Pizza orderPizza() {  
    Pizza pizza;  
  
    // At first we pull the object creation code out of the  
    // orderPizza Method. Then we place that code in an object  
    // that is only going to worry about how to create pizzas.  
    // If any other object needs a pizza created, this is the  
    // object to come to. Refer to the two previous slides  
    // what we have here earlier. We've got a name for this  
    // new object: we call it a Factory. Now See the next  
    // slide...  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
  
    return pizza;  
}
```

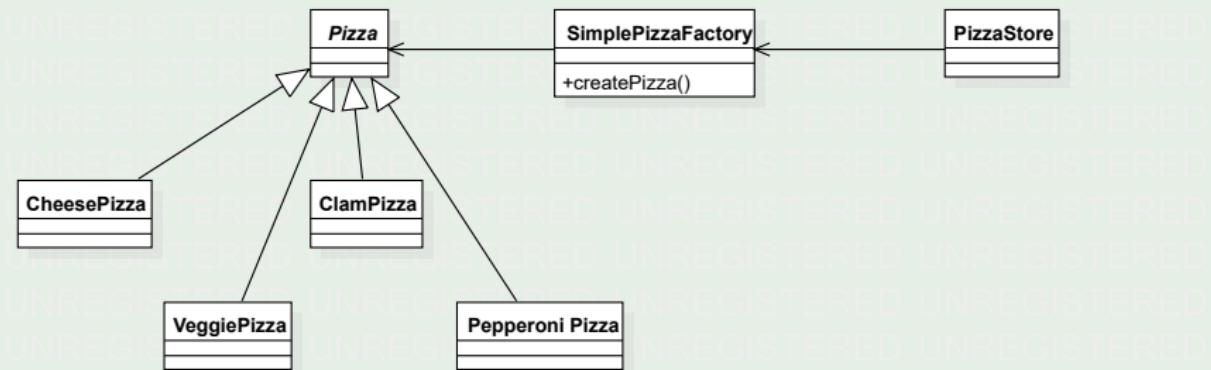
- Building a simple pizza factory

```
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;  
    }  
}
```

- Reworking the PizzaStore class

```
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 methods here  
}
```

- The Simple Factory defined—Just because Simple Factory isn't a REAL pattern doesn't mean we shouldn't check out how it's put together.
  - Let's take a look at the class diagram of our new Pizza Store:



- Franchising the pizza store
- Your **Objectville PizzaStore** has done so well that you've trounced the competition and now everyone wants a PizzaStore in their own neighborhood.
- As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code.
- But what about regional differences?
- Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, to name a few), depending on where the franchise store is located and the tastes of the local pizza connoisseurs.

- One approach.....
- If we take out `SimplePizzaFactory` and create three different factories, `NYPizzaFactory`, `ChicagoPizzaFactory` and `CaliforniaPizzaFactory`, then we can just compose the `PizzaStore` with the appropriate factory and a franchise is good to go. That's one approach.
- Let's see what that would look like.....

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
// Here we create a factory for making NY style pizzas.

PizzaStore nyStore = new PizzaStore(nyFactory);
//Then we create a PizzaStore and pass it a reference to the NY factory.

nyStore.order("Veggie");
//...and when we make pizzas, we get NY-styled pizzas.

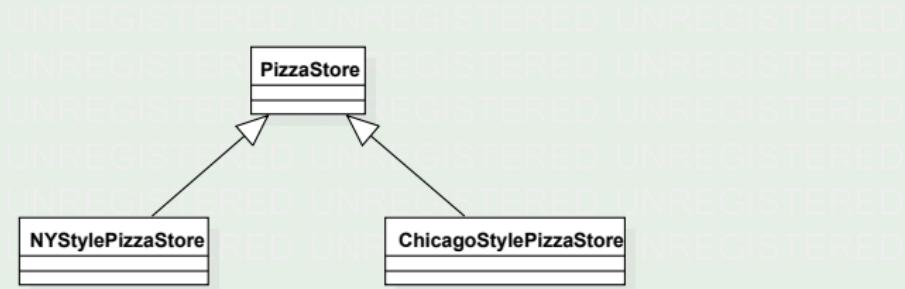
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
//Likewise for the Chicago pizza stores: we create a factory for
Chicago pizzas and create a store that is composed with a Chicago
factory. When we make pizzas, we get the Chicago flavored ones
```

- Rethinking the problem a bit, you see that what you'd really like to do is create a framework that ties the store and the pizza creation together, yet still allows things to remain flexible.
- In our early code, before the `SimplePizzaFactory`, we had the pizza-making code tied to the `PizzaStore`, but it wasn't flexible.

- A framework for the pizza store
- First, let's look at the changes to the PizzaStore:

```
public abstract class PizzaStore {  
    //PizzaStore is now abstract  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
  
        pizza = createPizza(type); //Now createPizza is back to  
            being a call to a method in the PizzaStore rather than  
            on a factory object.  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
    abstract Pizza createPizza(String type);  
    //Our "factory method" is now abstract in PizzaStore.  
}
```

- Allowing the subclasses to decide



- Let's make a PizzaStore: NYPizzaStore

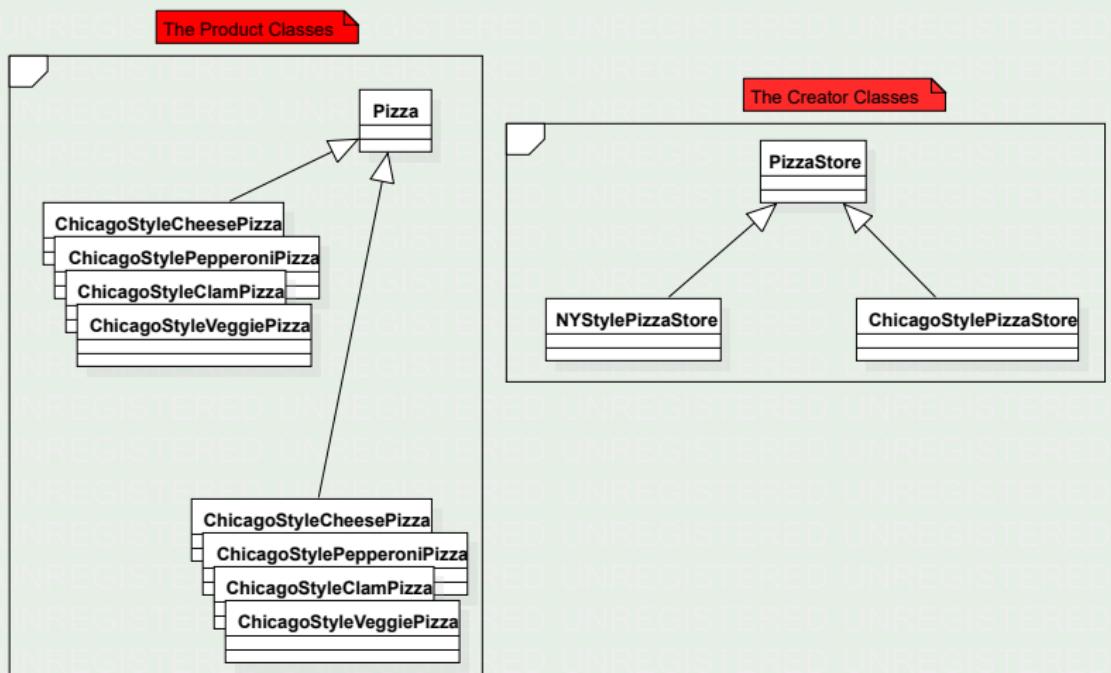
```
public class NYPizzaStore extends PizzaStore {  
  
    Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new NYStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new NYStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new NYStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new NYStylePepperoniPizza();  
        } else return null;  
    }  
}
```

- Let's make a PizzaStore: ChicagoPizzaStore

```
public class ChicagoPizzaStore extends PizzaStore {  
  
    Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new ChicagoStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new ChicagoStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new ChicagoStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new ChicagoStylePepperoniPizza();  
        } else return null;  
    }  
}
```

- You've waited long enough, time for some pizzas!

- It's finally time to meet the Factory Method Pattern (**See Lecture-7.2**)



- Meanwhile, back at the PizzaStore.....
- Ensuring consistency in your ingredients

## Chicago Pizza Menu vs New York PizzaMenu

- **Cheese Pizza:** Plum Tomato Sauce, Mozzarella, Parmesan, Oregano
- **Veggie Pizza:** Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives
- **Clam Pizza:** Plum Tomato Sauce, Mozzarella, Parmesan, Clams
- **Pepperoni Pizza:** Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives, Pepperoni
- **Cheese Pizza:** Marinara Sauce, Reggiano, Garlic
- **Veggie Pizza:** Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers
- **Clam Pizza:** Marinara Sauce, Reggiano, Fresh Clams
- **Pepperoni Pizza:** Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers, Pepperoni

- Families of ingredients. ....

- Building the ingredient factories

- Building the New York ingredient factory

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## SOLID

- SOLID is a collection of best-practice, object-oriented design principles which can be applied to your design, allowing you to accomplish various desirable goals such as low-coupling, high cohesion, etc.

SOLID is an acronym for the following principles

**S:** Single Responsibility Principle (SRP)

- A class should have one, and only one, reason to change.

**O:** Open-Closed Principle (OCP)

- You should be able to extend a classes behavior, without modifying it.

**L:** Liskov Substitution Principle (LSP)

- Derived classes must be substitutable for their base classes.

**I:** Interface Segregation Principle (ISP)

- Make fine grained interfaces that are client specific.

**D:** Dependency Inversion Principle (DIP)

- Depend on abstractions, not on concretions.

# SOLID: Single Responsibility Principle

## Single Responsibility Principle (SRP)

- In SRP a reason to change is defined as a responsibility, therefore SRP states

*“An object should have only one reason to change.”*

- If an object has more than one reason to change then it has more than one responsibility and is in violation of SRP. An object should have one and only one reason to change.
- Consider the following example where I have a `BankAccount` class that has some methods:

## Single Responsibility Principle (SRP): Example (contd.)

```
//BankAccount
public class BankAccount {

    private double balance;

    public BankAccount () {...};

    public void setBalance (double newBalance) {
        balance = newBalance;
    }
    public double getBalance () {
        return balance;
    }

    public void deposit(double amount){...}
    public void withdraw(double amount){...}
    public void addInterest(double amount){...}
    public void transferMoney(double fromAcc, double toAcc, double
        amount){...}
}
```

## Single Responsibility Principle (SRP): Example (contd.)

- Say we use this `BankAccount` class for a person's `Checking` and `Savings` account. That would cause this class to have more than two reasons to change.
- This is because `Checking` accounts do not have interest added to them and only `Savings` accounts have interest added to them on a monthly basis or however the bank calculates it.
- So, let's refactor this to be more SRP friendly.

## Single Responsibility Principle (SRP): Example (contd.)

```
//abstract BankAccount
public abstract class BankAccount {

    private double balance;

    public BankAccount () {...};

    public void setBalance (double newBalance) {
        balance = newBalance;
    }
    public double getBalance () {
        return balance;
    }

    public void deposit(double amount){...}
    public void withdraw(double amount){...}
    public void transferMoney(double fromAcc, double toAcc, double
        amount){...}
}
```

## Single Responsibility Principle (SRP): Example (contd.)

```
//ChequingAccount
public class ChequingAccount extends BankAccount {

    public ChequingAccount () {...};

    // Some other methods
}

//SavingsAccount
public class SavingsAccount extends BankAccount {

    public SavingsAccount () {...};

    public void addInterest(double amount)
    {
        //calculate interest
    }
}
```

## Single Responsibility Principle (SRP): Example (contd.)

- So what we have done is simply create an abstract class out of `BankAccount` and then created a concrete `ChequingAccount` and `SavingsAccount` class so that we can isolate the methods that are causing more than one reason to change.
- SRP is one of the hardest principles to enforce because there is always room for refactoring out one class to multiple; each class has one responsibility.

# **SOLID: Open-Closed Principle**

## Open-Closed Principle (OCP)

- The open/closed principle states  
*“software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**”;*  
that is, such an entity can allow its behavior to be modified without altering its source code.
- In practice, this means creating software entities whose behavior can be changed without the need to edit and recompile the code itself.

## Open-Closed Principle (OCP)

- The simplest way to demonstrate this principle is to consider a method that does one thing.
- The Open-Closed Principle can also be achieved in many other ways, including through the use of inheritance or through compositional design patterns like the Strategy pattern.

## Open-Closed Principle (OCP): Example

- Let's say that we've got a Rectangle class.

```
public class Rectangle {  
    private double width;  
    private double height;  
  
    public double getWidth() {  
        return width;  
    }  
    public void setWidth(double width) {  
        this.width = width;  
    }  
  
    public double getHeight() {  
        return height;  
    }  
    public void setHeight(double height) {  
        this.height = height;  
    }  
}
```

## Open-Closed Principle (OCP): Example (contd.)

- Now we want to build an application that can calculate the total area of a collection of rectangles.

```
import java.util.ArrayList;

public class AreaCalculator {

    public double Area(ArrayList<Rectangle> shapes)
    {
        double area = 0;

        for(Rectangle rect : shapes){
            area += rect.getWidth()*rect.getHeight();
        }
        return area;
    }
}
```

## Open-Closed Principle (OCP): Example (contd.)

- Congratulations! You have done a great job! Now we want to extend this application so that it can calculate the area of not only rectangles but also circles as well.

## Open-Closed Principle (OCP): Example (contd.)

- And we come up with a solution as given below:

```
import java.util.ArrayList;

public class AreaCalculator {
    public double Area(ArrayList<Object> shapes){
        double area = 0;
        for(Object shape : shapes){
            if (shape instanceof Rectangle){
                Rectangle rectangle = (Rectangle) shape;
                area += rectangle.getWidth()*rectangle.getHeight();
            }
            else{
                Circle circle = (Circle) shape;
                area += circle.getRadius()*Math.PI;
            }
        }
        return area;
    }
}
```

## Open-Closed Principle (OCP): Example (contd.)

- The solution presented in the previous slide works well. But the solution have modified the `AreaCalculator` method. Now we want to calculate the area of triangles.
- To incorporate the area of triangles, again we need to make changes in the `AreaCalculator` method.
- It seems easy to extend the program to calculate the area of triangles. But in a real world scenario where the code base is ten, a hundred or a thousand times larger and modifying the class means redeploying it's assembly/package to five different servers that can be a pretty big problem.
- Oh, and in the real world your software client would have changed the requirements five more times since you read the last sentence.

## Open-Closed Principle (OCP): Example (contd.)

- A solution that abides by the Open/Closed Principle. Let's define an abstract method for calculating the area of a shape.

```
public abstract class Shape {  
    public abstract double Area();  
}
```

## Open-Closed Principle (OCP): Example (contd.)

- Rectangle inherits from Shape

```
public class Rectangle extends Shape{  
    private double width;  
    private double height;  
    public double getWidth() {  
        return width;  
    }  
    public void setWidth(double width) {  
        this.width = width;  
    }  
    public double getHeight() {  
        return height;  
    }  
    public void setHeight(double height) {  
        this.height = height;  
    }  
    public double Area() {  
        return width*height;  
    }  
}
```

## Open-Closed Principle (OCP): Example (contd.)

- Circle inherits from Shape

```
public class Circle extends Shape{
    private double radius;

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    public double Area() {
        return radius*radius*Math.PI;
    }
}
```

## Open-Closed Principle (OCP): Example (contd.)

```
import java.util.ArrayList;

public class Area {

    public double Area(ArrayList<Shape> shapes)
    {
        double area = 0;

        for(Shape s : shapes){
            area += s.Area();
        }
        return area;
    }
}
```

## Open-Closed Principle (OCP): Example (contd.)

- As we've moved the responsibility of actually calculating the area away from `AreaCalculator`'s `Area` method it is now much simpler and robust as it can handle any type of `Shape` that we throw at it.
- In other words we've closed it for modification by opening it up for extension.

# **SOLID: Liskov Substitution Principle**

## Liskov Substitution Principle (LSP)

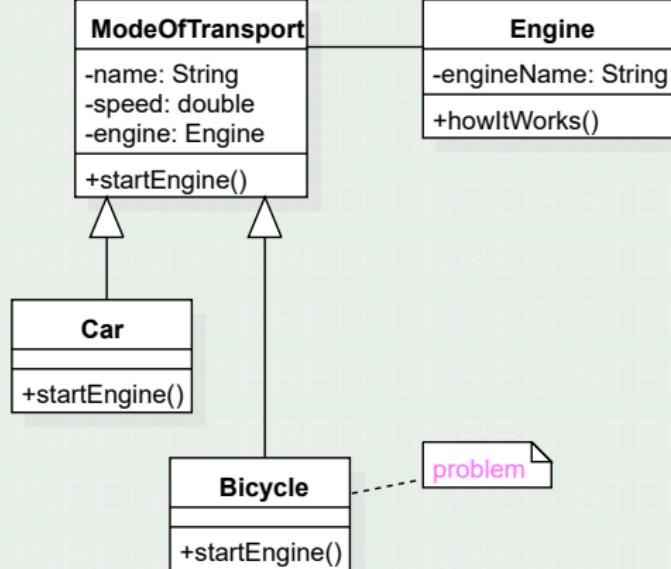
- The Liskov Substitution Principle (LSP) states that  
*“derived classes must be substitutable for the base class”.*
- When this principle is violated, it tends to result in a lot of extra conditional logic scattered throughout the application, checking to see the specific type of an object.
- This duplicate, scattered code becomes a breeding ground for bugs as the application grows.

