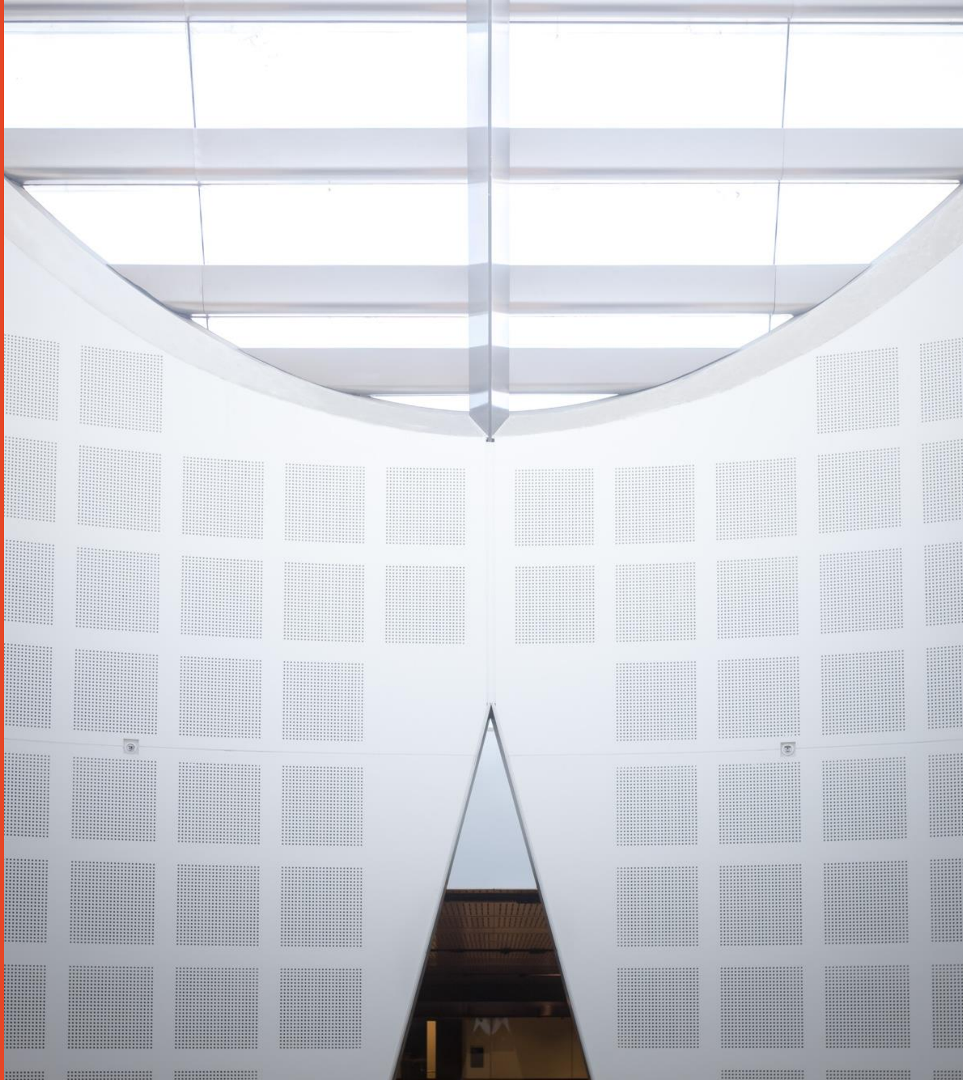
Software Design and Construction 2
SOFT3202 / COMP9202
Introduction
Software Testing