## Liskov Substitution Principle (LSP)

- Most introductions to object-oriented development discuss inheritance, and explain that one object can inherit from another if it has an “IS-A” relationship with the inherited object.
- However, this is necessary, but not sufficient. It is more appropriate to say that one object can be designed to inherit from another if it always has an “IS-SUBSTITUTABLE-FOR” relationship with the inherited object.

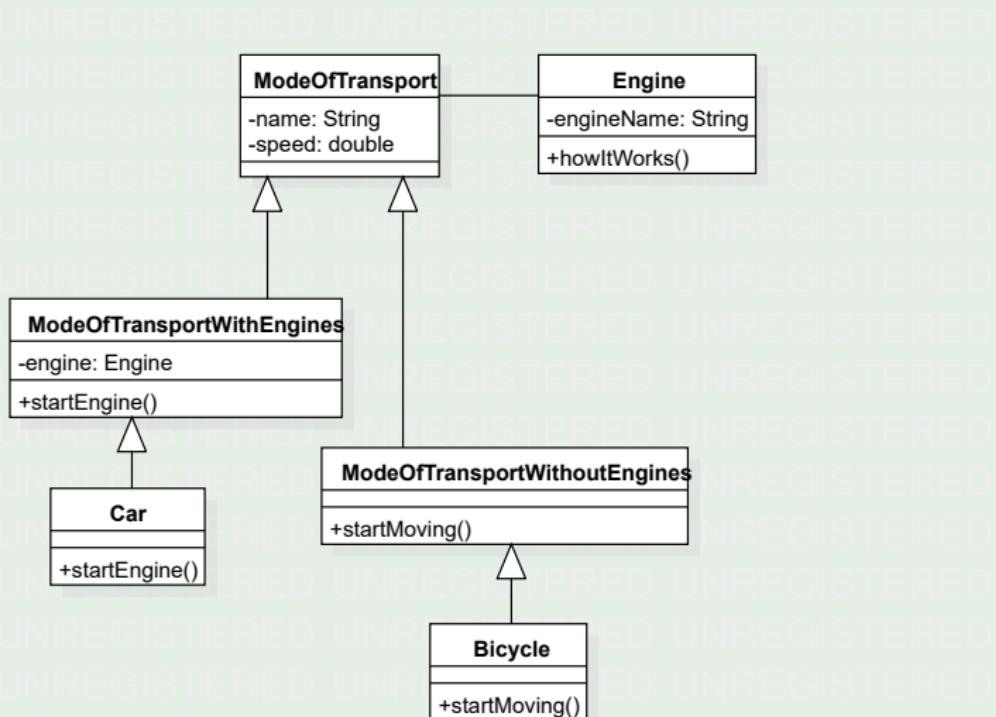
## Liskov Substitution Principle (LSP): Example

- Liskov Substitution Principle Violation



## Liskov Substitution Principle (LSP): Example (contd.)

- Following the Liskov Substitution Principle



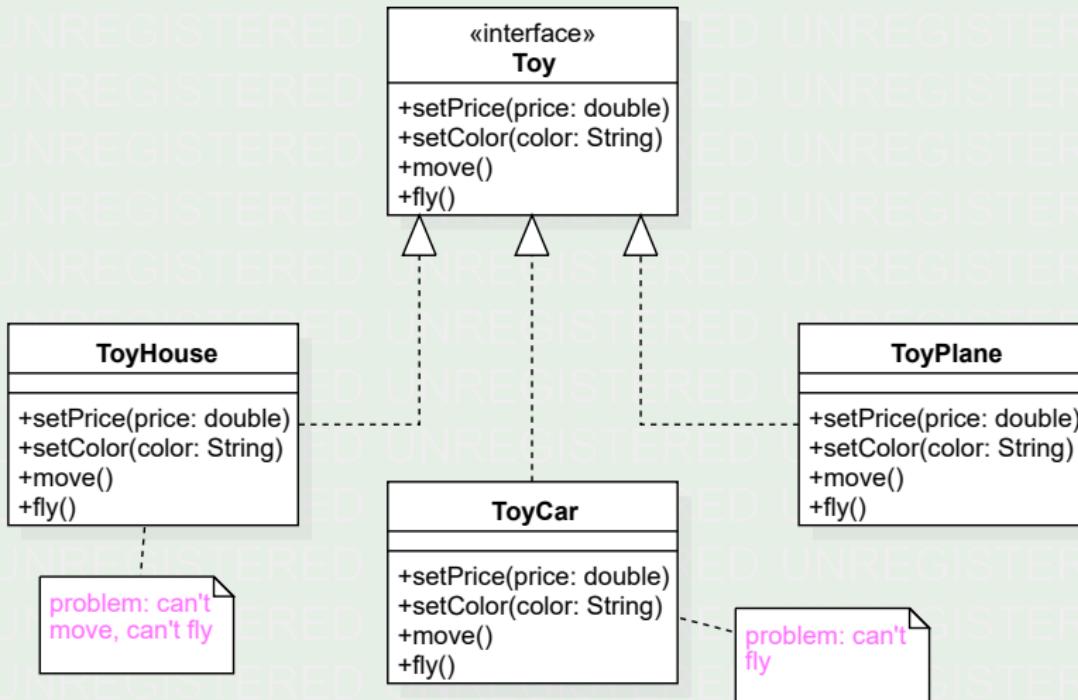
# **SOLID**: Interface Segregation Principle

## Interface Segregation Principle (ISP)

- The Interface Segregation Principle (ISP) states that  
*“clients should not be forced to depend on methods that they do not use”.*
- Application developers should favor thin, focused interfaces to “fat” interfaces that offer more functionality than a particular class or method needs.
- Ideally, your thin interfaces should be cohesive, meaning they have groups of operations that logically belong together.
- Another benefit of smaller interfaces is that they are easier to implement fully, and thus less likely to break the Liskov Substitution Principle by being only partially implemented.

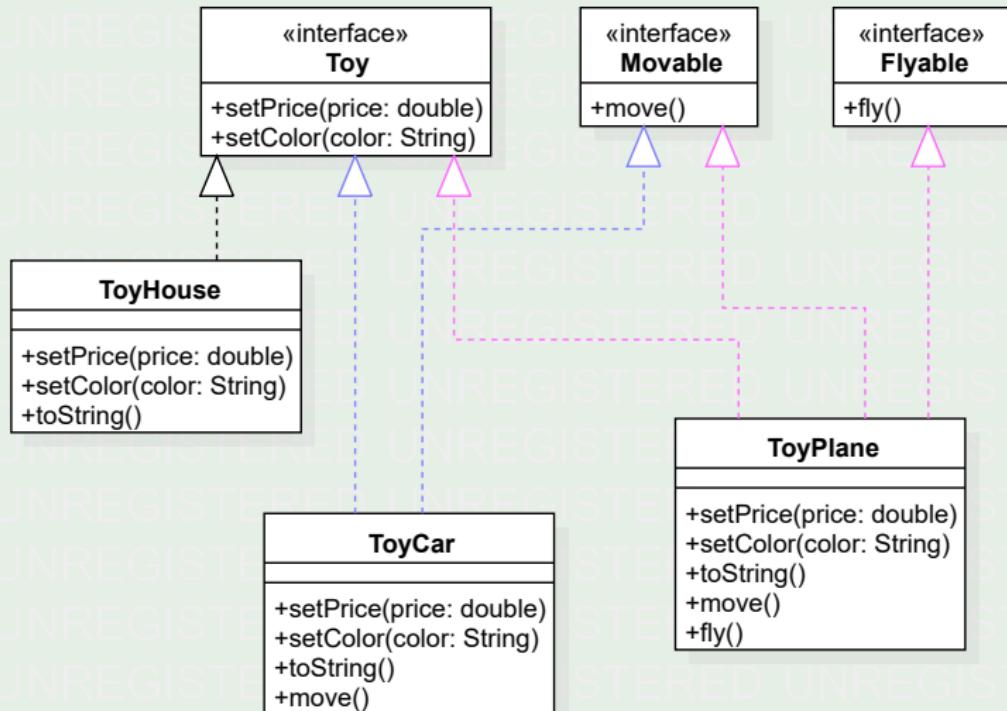
## Interface Segregation Principle (ISP): Example

- Interface Segregation Principle Violation



## Interface Segregation Principle (ISP): Example (contd.)

- Following the Interface Segregation Principle



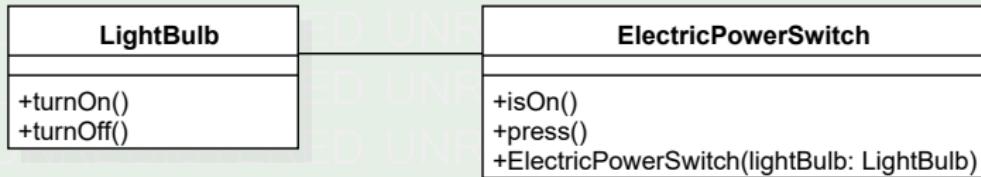
# **SOLID**: Dependency Inversion Principle

## Dependency Inversion Principle (DIP)

- The Dependency Inversion Principle (DIP) states that  
*“high level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.”*
- It's extremely common when writing software to implement it such that each module or method specifically refers to its collaborators, which does the same.
- This type of programming typically lacks sufficient layers of abstraction, and results in a very tightly coupled system, since every module is directly referencing lower level modules.

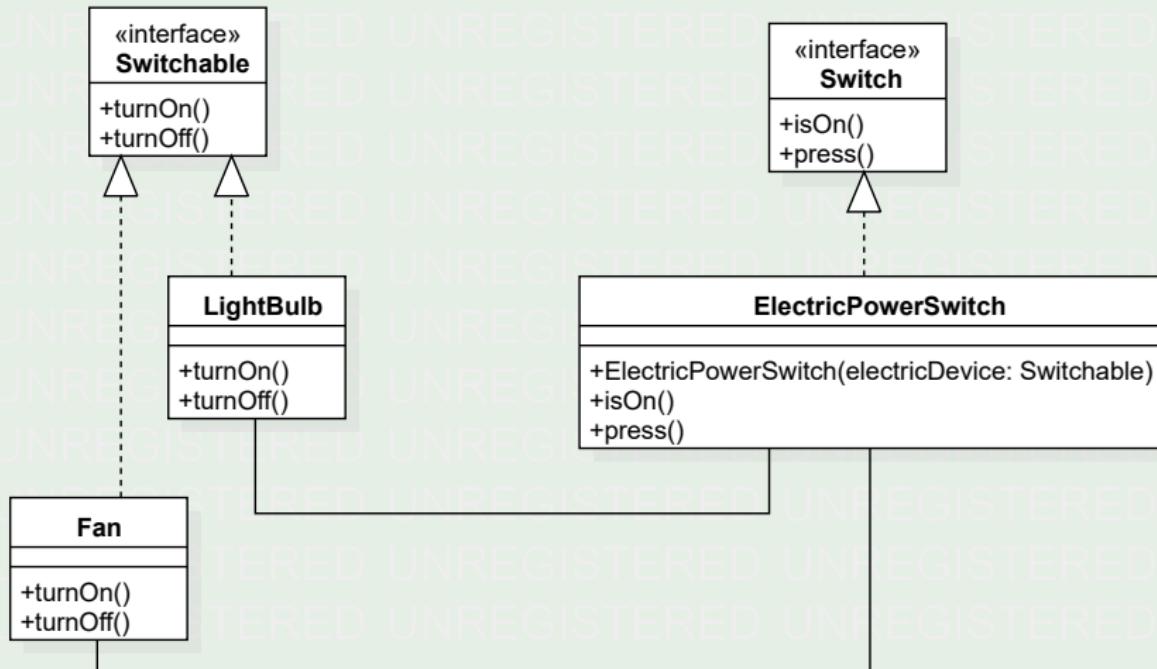
## Dependency Inversion Principle (DIP): Example

- Dependency Inversion Principle Violation



## Dependency Inversion Principle (DIP): Example (contd.)

- Following the Dependency Inversion Principle



# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

# Sample UP artifact influence

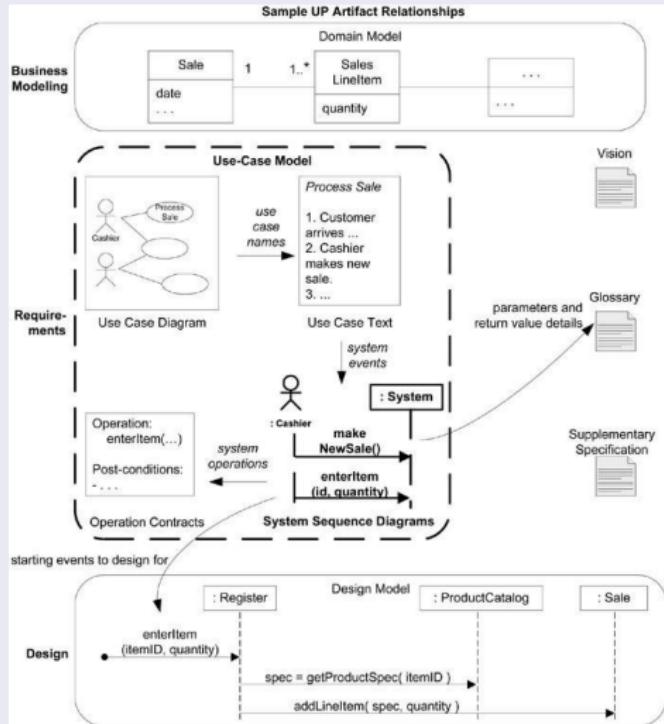


Figure: Sample UP artifact influence.

## System Sequence Diagrams (SSD)

- A **system sequence diagram (SSD)** is a fast and easily created artifact that illustrates input and output events related to the systems under discussion.
- They are **input** to operation contracts and most importantly—object design.
- The use case text and its implied system events are **input** to SSD creation.
- The SSD operations (such as `enterItem`) can in turn be analyzed in the operation contracts, detailed in the Glossary, and most important serve as the starting point for designing collaborating objects.

## Example: NextGen SSD

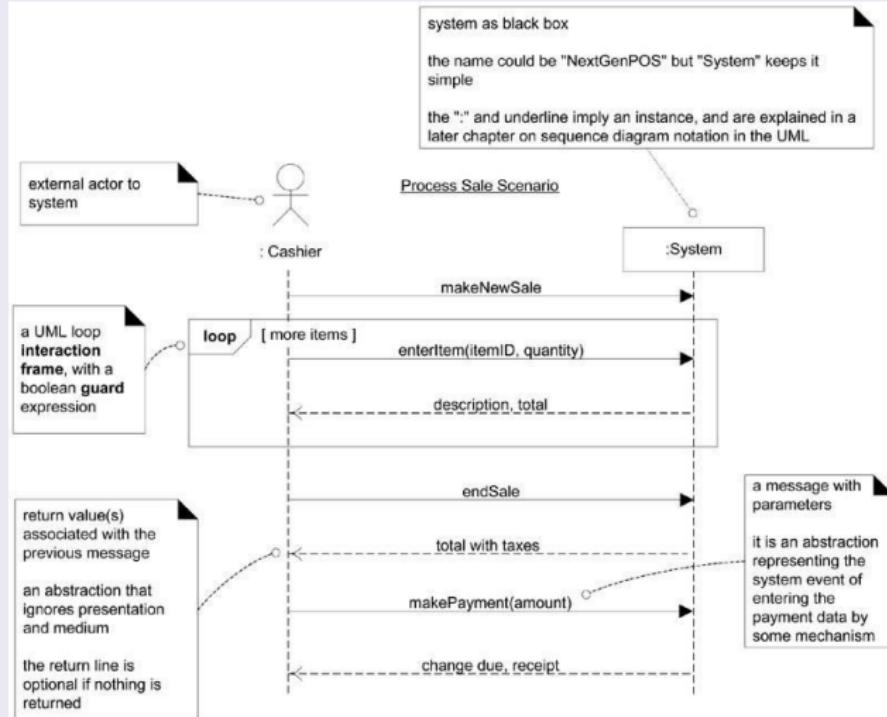


Figure: SSD for a Process Sale scenario.

## What are System Sequence Diagrams?

- Use cases describe how external actors interact with the software system we are interested in creating.
- During this interaction an actor generates **system events** to a system, usually requesting some **system operation** to handle the event.

For example,

when a cashier enters an item's ID, the cashier is requesting the POS system to record that item's sale (the `enterItem` event). That event initiates an operation upon the system.

## What are System Sequence Diagrams? (contd.)

- The UML includes **sequence diagrams** as a notation that can illustrate actor interactions and the operations initiated by them.
- A **system sequence diagram** is a picture that shows, for one particular scenario of a use case, the events that external actors generate, their order, and inter-system events.
- All systems are treated as a black box; the emphasis of the diagram is events that cross the system boundary from actors to systems.
- **Guideline:** Draw an SSD for a main success scenario of each use case, and frequent or complex alternative scenarios.

## Why Draw an SSD?

- What events are coming in to our system? Why?
- Because we have to design the software to handle these events (from the mouse, keyboard, another system, ...) and execute a response.
- Basically, a software system reacts to three things:
  - ① external events from actors (humans or computers),
  - ② timer events, and
  - ③ faults or exceptions (which are often from external sources).
- It is useful to know what, precisely, are the external (input) events—the system events. They are an important part of analyzing system behavior.

## Why Draw an SSD? (contd.)

- **System behavior** is a description of what a system does, without explaining how it does it.
- One part of that description is a system sequence diagram.
- Other parts include the use cases and system operation contracts.