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright Warning

Agenda

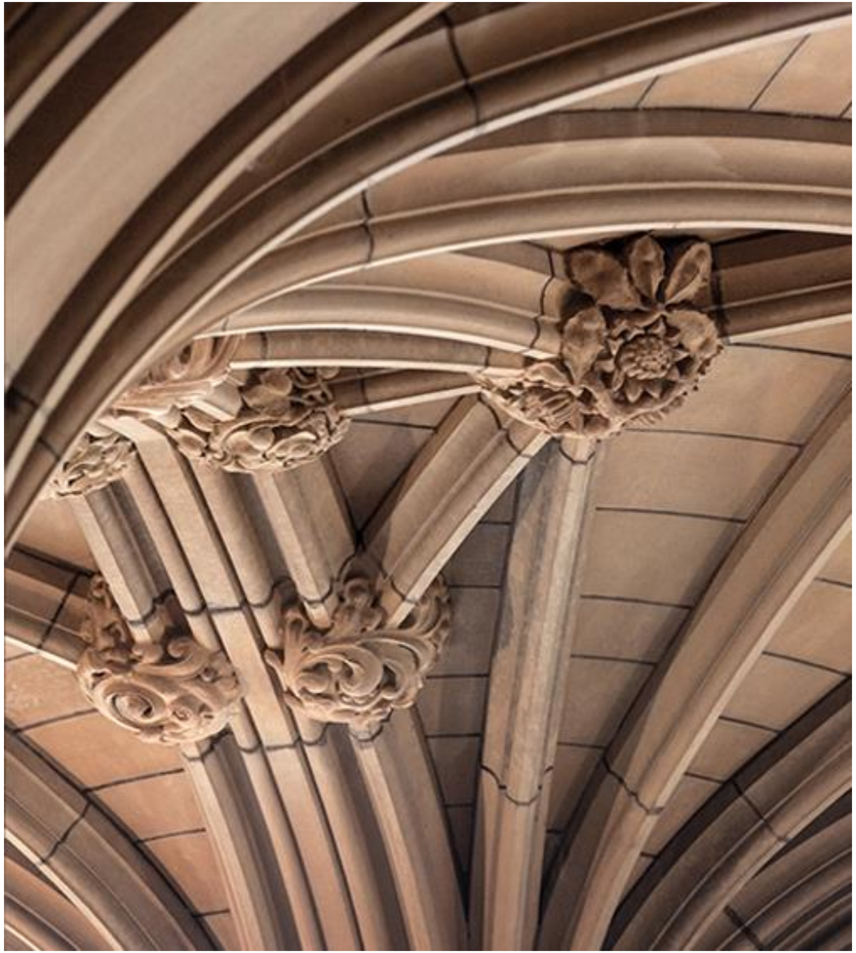
- OO Principles
- Design Principles
- Overview of Design Patterns
- GoF Design Patterns (review)

OO Principles

Review

Slides from SOFT2201 by Bernhard Scholz

THE UNIVERSITY OF SYDNEY



OO Principles

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

Abstract Classes

- Abstract Classes whose method implementations are deferred to sub-classes
- Important concept in OO
- Requires own key-word abstract
- No instance of an abstract class can be generated

Interfaces

- Java has no multi-inheritance
  - Interface is a way-out (introduction of multi-inheritance via the back-door)
- Interfaces is a class contract that ensures that a class implements a set of methods.
- Interfaces can inherit from other interfaces
- Ensures that a class has a certain set of behavior
- Interfaces are specified so that they form a directed acyclic graph
- Methods declared in an interface are always public and abstract
- Variables are permitted if they are static and final only

Example: Interface

```
// definition of interface
public interface A {
  int foo(int x);
}

// class X implements interface A
class X implements A {
  int foo(int x) {
    return x;
  }
}
```

Example: Interface

– Inheritance in interfaces

```
// definition of interface
public interface A {
  int foo(int x);
}

public interface B extends A{
  int hoo(int x);
}
```

– Interface B has methods foo() and hoo()

Virtual Dispatch

- Methods in Java permit a late binding
- Reference variable and its type does not tell which method is really invoked
- The type of reference variable and class instance may differ
- Class variables may override methods of super classes
- The method invoked is determined by the type of the class instance
- Binding is of great importance to understand OO