## Applying UML: Sequence Diagrams

- The UML does not define something called a “system” sequence diagram but simply a “sequence diagram.”
- The qualification is used to emphasize its application to systems as black boxes.
- *Later, sequence diagrams will be used in another context to illustrate the design of interacting software objects to fulfill work.*

# What is the Relationship Between SSDs and Use Cases?

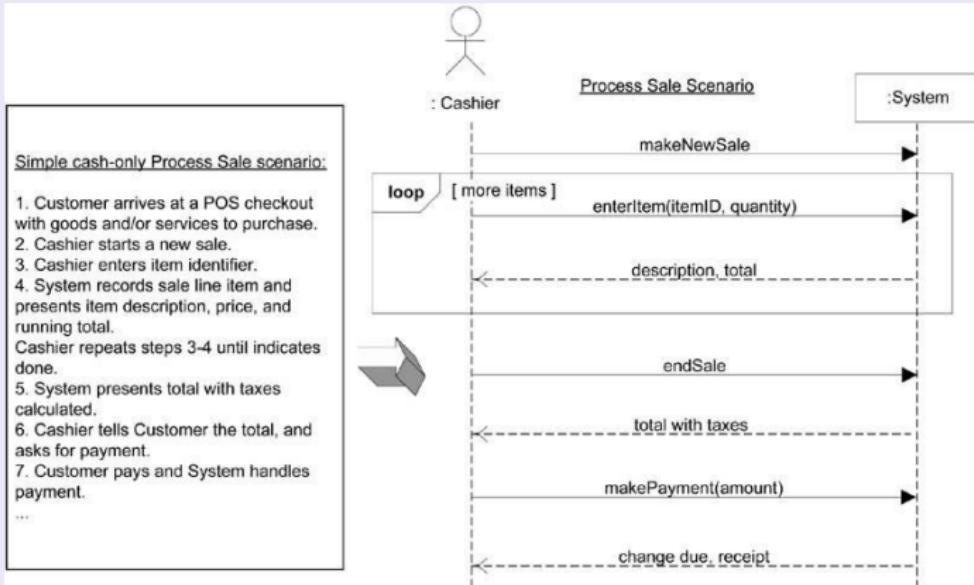
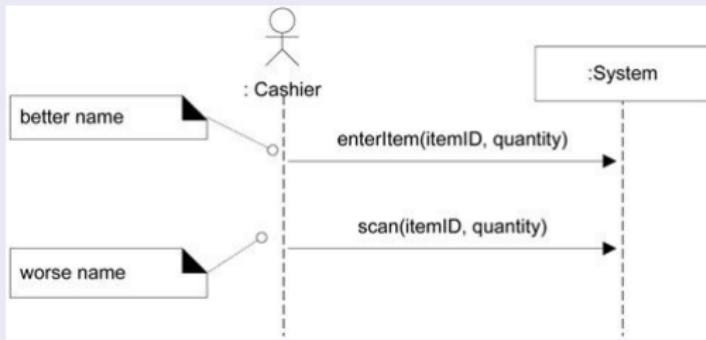


Figure: SSDs are derived from use cases; they show one scenario.

## Applying UML: Should We Show Use Case Text in the SSD?

- Not usually. If you name the SSD appropriately, you can indicate the use case; for example, “Process Sale Scenario”.

## How to Name System Events and Operations?



## How to Name System Events and Operations? (contd.)

- Which is better, `scan(itemID)` or `enterItem(itemID)`?
- System events should be expressed at the abstract level of intention rather than in terms of the physical input device.
- Thus `enterItem` is better than `scan` because it captures the intent of the operation while remaining abstract and noncommittal with respect to design choices about what interface is used to capture the system event. It could be via laser scanner, keyboard, voice input, or anything.
- It also improves clarity to start the name of a system event with a **verb** (`add...`, `enter...`, `end...`, `make...`), since it emphasizes these are commands or requests.

## How to Model SSDs Involving Other External Systems?

- SSDs can also be used to illustrate collaborations between systems, such as between the NextGen POS and the external credit payment authorizer.

## What SSD Information to Place in the Glossary?

- The elements shown in SSDs (**operation name, parameters, return data**) are terse.
- These may need proper explanation so that during design it is clear what is coming in and going out. The **Glossary** is a great place for these details.

For example,

in SSD, there is a return line containing the description  
“change due, receipt.”

- That's a vague description about the receipt—a complex report. So, the UP Glossary can have a receipt entry, that shows sample receipts (perhaps a digital picture), and detailed contents and layout.
- **Guideline:** In general for many artifacts, show details in the Glossary.

## Iterative and Evolutionary SSDs

- **Don't create SSDs for all scenarios.** Rather, draw them only for the scenarios chosen for the next iteration.
- And, they shouldn't take long to sketch - perhaps a few minutes or a half hour.
- SSDs are also very useful when you want to understand the interface and collaborations of existing systems, or to document the architecture.

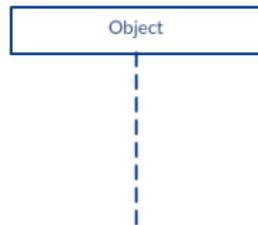
## SSDs Within the UP

- SSDs are part of the **Use-Case Model**—a visualization of the interactions implied in the scenarios of use cases.
- SSDs are an example of the many possible skillful and widely used analysis and design artifacts or activities that the UP documents do not mention.
- But the UP, being very flexible, encourages the inclusion of any and all artifacts and practices that add value.

## UP Phases

- Inception
  - SSDs are not usually motivated in inception, unless you are doing rough estimating (don't expect inception estimating to be reliable) involving a technique that is based on identifying system operations.
- Elaboration
  - Most SSDs are created during elaboration, when it is useful to identify the details of the system events to clarify what major operations the system must be designed to handle, write system operation contracts, and possibly to support estimation.

## Applying UML: Sequence Diagrams

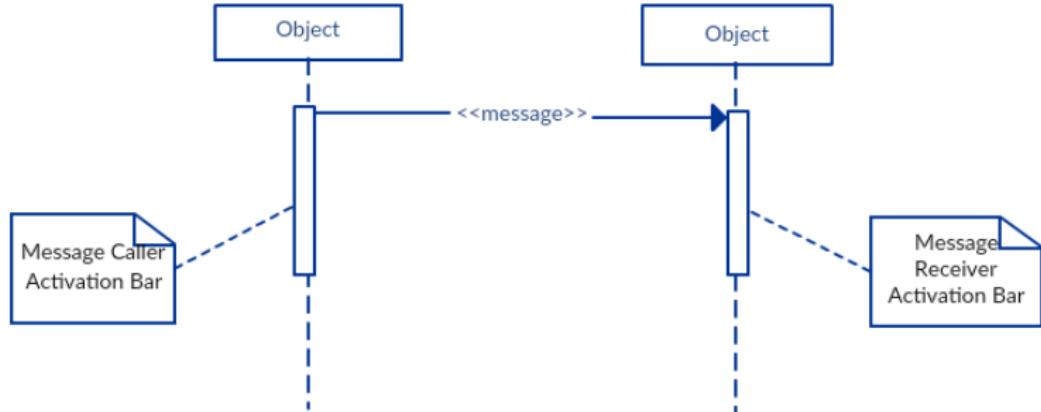


(a)



(b)

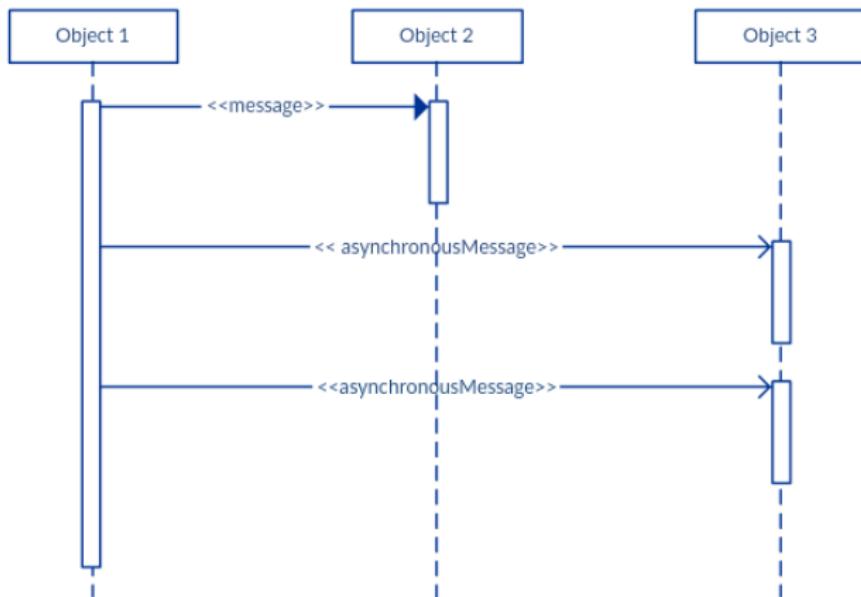
## Sequence Diagram: Activation Bars



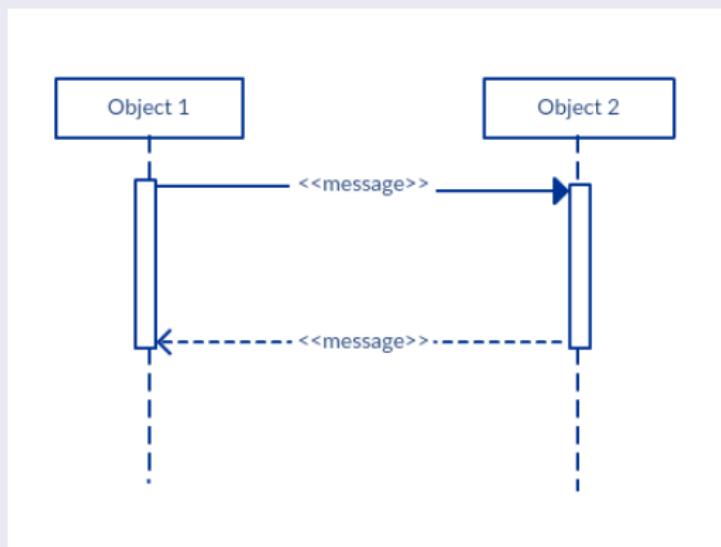
## Sequence Diagram: Synchronous Message



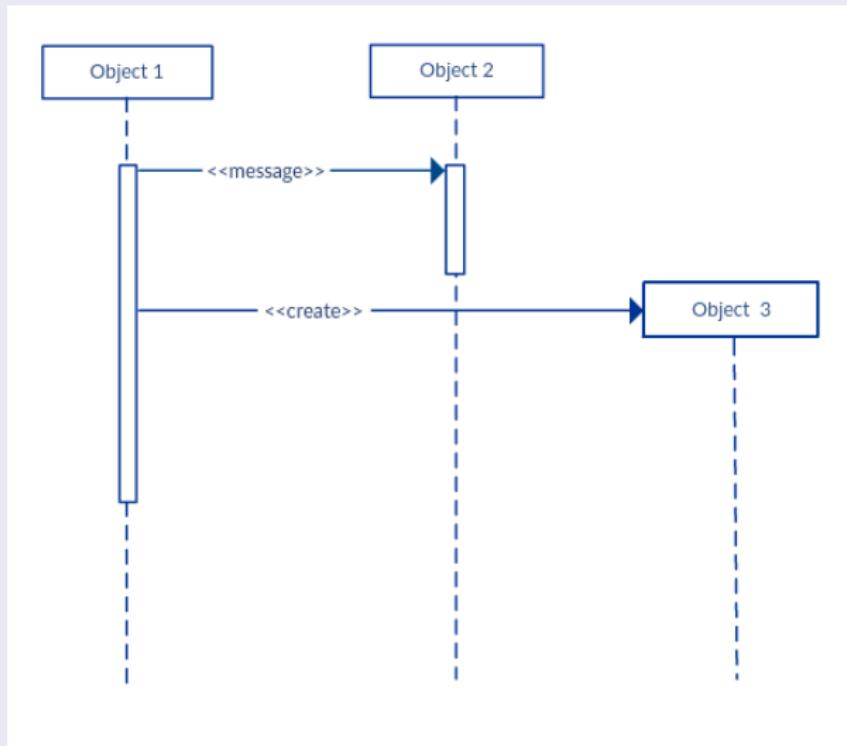
## Sequence Diagram: Asynchronous Message



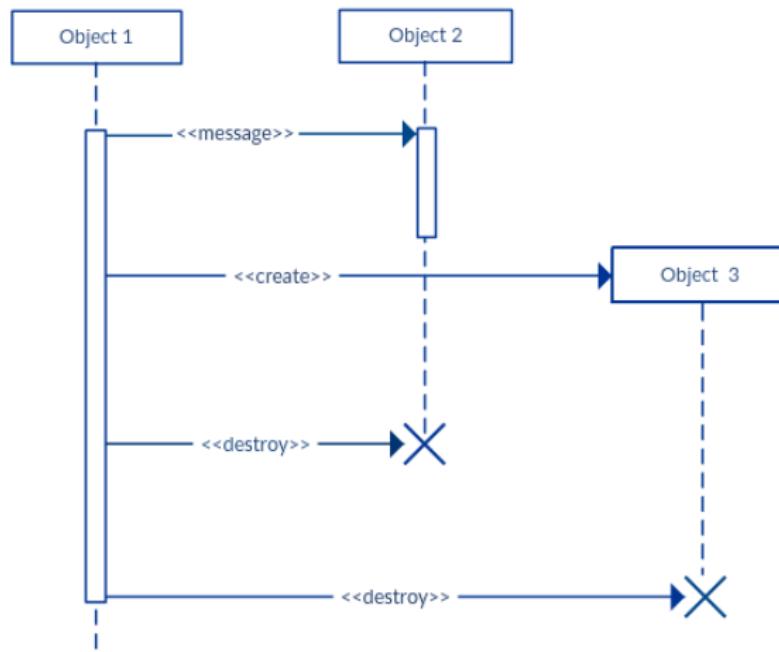
## Sequence Diagram: Return Message



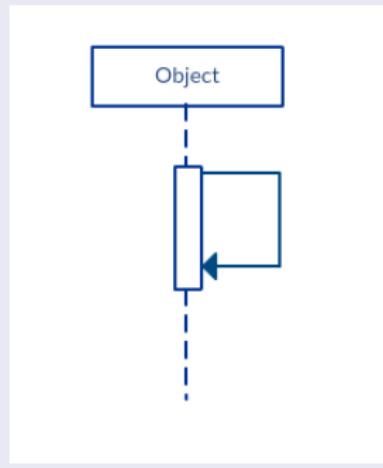
## Sequence Diagram: Participant Creation Message



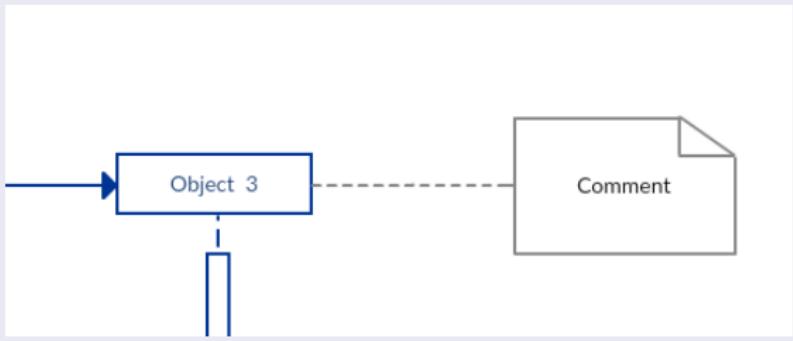
## Sequence Diagram: Participant Destruction Message



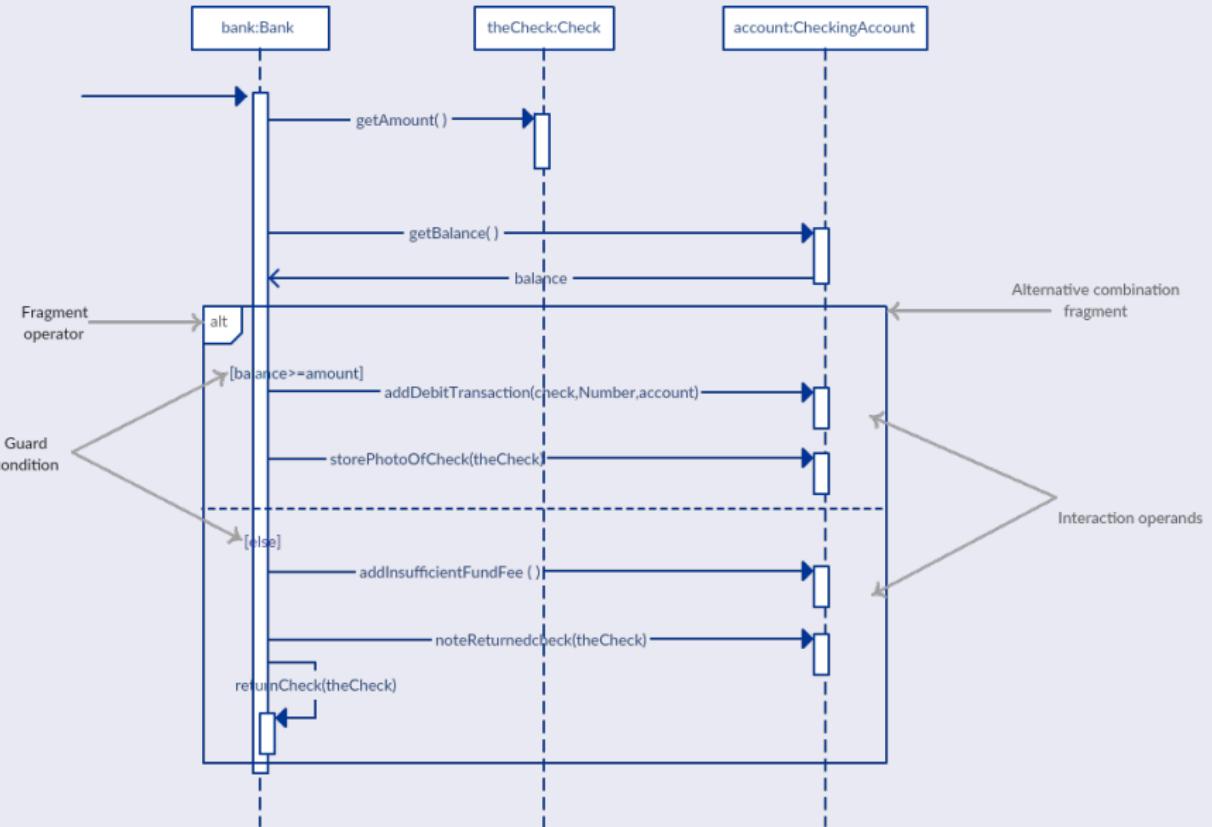
## Sequence Diagram: Reflexive Message



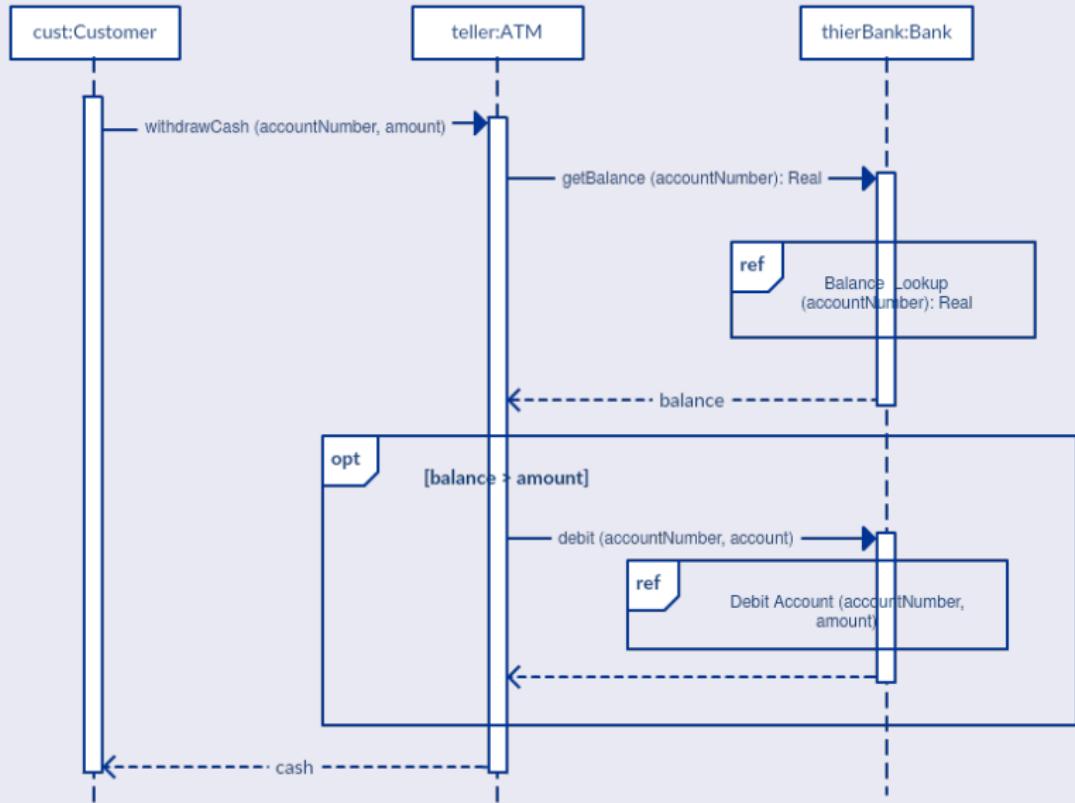
## Sequence Diagram: Comment



# Sequence Diagram: Alternatives



## Sequence Diagram: Options



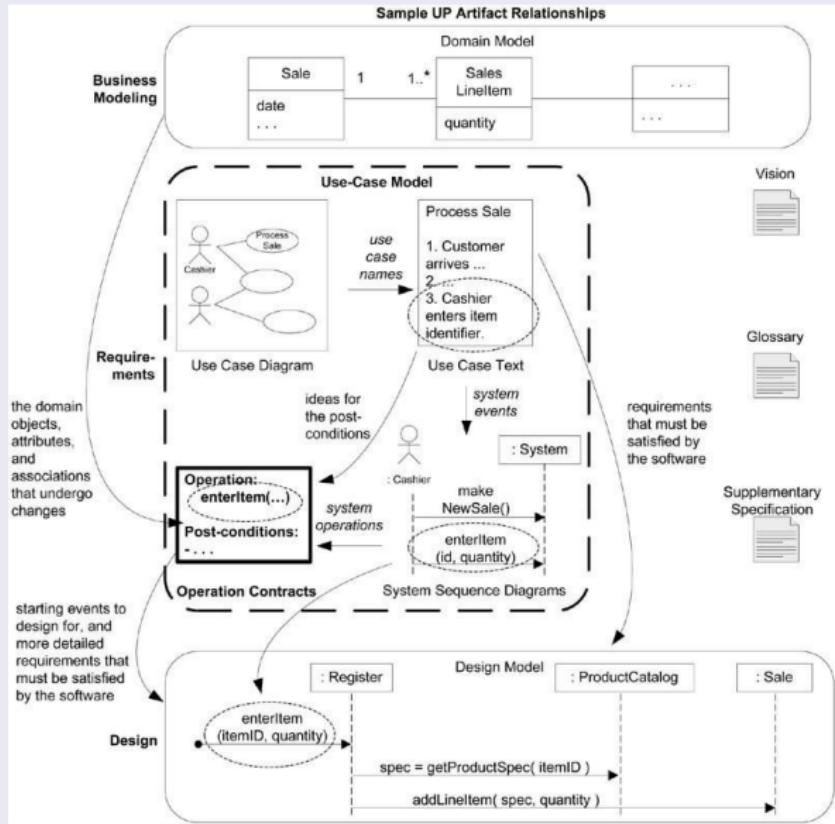
## It's Quiz Time

- ① Draw an SSD for a main success scenario of each use case, and frequent or complex alternative scenarios. (True or False)
- ② SSDs are not part of the Use-Case Model. (True or False)
- ③ Use cases describe how external actors interact with the software system we are interested in creating. (True or False)

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

# Operation Contracts (contd.)



## Operation Contracts

- Use cases or system features are the main ways in the UP to describe system behavior, and are usually sufficient.
- Sometimes a more detailed or precise description of system behavior has value.
- **Operation contracts** use a **pre- and post-condition form** to describe detailed changes to objects in a domain model, as the result of a system operation.
- A domain model is the most common OOA model, but **operation contracts** and **state models** can also be useful OOA-related artifacts.

## Operation Contracts (contd.)

- Operation contracts may be considered part of the **UP Use-Case Model** because they provide more analysis detail on the effect of the system operations implied in the use cases.
- The prime inputs to the contracts are:
  - ① the system operations identified in SSDs (such as `enterItem`),
  - ② the domain model, and
  - ③ domain insight from experts.
- *The contracts can in turn serve as input to the object design, as they describe changes that are likely required in the software objects or database.*

## An Example: Operation Contracts

### Contract CO2: enterItem

**Operation:** enterItem(itemID: ItemID, quantity: integer)

**Cross References:** Use Cases: Process Sale

**Preconditions:** There is a sale underway.

**Postconditions:** - A SalesLineItem instance sli was created (*instance creation*).

- sli was associated with the current Sale (*association formed*).

- sli.quantity became quantity (*attribute modification*).

- sli was associated with a ProductDescription, based on itemID match (*association formed*).

The categorizations such as "*(instance creation)*" are a learning aid, not properly part of the contract.

## Definition: What are the Sections of a Contract?

- 
- |                          |   |
|--------------------------|---|
| <b>Operation:</b>        | Name of operation, and parameters   |
| <b>Cross References:</b> | Use cases this operation can occur within   |
| <b>Preconditions:</b>    | Noteworthy assumptions about the state of the system or objects in the Domain Model before execution of the operation. These are non-trivial assumptions the reader should be told. |
| <b>Postconditions:</b>   | This is the most important section. The state of objects in the Domain Model after completion of the operation. Discussed in detail in a following section.                         |

## Definition: What is a System Operation?

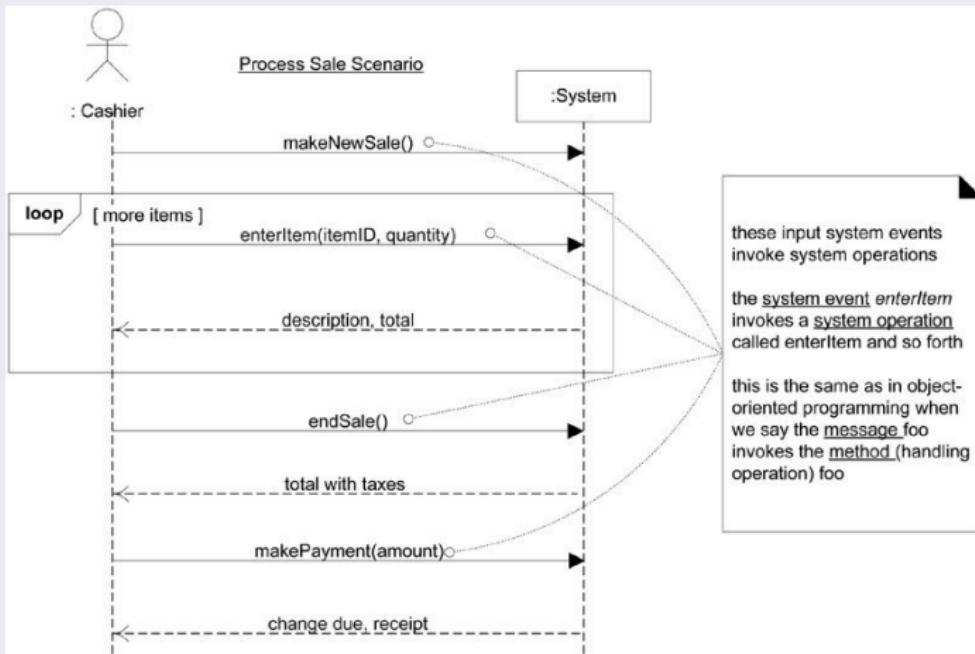


Figure: SSD. *System operations handle input system events.*

## Definition: What is a System Operation? (contd.)

- Operation contracts may be defined for system operations—operations that the system as a black box component offers in its public interface.
- *System operations can be identified while sketching SSDs.*
- To be more precise, the SSDs show system events or I/O messages relative to the system.
- Input system events imply the system has system operations to handle the events, just as an OO message (a kind of event or signal) is handled by an OO method (a kind of operation). (see the figure on the next slide.)
- The entire set of system operations, across all use cases, defines the public **system interface**, viewing the system as a single component or class.
- In the UML, the system as a whole can be represented as one object of a class named (for example) System.

## Definition: Postconditions

- The postconditions describe **changes in the state of objects** in the domain model. Domain model state changes include
  - instances created/deleted
  - associations formed or broken, and
  - attributes changed.
- **Postconditions are not actions to be performed during the operation;** rather, they are observations about the domain model objects that are true when the operation has finished after the smoke has cleared.
- The postconditions fall into these categories:
  - Instance creation and deletion.
  - Attribute change of value.
  - Associations formed and broken.

## How are Postconditions Related to the Domain Model?

- These postconditions are expressed in the context of the Domain Model objects.
- What instances can be created?
  - those from the Domain Model.
- What associations can be formed?
  - those in the Domain Model

## Why Postconditions?

- First, they aren't always necessary. But sometimes more detail and precision is useful. Contracts offer that.
- The postconditions **support fine-grained detail and precision in declaring what the outcome of the operation must be.**
  - It is also possible to express this level of detail in the use cases, but undesirable—they would be too verbose and low-level detailed.
- A contract is an excellent **tool of requirements analysis or OOA that describes in great detail the changes required by a system operation** (in terms of the domain model objects) without having to describe how they are to be achieved.
- In other words, the design can be deferred, and we can focus on the analysis of *what* must happen, rather than *how* it is to be accomplished.

## Guideline: How to Write a Postcondition?

- Express postconditions in the **past tense** to emphasize they are observations about state changes that arose from an operation, not an action to happen. That's why they are called postconditions!

For example:

- (better) A SalesLineItem was created.  
rather than
- (worse) Create a SalesLineItem, or, A SalesLineItem is created.

## Guideline: How Complete Should Postconditions Be? Agile vs. Heavy Analysis

- Generating a complete and detailed set of postconditions for all system operations is not likely or necessary.
- In the spirit of Agile Modeling, treat their creation as an initial best guess, with the understanding they will not be complete and that “perfect” complete specifications are rarely possible or believable.
- But understanding that light analysis is realistic and skillful doesn’t mean to abandon a little investigation before programming that’s the other extreme of misunderstanding.

## Example: `enterItem` Postconditions

This example dissects the motivation for the postconditions of the `enterItem` system operation.

- Instance Creation and Deletion

For example,

After the `itemID` and quantity of an item have been entered, what new object should have been created?

- A `SalesLineItem`. Thus:
- A `SalesLineItem` instance `sli` was created (instance creation).
- Note the naming of the instance. This name will simplify references to the new instance in other post-condition statements.

## Example: `enterItem` Postconditions (contd.)

- Attribute Modification

For example,

After the `itemID` and `quantity` of an item have been entered by the cashier, what attributes of new or existing objects should have been modified?

- The quantity of the `SalesLineItem` should have become equal to the `quantity` parameter. Thus:
- `sli.quantity` became `quantity` (attribute modification).

## Example: `enterItem` Postconditions (contd.)

- Associations Formed and Broken

For example,

- After the `itemID` and quantity of an item have been entered by the cashier, what associations between new or existing objects should have been formed or broken?
- The new `SalesLineItem` should have been related to its `Sale`, and related to its `ProductDescription`. Thus:
  - `sli` was associated with the current `Sale` (association formed).
  - `sli` was associated with a `ProductDescription`, based on `itemID` match (association formed).

### Guideline: Should We Update the Domain Model?

- In iterative and evolutionary methods (and reflecting the reality of software projects), all analysis and design artifacts are considered partial and imperfect, and evolve in response to new discoveries.

## Guideline: When Are Contracts Useful?

- In the UP, **the use cases are the main repository of requirements for the project.** They may provide most or all of the detail necessary to know what to do in the design, in which case, contracts are not helpful. However, there are situations where **the details and complexity of required state changes are awkward or too detailed to capture in use cases.**

For example:

consider an airline reservation system and the system operation addNewReservation. The complexity is very high regarding all the domain objects that must be changed, created, and associated. These fine-grained details can be written up in the use case, but it will make it extremely detailed (for example, noting each attribute in all the objects that must change).

### Guideline: When Are Contracts Useful? (contd.)

- Observe that the postcondition format offers and encourages a very precise, analytical language that supports detailed thoroughness.
- If developers can comfortably understand what to do without them, then avoid writing contracts.

## Guideline: How to Create and Write Contracts

- ① Identify system operations from the SSDs.
- ② For system operations that are complex and perhaps subtle in their results, or which are not clear in the use case, construct a contract.
- ③ To describe the postconditions, use the following categories:
  - instance creation and deletion
  - attribute modification
  - associations formed and broken

## Guideline: How to Create and Write Contracts (contd.)

### Writing Contracts

- As mentioned, write the postconditions in a declarative, passive past tense form (was ...) to emphasize the observation of a change rather than a design of how it is going to be achieved.

For example,

- (better) A SalesLineItem was created.
- (worse) Create a SalesLineItem.

- Remember to establish an association between existing objects or those newly created.

For example,

it is not enough that a new SalesLineItem instance is created when the `enterItem` operation occurs. After the operation is complete, it should also be true that the newly created instance was associated with Sale; thus:

The SalesLineItem was associated with the Sale (association formed).

## Guideline: How to Create and Write Contracts (contd.)

### What's the Most Common Mistake?

- The most common problem is forgetting to include the **forming of associations**.
  - Particularly when new instances are created, it is very likely that associations to several objects need be established. Don't forget!

## Example-1: NextGen POS Contracts-System Operations of the *Process Sale* Use Case

### Contract C01: makeNewSale

**Operation:** makeNewSale()

**Cross References:** Use Cases: Process Sale

**Preconditions:** none

**Postconditions:** - A Sale instance s was created (instance creation).

- s was associated with a Register (association formed).

- Attributes of s were initialized.