Example: Virtual Dispatch

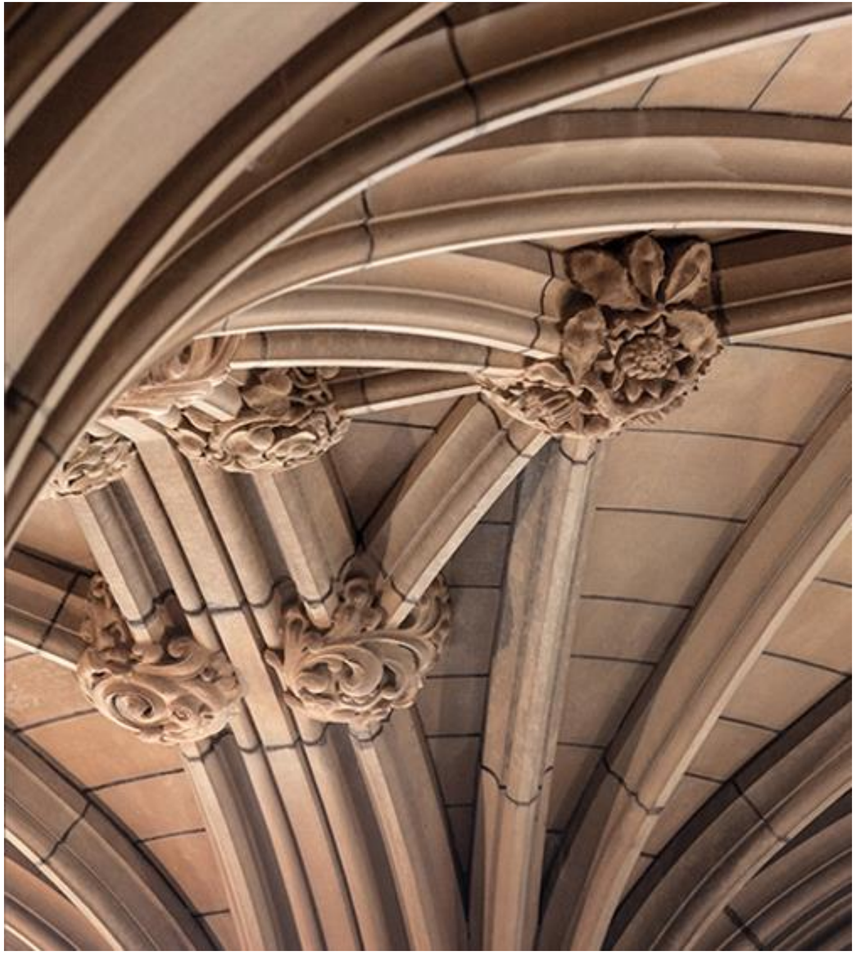
- Example:

```
public class Shape extends Object {
  double area() { }
}

public class Rectangle extends Shape {
  double area() { }
}
…
Shape X = new Shape();
Shape Y = new Rectangle();

double a1 = X.area() // invokes area of Shape
double a2 = Y.area() // invokes area of Rectangle
```

OO Design Principles

Revisit

.



GRASP: Methodological Approach to OO Design

General **R**esponsibility **A**ssignment **S**oftware **P**atterns

The five basic principles:

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller

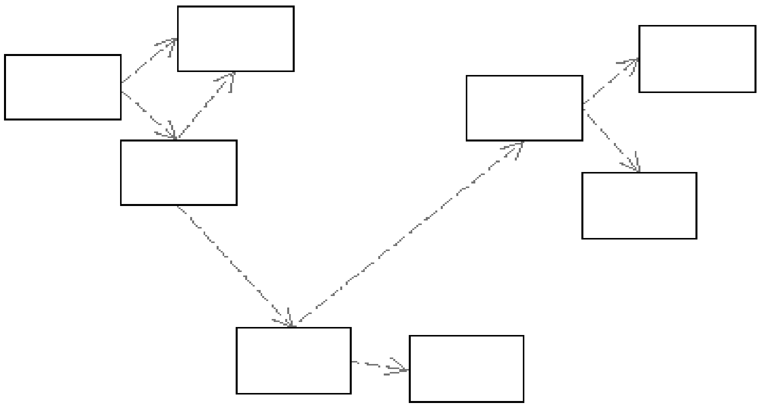
Dependency

- A dependency exists between two elements if changes to the definition of one element (the **supplier**) may cause changes to the other (the **client**)

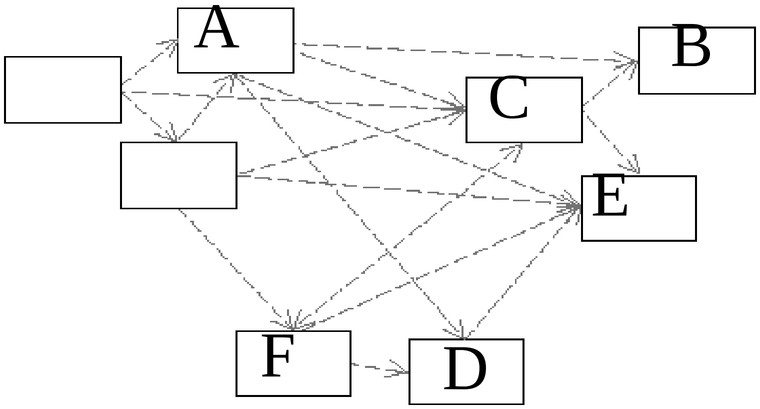
- Various reason for dependency
  - Class send message to another
  - One class has another as its data
  - One class mention another as a parameter to an operation
  - One class is a superclass or interface of another

Coupling

- How strongly one element is connected to, has knowledge of, or depends on other elements
- Illustrated as dependency relationship in UML class diagram



Low coupling



High coupling

GRASP: Low Coupling Principle

Problem

How to reduce the impact of change, to support low dependency, and increase reuse?