## Example-2: NextGen POS Contracts-System Operations of the *Process Sale* Use Case (contd.)

### Contract CO2: enterItem

<b>Operation:</b>	enterItem(itemID: ItemID, quantity: integer)
<b>Cross References:</b>	Use Cases: Process Sale
<b>Preconditions:</b>	There is a sale underway.
<b>Postconditions:</b>	<ul style="list-style-type: none"><li>- A SalesLineItem instance sli was created (instance creation).</li><li>- sli was associated with the current Sale (association formed).</li><li>- sli.quantity became quantity (attribute modification).</li><li>- sli was associated with a ProductDescription, based on itemID match (association formed).</li></ul>

## Example-3 & 4: NextGen POS Contracts-System Operations of the Process Sale Use Case (contd.)

### Contract CO3: endSale

<b>Operation:</b>	endSale()
<b>Cross References:</b>	Use Cases: Process Sale
<b>Preconditions:</b>	There is a sale underway.
<b>Postconditions:</b>	<ul style="list-style-type: none"><li>- Sale.isComplete became true (attribute modification).</li></ul>

### Contract CO4: makePayment

<b>Operation:</b>	makePayment( amount: Money )
<b>Cross References:</b>	Use Cases: Process Sale
<b>Preconditions:</b>	There is a sale underway.
<b>Postconditions:</b>	<ul style="list-style-type: none"><li>- A Payment instance p was created (instance creation).</li><li>- p.amountTendered became amount (attribute modification).</li><li>- p was associated with the current Sale (association formed).</li><li>- The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales)</li></ul>

## Changes to the POS Domain Model

- There is at least one point suggested by these contracts that is not yet represented in the domain model: completion of item entry to the sale.
- The `endSale` specification modifies it, and it is probably a good idea later during design work for the `makePayment` operation to test it, to disallow payments until a sale is complete (meaning, no more items to add).
- One way to represent this information is with an `isComplete` attribute in the `Sale`:



## Operation Contracts Within the UP

- Inception
  - Contracts are not motivated during inception—they are too detailed.
- Elaboration
  - If used at all, most contracts will be written during elaboration, when most use cases are written.
  - *Only write contracts for the most complex and subtle system operations.*

## It's Quiz Time

- ① Operation contracts may be considered part of the UP Business Modeling (True or False)
- ② The postconditions support fine-grained detail and precision in declaring what the outcome of the operation must be. (True or False)
- ③ “Identify system operations from the SSDs” is one of the advices to create contracts. (True or False)

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

# Requirements To Design - Iteratively

## Iteratively Do the Right Thing, Do the Thing Right

- The requirements and object-oriented analysis covered so far has focused on learning to **do the right thing**; that is, understanding some of the outstanding goals for the case studies, and related rules and constraints.
- By contrast, the following design work will stress **do the thing right**; that is, skillfully designing a solution to satisfy the requirements for this iteration

## Iteratively Do the Right Thing, Do the Thing Right (contd.)

- In iterative development, a transition from primarily a requirements or analysis focus to primarily a design and implementation focus will occur in each iteration.
- Early iterations will spend relatively more time on analysis activities.
- In later iterations it is common that analysis lessens; there's more focus on just building the solution.

## Provoking Early Change

- It is natural and healthy to discover and change some requirements during the design and implementation work, especially in the early iterations.
- Iterative and evolutionary methods “embrace change” although we try to provoke that inevitable change in early iterations,
- so that we have a more stable goal (and estimate and schedule) for the later iterations.
- Early programming, tests, and demos help provoke the inevitable changes early on.
- This sounds a simple idea, yes, it lies at the heart of why iterative development works.

## Provoking Early Change (contd.)

- Over the course of these early elaboration iterations, the requirements discovery should stabilize, so that by the end of elaboration, perhaps 80% of the requirements are reliably defined—defined and refined as a result of feedback, early programming and testing, rather than speculation, as occurs in a waterfall method.

## Didn't All That Analysis and Modeling Take Weeks To Do?

- When one is comfortable with the skills of use case writing, domain modeling, and so forth, the duration to do all the actual modeling that has been explored so far is realistically just a few hours or days.
- However, that does not mean that only a few days have passed since the start of the project.
- Many other activities, such as proof-of-concept programming, finding resources (people, software,...), planning, setting up the environment, and so on, could consume a few weeks of preparation.

# On to Object Design

## Introduction

- How do you design objects?
  - ① Code
    - Design-while-coding (Java, C#, ...). From mental model to code.
  - ② Draw, then code
    - Drawing some UML on a whiteboard or UML CASE tool, then switching to #1 with a text-strong IDE (e.g., Eclipse or Visual Studio).
  - ③ Only draw
    - Somehow, the tool generates everything from diagrams. Many tool vendor has washed onto the shores of this steep island. "Only draw" is a misnomer, as this still involves a text programming language attached to UML graphic elements.
- If we use **Draw, then code** (the most popular approach with UML), the drawing overhead should be worth the effort.

## Agile Modeling and Lightweight UML Drawing

- Reduce drawing overhead and model to understand and communicate, rather than to document.
- Modeling with others.
- Creating several models in parallel.

For example,

five minutes on a wall of interaction diagrams, then five minutes on a wall of related class diagrams.

## UML CASE Tools: Guidelines

- Choose a UML CASE tool that integrates with popular text-strong IDEs, such as Eclipse or Visual Studio.
- Choose a UML tool that can reverse-engineer (generate diagrams from code) not only class diagrams (common), but also interaction diagrams (more rare, but very useful to learn call-flow structure of a program).
- Many developers find it useful to code awhile in their favorite IDE, then press a button, reverse-engineer the code, and see a UML big-picture graphical view of their design.

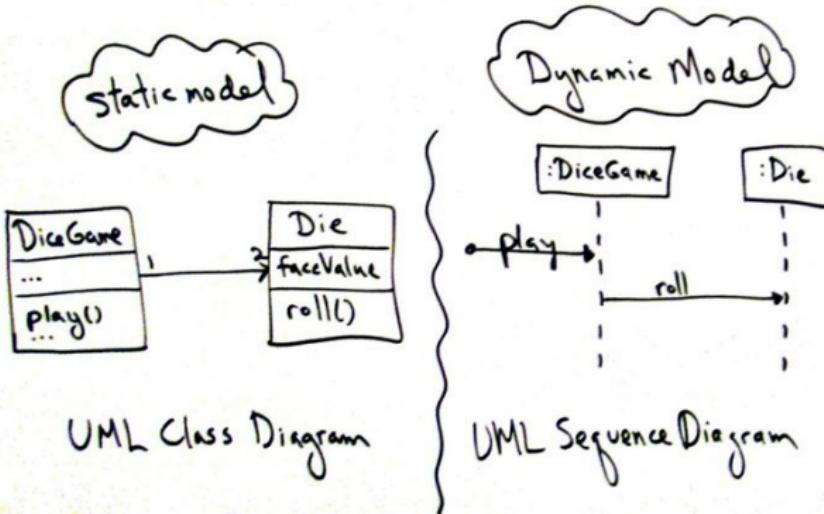
## How Much Time Spent Drawing UML Before Coding? Guideline

- For a three-week timeboxed iteration, spend a few hours or at most one day (with partners) near the start of the iteration “at the walls” (or with a UML CASE tool) drawing UML for the hard, creative parts of the detailed object design.
- Then stop - and if sketching-perhaps take digital photos, print the pictures, and transition to coding for the remainder of the iteration, using the UML drawings for inspiration as a starting point, but recognizing that the final design in code will diverge and improve.
- Shorter drawing/sketching sessions may occur throughout the iteration.

## Designing Objects

- Two kinds of object models: dynamic and static.
- **Dynamic models**, such as UML interaction diagrams (sequence diagrams or communication diagrams), help design the logic, the behavior of the code or the method bodies.
- They tend to be the more interesting, difficult, important diagrams to create.
- **Static models**, such as UML class diagrams, help design the definition of packages, class names, attributes, and method signatures (but not method bodies).

## Static and Dynamic Modeling



## Dynamic Object Modeling

- There's a relationship between static and dynamic modeling and the agile modeling practice of create models in parallel:
  - Spend a short period of time on interaction diagrams (dynamics), then switch to a wall of related class diagrams (statics).
- Guidelines
  - Spend significant time doing interaction diagrams (sequence or communication diagrams), not just class diagrams.
  - Ignoring this guideline is a very common worst-practice with UML.
- Note that it's especially during dynamic modeling that we apply responsibility-driven design and the GRASP principles.
- There are other dynamic tools in the UML kit, including **state machine diagrams** and **activity diagrams**.

## Static Object Modeling

- The most common static object modeling is with UML class diagrams.
- Note, though, that if the developers are applying the agile modeling practice of create several models in parallel, they will be drawing both interaction and class diagrams concurrently.
- Other support in the UML for static modeling includes **package diagrams** and **deployment diagrams**.

## The Importance of Object Design Skill over UML Notation Skill

- It's been said before, but is important to stress:
  - What's important is knowing how to think and design in objects, and apply object design best-practice patterns, which is a very different and much more valuable skill than knowing UML notation.
- While drawing a UML object diagram, we need to answer key questions:
  - What are the responsibilities of the object?
  - Who does it collaborate with?
  - What design patterns should be applied?

## Object Design Skill vs. UML Notation Skill

- Drawing UML is a reflection of making decisions about the design.
- The object design skills are what matter, not knowing how to draw UML.
- Fundamental object design requires knowledge of:
  - principles of responsibility assignment
  - design patterns

# CRC Cards

## CRC Cards

- A popular text-oriented modeling technique is Class Responsibility Collaboration (CRC) cards, created by the agile, influential minds of Kent Beck and Ward Cunningham (also founders of the ideas of XP and design patterns).
- CRC cards are paper index cards on which one writes the **responsibilities** and **collaborators** of classes.
- Each card represents one class.
- A CRC modeling session involves a group sitting around a table, discussing and writing on the cards as they play "what if" scenarios with the objects, considering what they must do and what other objects they must collaborate with.

## CRC Cards: An Example

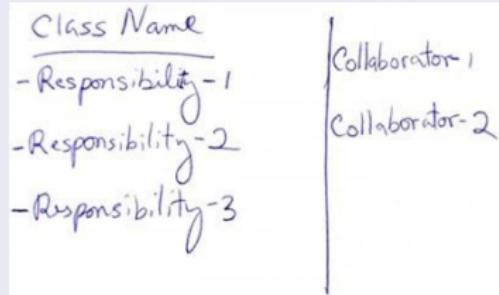


Figure: *Template for a CRC card.*

## CRC Cards: Class

- A Class represents a collection of similar objects.
- Objects are things of interest in the system being modeled.
- They can be a person, place, thing, or any other concept important to the system at hand.
- The Class name appears across the top of the CRC card.

## CRC Cards: Responsibilities

- Responsibilities of a class can be broken into two separate types:  
**knowing** and **doing**.

## CRC Cards: Responsibilities: Knowing

- Knowing responsibilities are those things that an instance of a class must be capable of knowing.
- An instance of a class typically knows the values of its attributes and its relationships.

## CRC Cards: Responsibilities: Doing

- Doing responsibilities are those things that an instance of a class must be capable of doing.
- In this case, an instance of a class can execute its operations or it can request a second instance, which it knows about, to execute one of its operations on behalf of the first instance.

## CRC Cards: Collaborator

- A Collaborator is another class that is used to get information for, or perform actions for the class at hand.
- It often works with a particular class to complete a step (or steps) in a scenario.
- The Collaborators of a class appear along the right side of the CRC card.

## Developing CRC Cards

- The answers to the following questions are used to add detail to the evolving CRC cards.
  - Who or what are you?
  - What do you know?
  - What can you do?

# Elements of a CRC Card

Front:		
Class Name: Old Patient	ID: 3	Type: Concrete, Domain
Description: An individual who needs to receive or has received medical attention		Associated Use Cases: 2
Responsibilities		Collaborators
Make appointment	Appointment	
Calculate last visit		
Change status		
Provide medical history	Medical history	
Back:		
Attributes:		
Amount (double)		
Insurance carrier (text)		
Relationships:		
Generalization (a-kind-of):	Person	
Aggregation (has-parts):	Medical History	
Other Associations:	Appointment	

Figure: Sample CRC Card

## It's Quiz Time

- ① The design work will stress do the thing right. (True or False)
- ② Agile modeling does not support “modeling with others”. (True or False)
- ③ Dynamic models help design the definition of packages, class names, attributes, and method signatures. (True or False)

# Object-Oriented Software Analysis and Design

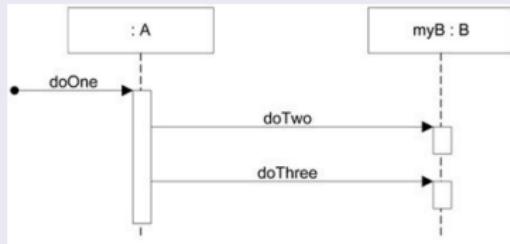
School of Computer Science  
University of Windsor

## UML Interaction Diagrams

- The UML includes interaction diagrams to illustrate **how objects interact via messages**.
- They are used for **dynamic object modeling**.
- The term interaction diagram is a generalization of two more specialized UML diagram types:
  - Sequence diagrams
  - Communication diagrams
- *Both can express similar interactions.*
- A related diagram is the **interaction overview diagram**;
  - it provides a big-picture overview of how a set of interaction diagrams are related in terms of logic and process-flow.

## Sequence diagrams

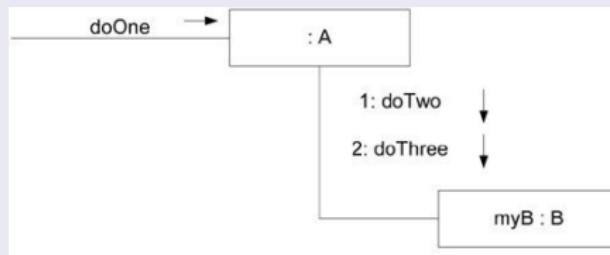
- Sequence diagrams illustrate interactions **in a kind of fence format**, in which each new object is added to the right.



```
public class A
{
    private B myB = new
        B();
    public void doOne()
    {
        myB.doTwo();
        myB.doThree();
    }
    // ...
}
```

## Communication diagrams

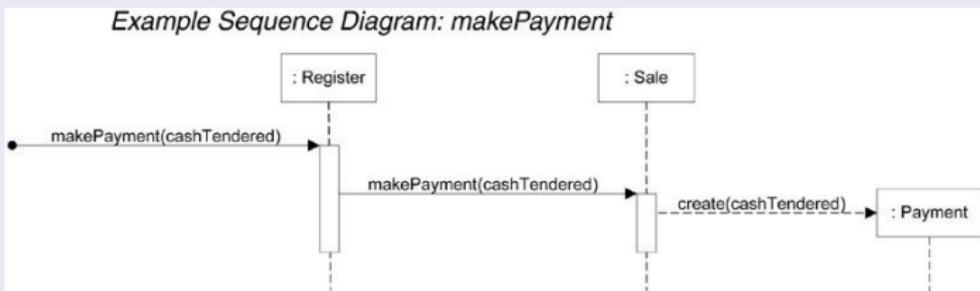
- Communication diagrams illustrate object interactions **in a graph or network format**, in which objects can be placed anywhere on the diagram



## Strengths and Weaknesses of Sequence vs. Communication Diagrams

Type	Strengths	Weaknesses
sequence	<p>clearly shows sequence or time ordering of messages</p> <p>large set of detailed notation options</p>	forced to extend to the right when adding new objects; consumes horizontal space
communication	<p>space economical flexibility to add new objects in two dimensions</p>	<p>more difficult to see sequence of messages</p> <p>fewer notation options</p>

## Example Sequence Diagram: *makePayment*



- The message `makePayment` is sent to an instance of a `Register`. The sender is not identified.
- The `Register` instance sends the `makePayment` message to a `Sale` instance.
- The `Sale` instance creates an instance of a `Payment`.

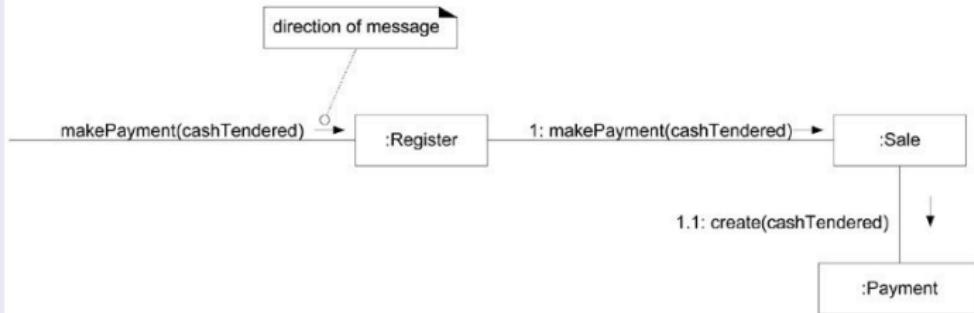
## Example Sequence Diagram: makePayment

```
public class Sale
{
    private Payment payment;

    public void makePayment( Money cashTendered )
    {
        payment = new Payment( cashTendered );
        //...
    }
    //...
}
```

## Example Communication Diagram: makePayment

*Example Communication Diagram: makePayment*



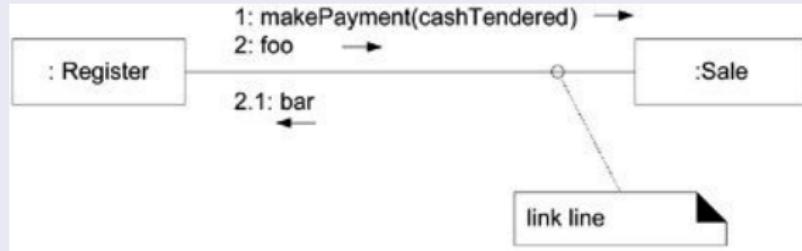
- The message `makePayment` is sent to an instance of a `Register`. The sender is not identified.
- The `Register` instance sends the `makePayment` message to a `Sale` instance.
- The `Sale` instance creates an instance of a `Payment`.

## UML Interaction Diagrams: Guideline

- Spend time doing dynamic object modeling with interaction diagrams, not just static object modeling with class diagrams.
- **Why?** Because it's when we have to think through the concrete details of what messages to send, and to whom, and in what order, in terms of thinking through the true OO design details.

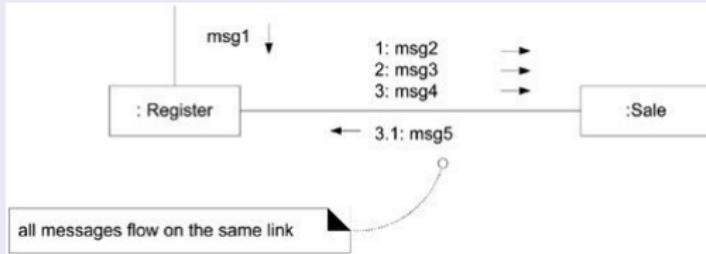
# Basic Communication Diagram Notation

## Links



- A **link** is a connection path between two objects
- It indicates some form of **navigation and visibility** between the objects is possible.
- More formally, a link is an **instance of an association**.
- For example, there is a link or path of navigation from a Register to a Sale, along which messages may flow, such as the `makePayment` message.

## Messages



- Each message between objects is represented with a **message expression and small arrow** indicating the direction of the message.
- **Many messages** may flow along this link.
- A **sequence number** is added to show the sequential order of messages in the current thread of control.
- **Guideline:** Don't number the starting message. It's legal to do so, but simplifies the overall numbering if you don't.

## Basic Message Expression Syntax

- Interaction diagrams show messages between objects; the UML has a standard syntax for these message expressions:

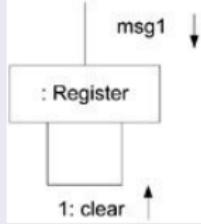
```
return = message(parameter:parameterType):returnType
```

- Parentheses are usually excluded if there are no parameters, though still legal.
- Type information may be excluded if obvious or unimportant.

For example,

```
initialize(code)
initialized
d = getProductDescription(id)
d = getProductDescription(id:ItemID)
d = getProductDescription(id:ItemID) : ProductDescription
```

## Messages to “self” or “this”

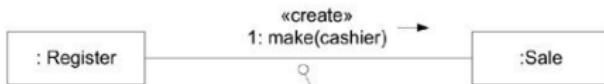
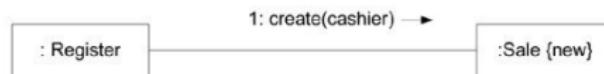
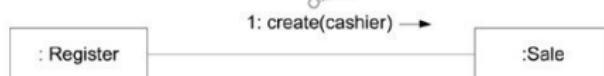


- A message can be sent **from an object to itself**. This is illustrated by a link to itself, with messages flowing along the link.

## Creation of Instances

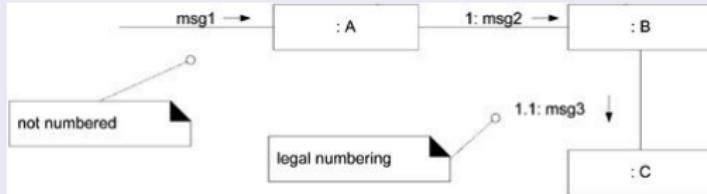
Three ways to show creation in a communication diagram

create message, with optional initializing parameters. This will normally be interpreted as a constructor call.



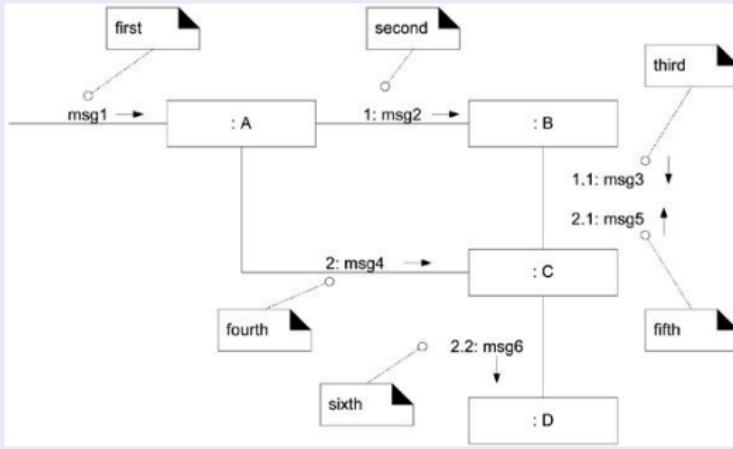
if an unobvious creation message name is used, the message may be stereotyped for clarity

## Message Number Sequencing

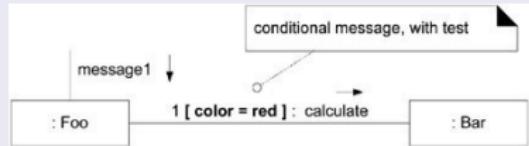


- The first message is not numbered. Thus, `msg1` is unnumbered.
- The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have a number appended to them.
- You denote nesting by prepending the incoming message number to the outgoing message number.

## Message Number Sequencing (contd.)

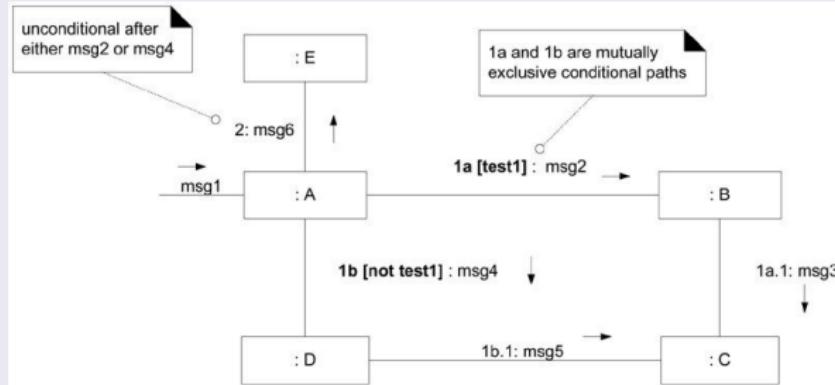


## Conditional Messages



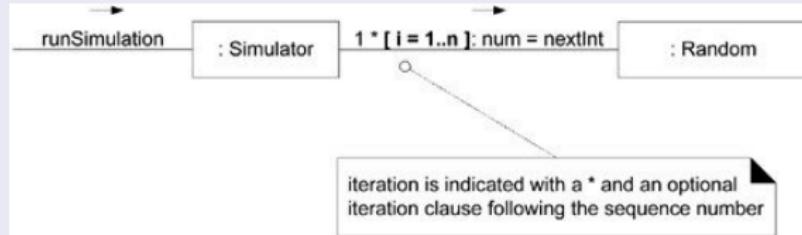
- You show a conditional message by following a sequence number with a conditional clause in square brackets, similar to an iteration clause. The message is only sent if the clause evaluates to **true**.

## Mutually Exclusive Conditional Paths



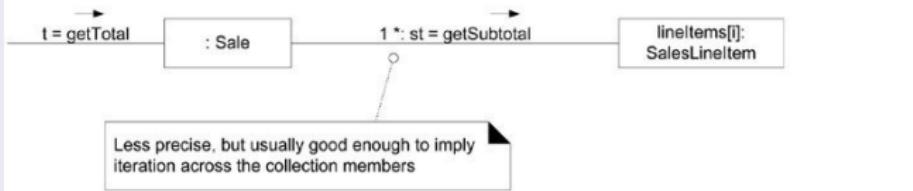
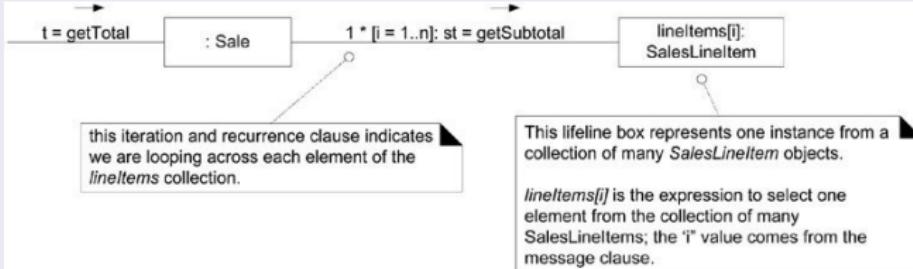
- In this case we must modify the sequence expressions with a conditional path letter. The first letter used is a by convention. Figure above states that either 1a or 1b could execute after msg1. Both are sequence number 1 since either could be the first internal message.
- Note that subsequent nested messages are still consistently prepended with their outer message sequence. Thus 1b.1 is nested message within 1b.

## Iteration or Looping



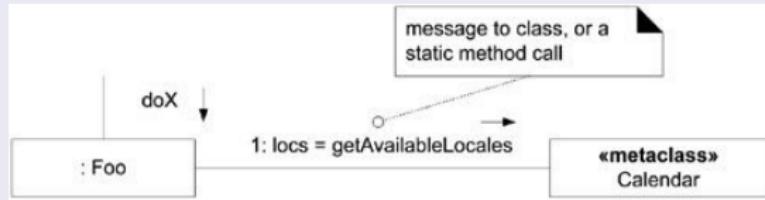
- If the details of the iteration clause are not important to the modeler, a simple \* can be used.

## Iteration Over a Collection

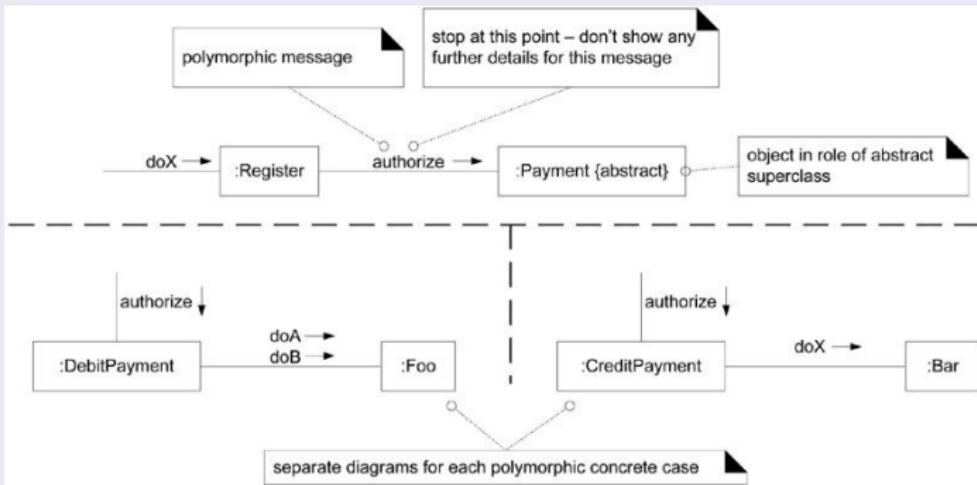


- A common algorithm is to iterate over all members of a collection (such as a list or map), sending the same message to each.

## Messages to a Classes to Invoke Static (Class) Methods

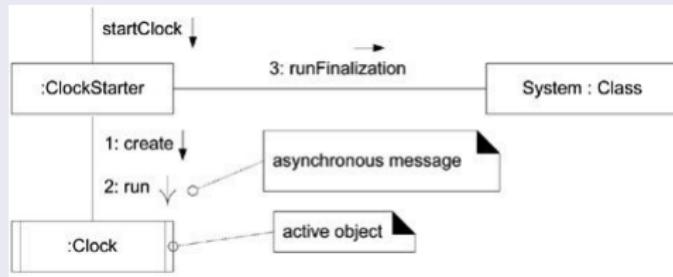


## Polymorphic Messages and Cases



- As in the sequence diagram case, multiple communication diagrams can be used to show each concrete polymorphic case.

## Asynchronous and Synchronous Calls



- As in sequence diagrams, asynchronous calls are shown with a stick arrow; synchronous calls with a filled arrow.

## It's Quiz Time

- ① Communication diagrams illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram. (True or False)
- ② Communication diagram forced to extend to the right when adding new objects; consumes horizontal space. (True or False)
- ③ The term interaction diagram is a generalization of two more specialized UML diagram types: sequence diagrams and communication diagrams. (True or False)

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

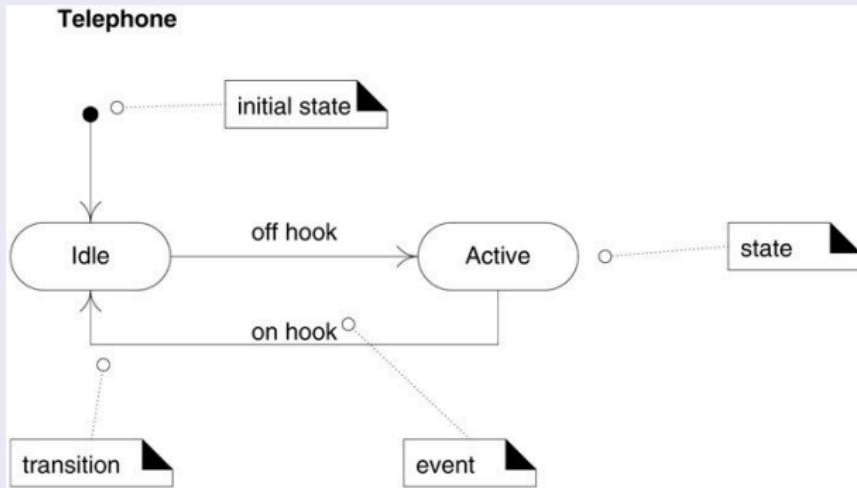
## UML State Machine Diagrams

- A UML state machine diagram, illustrates the interesting **events** and **states** of an object, and the **behavior** of an object in reaction to an event.
- **Transitions** are shown as arrows, labeled with their event.
- **States** are shown in rounded rectangles.
- It is common to include an **initial pseudo-state**, which automatically transitions to another state when the instance is created.

## UML State Machine Diagrams (contd.)

- A state machine diagram shows the lifecycle of an object:
  - what **events** it experiences,
  - its **transitions**, and
  - the **states** it is in between these events.
- *It need not illustrate every possible event*; if an event arises that is not represented in the diagram, the event is ignored as far as the state machine diagram is concerned.
- Therefore, we can create a state machine diagram that describes the lifecycle of an object at arbitrarily **simple or complex levels of detail, depending on our needs**.

## UML State Machine Diagrams: Example



*Figure: State machine diagram for a telephone.*

## UML State Machine Diagrams: Definitions

- An **event** is a significant or noteworthy occurrence.

For example,

A telephone receiver is taken off the hook.

- A **state** is the condition of an object at a moment in time—the time between events.

For example,

A telephone is in the state of being “idle” after the receiver is placed on the hook and until it is taken off the hook.

- A **transition** is a relationship between two states that indicates that when an event occurs, the object moves from the prior state to the subsequent state.

For example,

When the event “off hook” occurs, transition the telephone from the “idle” to “active” state.

## How to Apply State Machine Diagrams? State-Independent Objects

- If an object always responds the same way to an event, then it is considered state-independent (or modeless) with respect to that event.

For example,

if an object receives a message, and the responding method always does the same thing. The object is state-independent with respect to that message.

## How to Apply State Machine Diagrams? State-Dependent Objects

- By contrast, state-dependent objects react differently to events depending on their state or mode.

For example,

a telephone is very state-dependent. The phone's reaction to pushing a particular button (generating an event) depends on the current mode of the phone off hook, engaged, in a configuration subsystem, and so forth.

- It's for these kind of complex state-dependent problems that a state machine diagram may add value to either understand or document something.

## How to Apply State Machine Diagrams? State-Independent and State-Dependent Objects

- **Guideline:**

- Consider state machines for state-dependent objects with complex behavior, not for state-independent objects.
- In general, business information systems have few complex state-dependent classes. It is seldom helpful to apply state machine modeling.
- By contrast, process control, device control, protocol handlers, and telecommunication domains often have many state-dependent objects. If you work in these domains, definitely know and consider state machine modeling.

## How to Apply State Machine Diagrams? Modeling State-dependent Objects

- Broadly, state machines are applied in two ways:
  - ① To model the behavior of a complex reactive object in response to events.
  - ② To model legal sequences of operations protocol or language specifications.
    - This approach may be considered a specialization of #1, if the "object" is a language, protocol, or process. A formal grammar for a context-free language is a kind of state machine.
- The following is a list of common objects which are often state-dependent, and for which it may be useful to create a state machine diagram:
  - Complex Reactive Objects
  - Protocols and Legal Sequences

## How to Apply State Machine Diagrams? Modeling State-dependent Objects- Complex Reactive Objects

- Physical Devices controlled by software
- Transactions and related Business Objects
- Role Mutators

## How to Apply State Machine Diagrams? Modeling State-dependent Objects- Protocols and Legal Sequences

- Communication Protocols
- UI Page/Window Flow or Navigation
- UI Flow Controllers or Sessions
- Use Case System Operations
- Individual UI Window Event Handling

## Transition Actions and Guards

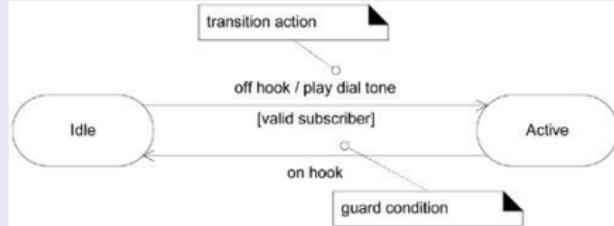


Figure: *Transition action and guard notation.*

- A transition can cause an action to fire. In a software implementation, this may represent the invocation of a method of the class of the state machine diagram.
- A transition may also have a conditional guard or boolean test. The transition only occurs if the test passes.

## Nested States

- A state allows nesting to contain substates; a **substate** inherits the transitions of its **superstate** (the enclosing state).
- Substates may be graphically shown by nesting them in a superstate box.

## Nested States (contd.)

For example,

when a transition to the Active state occurs, creation and transition into the PlayingDialTone substate occurs. No matter what substate the object is in, if the on hook event related to the Active superstate occurs, a transition to the Idle state occurs.

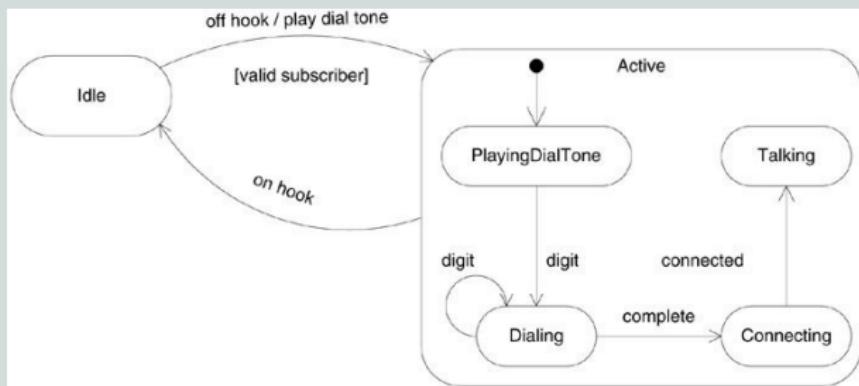


Figure: Nested states.

## UML State Machine Diagrams: Example (UI Navigation Modeling)

- Some UI applications, especially Web UI applications, have complex page flows.
- State machines are a great way to document that, for understanding, and a great way to model page flows, during creative design.
- The states represent the pages and the events represent the events that cause transfer from one page to another, such as a button click.

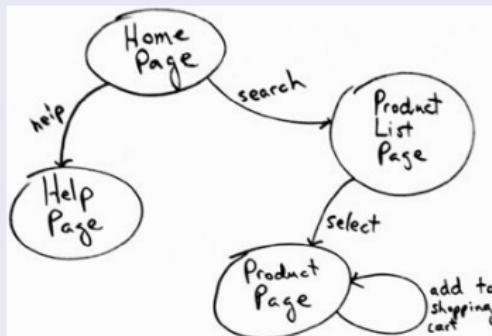


Figure: Applying a state machine to Web page navigation modeling.

## UML State Machine Diagrams: Example (NextGen POS Use Case)

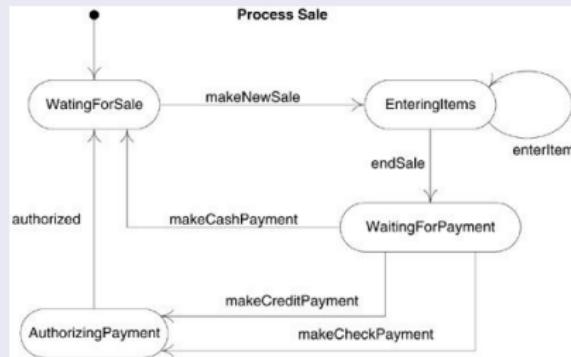


Figure: A sample state machine for legal sequence of use case operations.

## It's Quiz Time

- ① A ..... is a relationship that represents the movement of an object from one state to another state.
  - ① event
  - ② state
  - ③ transition
- ② An ..... is something that takes place at a certain point in time and changes a value or values that describe an object, which, in turn, changes the object's state.
  - ① event
  - ② state
  - ③ transition
- ③ A transition may also have a conditional guard or boolean test. The transition only occurs if the test passes. (True or False)

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## UML Class Diagrams

- The UML includes class diagrams to illustrate classes, interfaces, and their associations.
- They are used for **static object modeling**.

# Applying UML: Common Class Diagram Notation

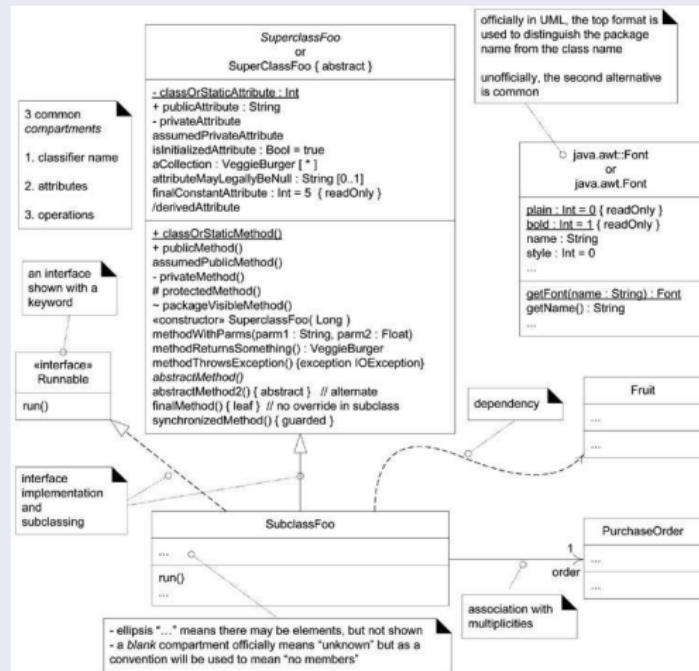


Figure: Common UML class diagram notation.

## Definition: Design Class Diagram (DCD)

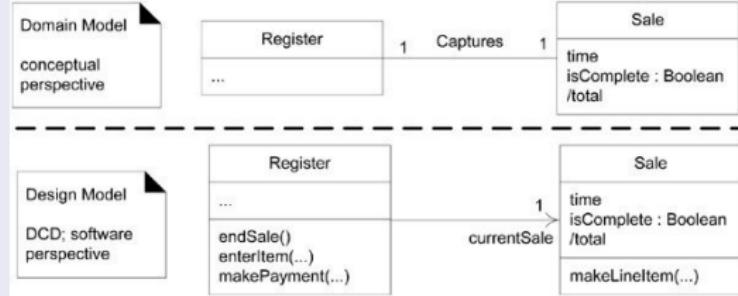


Figure: UML class diagrams in two perspectives.

## Definition: Classifier

- A UML classifier is “a model element that describes behavioral and structure features”.
- Classifiers can also be specialized.
- They are a generalization of many of the elements of the UML, including classes, interfaces, use cases, and actors.
- In class diagrams, the two most common classifiers are **regular classes** and **interfaces**.

## Ways to Show UML Attributes: Attribute Text and Association Lines

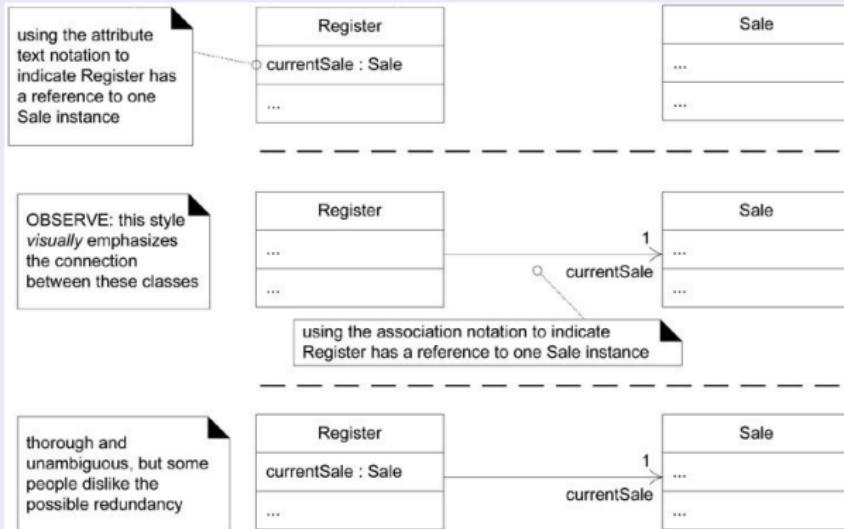


Figure: *Attribute text versus association line notation for a UML attribute.*

- The full format of the attribute text notation is:  
`visibility name : type multiplicity = default property-string`
- **Guideline:** Attributes are usually assumed private if no visibility is given.

## Ways to Show UML Attributes: Attribute Text and Association Lines

(contd.)

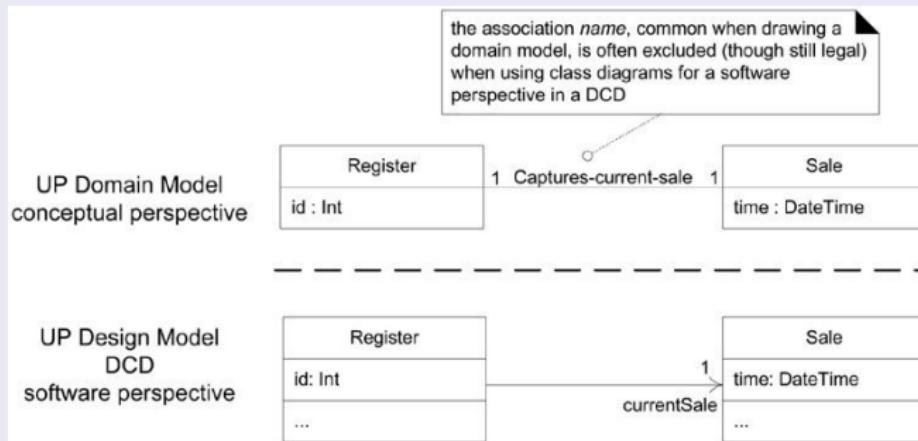


Figure: *Idioms in association notation usage in different perspectives.*

## Guideline: When to Use Attribute Text versus Association Lines for Attributes?

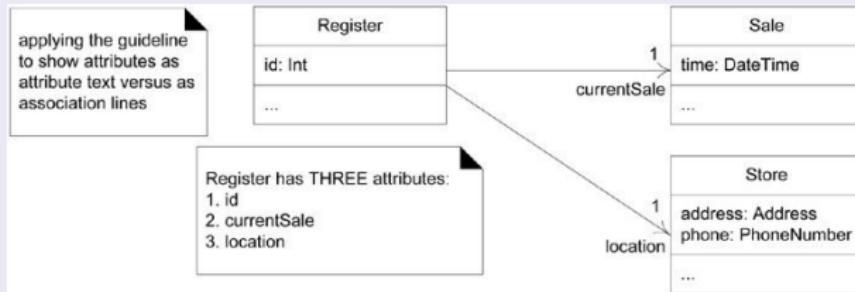


Figure: Applying the guidelines to show attributes in two notations.

## The UML Notation for an Association End

- The end of an association can have a **navigability arrow**.
- It can also include an optional **rolename** (officially, an association end name) to indicate the attribute name.
- And of course, the association end may also show a **multiplicity** value, such as '\*' or '0..1'.

## How to Show Collection Attributes with Attribute Text and Association Lines?

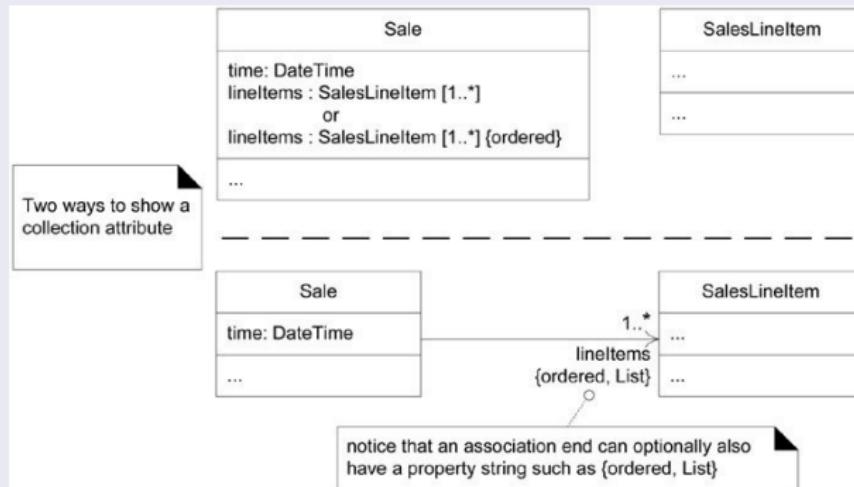


Figure: *Two ways to show a collection attribute in the UML.*

## Note Symbols: Notes, Comments, Constraints, and Method Bodies



Figure: How to show a method body in a class diagram.

- A UML **note symbol** is displayed as a dog-eared rectangle with a dashed line to the annotated element.
- A note symbol may represent several things, such as:
  - a UML **note** or **comment**, which by definition have no semantic impact
  - a UML **constraint**, in which case it must be encased in braces '{...}'
  - a **method** body—the implementation of a UML operation

## Operations

- The full, official format of the operation syntax is:  
`visibility name (parameter-list)property-string`
- Notice there is no **return** type element, an obvious problem, but purposefully injected into the UML 2 specification for inscrutable reasons.  
`visibility name (parameter-list): return-type property-string`
- **Guideline:** Assume the version that includes a return type.
- **Guideline:** Operations are usually assumed **public** if no visibility is shown.

## Operations (contd.)

- **Accessing operations** retrieve or set attributes, such as `getPrice` and `setPrice`.
- These operations are often excluded (or filtered) from the class diagram because of the high noise-to-value ratio they generate; for  $n$  attributes, there may be  $2n$  uninteresting **getter** and **setter** operations.
- Most UML tools support filtering their display, and it's especially common to ignore them while wall sketching.

## Keywords

- A UML keyword is a textual adornment to categorize a model element.

For example,

the keyword to categorize that a classifier box is an interface is «interface».

Keyword	Meaning	Example Usage
«actor»	classifier is an actor	in class diagram, above classifier name
«interface»	classifier is an interface	in class diagram, above classifier name
{abstract}	abstract element; can't be instantiated	in class diagrams, after classifier name or operation name
{ordered}	a set of objects have some imposed order	in class diagrams, at an association end

Figure: A few sample predefined UML keywords

## Decomposition

- The design principle of decomposition takes a whole thing and divides it into different parts.
- Alternately, decomposition can also indicate taking separate parts with different functionalities and combining them to create a whole.
- Decomposition allows problems to be broken into smaller pieces that are easier to understand and solve.
- There are three types of relationships in decomposition, which define the interaction between the whole and the parts:
  - ① Association
  - ② Aggregation
  - ③ Compositon

## Decomposition: Association

- Association indicates a **loose relationship** between two objects, which may interact with each other for some time.
- They are not dependent on each other—if one object is destroyed, the other can continue to exist, and there can be any number of each item in the relationship.
- One object does not belong to another, and they may have numbers that are not tied to each other.

## Association in Java and UML

- An example of an association relationship could be a `person` and a `hotel`.
- A `person` might interact with a `hotel` but not own one. A `hotel` may interact with many people.
- Association is represented in UML diagrams as below:



## Aggregation

- Aggregation is a “has-a” relationship where a whole has parts that belong to it.
- Parts may be shared among wholes in this relationship.
- Aggregation relationships are typically **weak**. This means that although parts can belong to wholes, they can also exist independently.

## Aggregation in Java and UML

- Aggregation can be represented in UML class diagrams with the symbol of an empty diamond as below:



- The empty diamond indicates which object is considered the whole and not the part in the relationship.

## Aggregation in Java and UML (contd.)

- Aggregation can be represented in Java code as below:

```
public class Airliner {  
    private ArrayList<CrewMember> crew;  
    public Airliner() {  
        crew = new ArrayList<CrewMember>();  
    }  
    public void add( CrewMember crewMember ) {  
        ...  
    }  
}
```

- In the `Airliner` class, there is a list of `crew` members.
- The list of `crew` members is initialized to be empty and a public method allows new `crew` members to be added.
- An `airliner` has a `crew`. This means that an `airliner` can have zero or more `crew` members.

## Composition

- Composition is one of the most dependent of the decomposition relationships.
- This relationship is an exclusive containment of parts, otherwise known as a **strong** “has-a” relationship.
- In other words, a whole cannot exist without its parts, and if the whole is destroyed, then the parts are destroyed too.
- In this relationship, you can typically only access the parts through its whole.
- Contained parts are exclusive to the whole.

## Composition in Java and UML

- An example of a composition relationship is between a `house` and a `room`. A `house` is made up of multiple `rooms`, but if you remove the `house`, the `room` no longer exists.
- Composition can be represented with a filled-in diamond using UML class diagrams, as below:



- The filled-in diamond next to the `House` object means that the `house` is the whole in the relationship.
- If the diamond is filled-in, it symbolizes that the “has-a” relationship is **strong**.

## Composition in Java and UML (contd.)

- Composition can be represented using Java code as well.

```
public class House {  
    private Room room;  
  
    public House(){  
        room = new Room();  
    }  
}
```

- In this example, a `Room` object is created at the same time that the `House` object is, by instantiating the `Room` class.
- This `Room` object does not need to be created elsewhere, and it does not need to be passed in when creating the `House` object.
- The two parts are tightly dependent with one not being able to exist without the other.

## Generalization

- The design principle of generalization takes repeated, common, or shared characteristics between two or more classes and factors them out into another class, so that code can be reused, and the characteristics can be inherited by subclasses.
- Generalization and inheritance can be represented UML class diagrams using a solid-lined arrow as shown below:

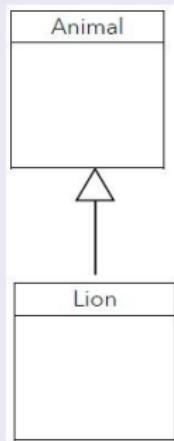


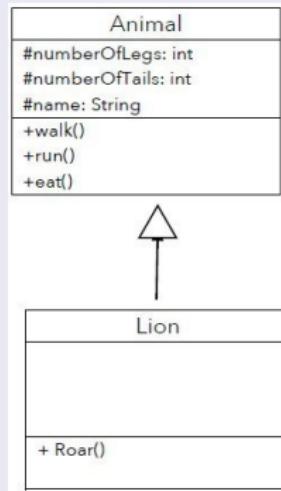
Figure: *Class diagram*

## Generalization in Java and UML

- The solid-lined arrow indicates that two classes are connected by inheritance.
- The superclass is at the head of the arrow, while the subclass is at the tail.
- It is conventional to have the arrow pointing upwards.
- The class diagram is structured so that superclasses are always on top and subclasses are towards the bottom.
- Inherited superclass' attributes and behaviors do not need to be rewritten in the subclass. Instead, the arrow symbolizes that the subclass will have the superclass' attributes and methods.
- Superclasses are the generalized classes, and the subclasses are the specialized classes.

## Generalization in Java and UML (contd.)

- It is possible to translate UML class diagrams into code. Let us build on the example above.



## Generalization in Java and UML (contd.)

- The UML class diagrams can be translated into code.

```
public abstract class Animal {  
    protected int numberofLegs;  
    protected int numberofTails;  
    protected String name;  
  
    public Animal( String petName, int legs, int tails ) {  
        this.name = petName;  
        this.numberofLegs = legs;  
        this.numberofTails = tails;  
    }  
    public void walk() { ... }  
    public void run() { ... }  
    public void eat() { ... }  
}
```

## Generalization in Java and UML (contd.)

- Since the `Animal` class is a generalization, it should not be created as an object on its own. The keyword `abstract` indicates that the class cannot be instantiated. In other words, an `Animal` object cannot be created.
- Here is the code for creating a `Lion` subclass:

```
public class Lion extends Animal {  
    public Lion(String name, int legs, int tails) {  
        super( name, legs, tails );  
    }  
    public void roar() { ... }  
}
```

- This mirrors the UML class diagram, as only specialized attributes and methods are declared in the superclass and subclass.
- Inheritance is declared in Java using the keyword `extends`.

## Generalization in Java and UML (contd.)

Classes can have **implicit** constructors or **explicit** constructors.

- Below is an example of an **implicit constructor**:

```
public abstract class Animal {  
    protected int numberOfLegs;  
  
    public void walk() { ... }  
}
```

- In this implementation, we have not written our own constructor. All attributes are assigned zero or null when using the default constructor.

## Generalization in Java and UML (contd.)

- Below is an example of an **explicit** constructor:

```
public abstract class Animal {  
    protected int number0fLegs;  
  
    public Animal( int legs ) {  
        this.number0fLegs = legs;  
    }  
}
```

- In this implementation, an explicit constructor will let us instantiate an animal with as many legs we want. Explicit constructors allow you to assign values to attributes during instantiation.

```
public class Lion extends Animal {  
    public Lion( int legs ) {  
        super( legs );  
    }  
}
```

## Generalization in Java and UML (contd.)

- Subclasses can override the methods of its superclass, meaning that a subclass can provide its own implementation for an inherited superclass' method.

```
public abstract class Animal {  
    protected int number0fLegs;  
    public void walk() {  
        System.out.println("Animal is walking");  
    }  
}  
  
public class Lion extends Animal {  
    public void walk() {  
        System.out.println("I'd rather nap");  
    }  
}
```

- In the above example, the `Lion` class has overridden the `Animal` class's `walk` method. If asked to walk, the system would tell us that a `Lion` object would rather nap.

## Single Inheritance

- In Java, only single implementation inheritance is allowed. This means that while a superclass can have multiple subclasses, a subclass can only inherit from a single superclass.

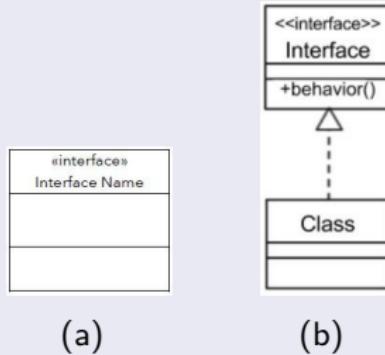
For example,

the `Animal` class might be a superclass to multiple subclasses: a `Lion` class, a `Wolf` class, or a `Deer` class.

- Subclasses can also be a superclass to another class. Inheritance can trickle down through as many classes as desired.

## Interface Inheritance

- Interfaces can be drawn in a similar way to classes in UML diagrams. Interfaces are explicitly noted using guillemets, or French quotes, to surround the word «interface».



## Interface Inheritance (contd.)

- In Java, the keyword `interface` is used to indicate that one is being defined. The letter “I” is sometimes placed before an actual name to indicate an interface.

```
public interface IAnimal {  
    public void move();  
    public void speak();  
    public void eat();  
}
```

- In order to use an interface, you must declare that you are going to fulfill the contract as described in the interface. The keyword in Java for this action is `implements`.

```
public class Lion implements IAnimal {  
    /* Attributes of a lion can go here */  
    public void move() {...}  
    public void speak() {...}  
    public void eat() {...}  
}
```

## Constraints

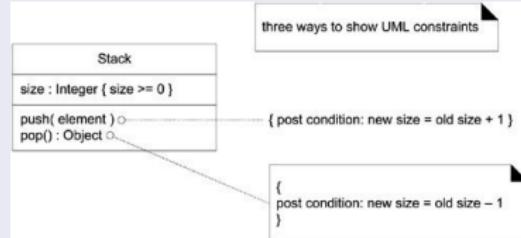


Figure: *Constraints*

- Constraints may be used on most UML diagrams, but are especially common on class diagrams.
- A UML constraint is a restriction or condition on a UML element. It is visualized in text between braces;

For example,

`size >= 0.`

- The text may be natural language or anything else, such as UML's formal specification language, the Object Constraint Language (OCL) (as shown in the Figure).

## Singleton Classes

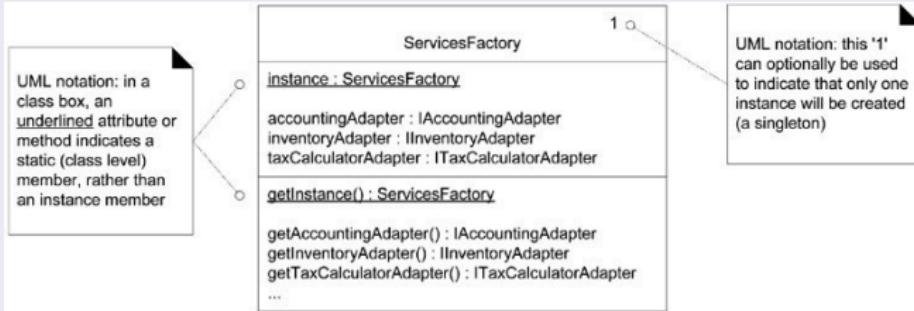


Figure: *Showing a singleton.*

- In the world of OO design patterns, there is one that is especially common, called the **Singleton pattern**.
- It is explained later, but an implication of the pattern is that there is only one instance of a class instantiated never two.
- In other words, it is a “singleton” instance.
- In a UML diagram, such a class can be marked with a ‘1’ in the upper right corner of the name compartment (as shown in the Figure).

## User-Defined Compartments



Figure: *Compartments.*

- In addition to common predefined compartments class compartments such as name, attributes, and operations, user-defined compartments can be added to a class box (as shown in the Figure).

## What's the Relationship Between Interaction and Class Diagrams?

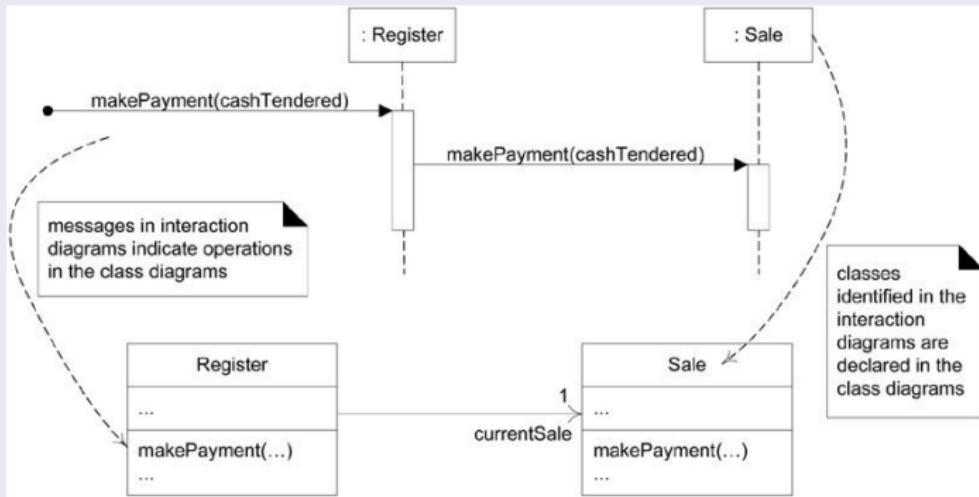


Figure: *The influence of interaction diagrams on class diagrams.*

## It's Quiz Time

- ① UML class diagrams are used for dynamic object modeling. (True or False)
- ② Compared with composition, aggregation relationships are typically weak. (True or False)
- ③ The relationship between a house and a room is an example of a composition relationship. (True or False)

# Object-Oriented Software Analysis and Design

School of Computer Science  
University of Windsor

## Introduction to Design Patterns

- There are many ways to deal with **software design problems**, but more **flexible** or **reusable** solutions are preferred in the industry.
- One of these preferred methods is that of **design patterns**.

## Introduction to Design Patterns (contd.)

- A design pattern is a **practical proven solution** to a recurring design problem.
- It **allows you to use previously outlined solutions that expert developers have often used to solve a software problem**, so that you do not need to build a solution from the basics of object-oriented programming principles every time.
- These solutions are **not just theoretical**—they are **actual solutions used in the industry**.
- Design patterns may also be **used to help fix tangled, structureless software code**, also known as “spaghetti code.”

## Introduction to Design Patterns (contd.)

- It is not always obvious which design pattern to use to solve a software design problem, especially as many design patterns exist.
- **Practice and experience** will make you better at selecting which design patterns to use for different problems. Some people even become **design experts** who know the design patterns to use in solving particular software design problems!

## Introduction to Design Patterns (contd.)

- It is important to understand that design patterns are **not just a concrete set of source code that you memorize and put into your software**, like a Java library or framework.
- Instead, **design patterns are more conceptual**. They are knowledge that you can apply within your software design, to guide its structure, and make it more flexible and reusable.
- Design patterns **help software developers** so that they have a guide to help them solve design problems the way an expert might, so not everything needs to be built from scratch.
- Design patterns **serve almost like a coach to help developers** reach their full potential!

## Introduction to Design Patterns (contd.)

- A **strong advantage** of design patterns is that they have already been **proven by experts**.
  - This means that you do not need to go through the trials they have, and you go straight to creating better written software.
- Another **advantage** of design patterns is that they **help create a design vocabulary**.
  - This means that design patterns provide a simplified means of discussing design solutions, so that they do not need to be explained over and over.

For example,

design patterns might give **patterns names**, making it easier to discuss them. This saves time, and ensures that everyone is referring to the same pattern. This leaves less room for misunderstanding

## Gang of Four's Pattern Catalogue

- The four authors of the famous book **Design Patterns: Elements of Reusable Object-Oriented Software**—Gamma, Helm, Johnson, and Vlissides—have been collectively nicknamed the **Gang of Four**.
- These authors wrote their book based on their own experiences as developers. When each had developed programs and graphical applications, they had discovered patterns emerging in their design solutions.
- They formalized those patterns into a reference, which became their book.
- The patterns developed by the Gang of Four were organized to be readable, and often named after their purpose.
- The grouping of patterns together forms a catalog, otherwise known as the **Gang of Four's design pattern catalog**.

## Categories of Patterns

- The Gang of Four's pattern catalog contains twenty-three (23) patterns.
- These patterns can be sorted into three different categories:
  - Creational patterns,
  - Structural patterns, and
  - Behavioural patterns
- These categories were used to simply organize and characterize the patterns in their book.

## Creational Patterns

- Creational patterns deal with the **creation or cloning new objects**.
- Cloning an object occurs when you are creating an object that is similar to an existing one, and instead of instantiating a new object, you clone existing objects instead of instantiating them.
- The different ways of creating objects will greatly influence how a problem is solved. Different languages therefore impact what patterns are possible to use.

## Structural Patterns

- Structural patterns describe **how objects are connected to each other**. These patterns relate to the design principles of **decomposition** and **generalization**.
- Structural patterns use these relationships and describe how they should work to achieve a particular design goal. Each structural pattern **determines the various suitable relationships among the objects**.
- A **good metaphor** for considering structural patterns is that of pairing different kinds of foods together: flavor determines what ingredients can be mixed together to form a suitable relationship.

## Behavioral Patterns

- Behavioral patterns focus on **how objects distribute work**, and describe **how each object does a single cohesive function**.
- Behavioral patterns also focus on **how independent objects work towards a common goal**.
- A **good metaphor** for considering behavioural patterns is that of a race car pit crew at a track. Every member of the crew has a role, but together they work as a team to achieve a common goal. Similarly, a behavioural pattern lays out the overall goal and purpose for each object.

## Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

## Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory Builder Factory Method Prototype <b>Singleton</b>	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

## Singleton Pattern: Intent

- Ensure a class only has one instance, and provide a global point of access to it.

## Singleton Pattern: Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

## Singleton Pattern

- A singleton is a **creational pattern**, which describes a way to create an object. It is a powerful technique, but also one of the simplest examples of a design pattern.
- In a singleton design pattern **only has one object of a class**.
- Another **goal** of singleton design pattern is that **the single object is globally accessible within the program**.

## Singleton Pattern (contd.)

- In order to implement a singleton design pattern, best practice is to build the **one and only one** goal into the class itself, so that creating another instance of a Singleton class is not even possible.
- Let us examine this in code.
- If a class has a `public` constructor, an object of this class can be instantiated at any time.

```
//Class NotSingleton

public class NotSingleton
{
    //Public Constructor
    public notSingleton()
    {

    }
}
```

## Singleton Pattern (contd.)

- Instead, if the class has a private constructor, then the constructor cannot be called from outside the class. Although this seems to prevent creating an object of the class, there are two key components for getting around this.
  - First, **declare a class variable**. In this case, it is called `uniqueInstance`. This class variable will refer to the one instance of your Singleton class. As the variable is private, it can only be modified within the class.
  - Second, **create a `public` method in the class** that will create an instance of this class, but only if an instance does not exist already. In this case, the method is called `getInstance`. It will check if the `uniqueInstance` variable is `null`.
    - If it is `null`, then it will instantiate the class and set this variable to reference the object.
    - On the other hand, if the `uniqueInstance` class variable currently references an object, meaning that there is already an object of this class, then the method will simply return that object.
    - As the `getInstance` method is `public`, it can be called globally and used to create one instance of the class. In a sense, it replaces the normal constructor.

## Singleton Pattern (contd.)

```
public class ExampleSingleton
{
    // lazy construction
    // the class variable is null if no instance is instantiated
    private static ExampleSingleton uniqueInstance = null;

    private ExampleSingleton()
    {
    }

    // lazy construction of the instance
    public static ExampleSingleton getInstance()
    {
        if (uniqueInstance == null)
        {
            uniqueInstance = new ExampleSingleton();
        }
        return uniqueInstance;
    }
}
```

## Singleton Pattern (contd.)

- In the example, **the regular constructor is hidden.**
- Other classes are forced to call the public `getInstance` method. This puts in place basic gatekeeping, and **ensures that only one object of this class is created.**
- The same method can be used to globally reference the single object if it has already been created.

## Singleton Pattern (contd.)

- An **advantage** of this version of a Singleton class is **lazy creation**.
  - Lazy creation means that the object is not created until it is truly needed. This is helpful, especially if the object is large. As the object is not created until the `getInstance` method is called, the program is more efficient.
- There are **trade-offs** to the Singleton design principle.
  - If there are multiple computing threads running, there could be issues caused by the threads trying to access the shared single object.

## Singleton Pattern (contd.)

- In real use, there may be **variations of how Singleton is realized**, as design patterns are **defined by purpose and not exact code**.
- The intent of a Singleton pattern is **to provide global access to a class that is restricted to one instance**.
- In general, **this is achieved by having a private constructor**, with a **public** method that instantiates the class “if” it is not already instantiated.

## It's Quiz Time

- ① ..... patterns describe how objects are connected to each other. These patterns relate to the design principles of decomposition and generalization.
- ① Creational
  - ② Structural
  - ③ Behavioral
- ② A goal of singleton design pattern is that the single object is globally accessible within the program. (True or False)
- ③ If a class has a **public** constructor, an object of this class cannot be instantiated at any time. (True or False)