Solution

Assign a responsibility so that coupling remains low

Cohesion

- How strongly related and focused the responsibilities of an element are

- How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - Assign responsibilities so that cohesion remains high

OO Design Principles

- Separate aspects of your application that vary from what does not change

- Program to an interface not an implementation

- Behavior delegation

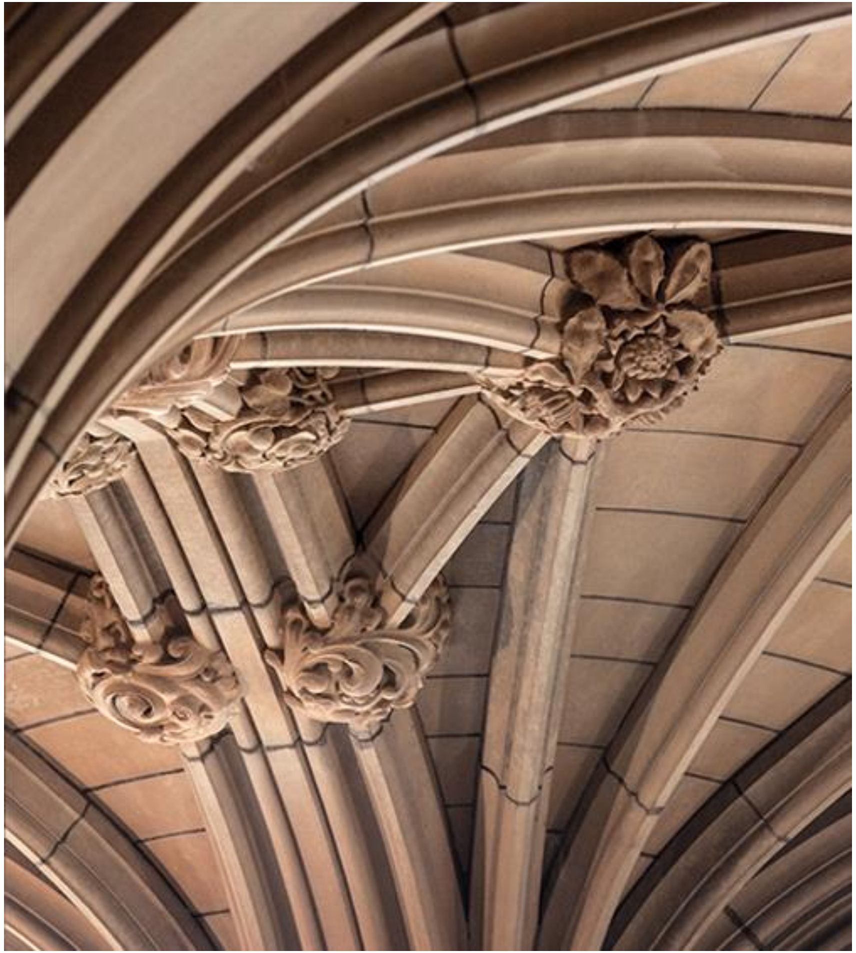
- Composition vs Inheritance
  - Composition provides a lot of flexibility and change the behavior at runtime (the object you're composing with implements the correct behavior interface)

- Quality of OO designs are evaluated based on reusability, extensibility, and maintainability

Design Patterns Review

Revisit

Slides from SOFT2201



Catalogue of Design Patterns

Design Pattern	Description
Gang of Four (Gof)	First and most used. Fundamental patterns OO development, but not specifically for enterprise software development.
Enterprise Application Architecture (EAA)	Layered application architecture with focus on domain logic, web, database interaction and concurrency and distribution. Database patterns (object-relational mapping issues)
Enterprise Integration	Integrating enterprise applications using effective messaging models
Core J2EE	EAA Focused on J2EE platform. Applicable to other platforms
Microsoft Enterprise Solution	MS enterprise software patterns. Web, deployment and distributed systems

https://martinfowler.com/articles/enterprisePatterns.html

Catalogue of Design Patterns

Design Pattern	Description
Microsoft Integration	Microsoft view on integration layer, system connections and topologies for integration
Data Model	Common patterns for data modelling and useful for object modelling
Microsoft Data	Patterns on data movement; replication and synchronization

https://martinfowler.com/articles/enterprisePatterns.html

Aspects of Enterprise Software

- Enterprise Application Architecture
  - EAA typical structured into logical layers
  - Some differences but some common aspects
  - Technology independent/dependent
- Patterns of EAA
  - Technology independent
- Core J2EE
  - J2EE context
- Microsoft Enterprise Solution Patterns
  - .NET views

Aspects of Enterprise Software

- Enterprise Integration
  - EA developed independently but needs to work together
  - Integration was not considered, or dependent on certain technology

- Enterprise Integration Patterns
  - messaging
- Microsoft Integration Patterns
  - strategies
- Microsoft Data Patterns
  - replication and synchronization

Aspects of Enterprise Software

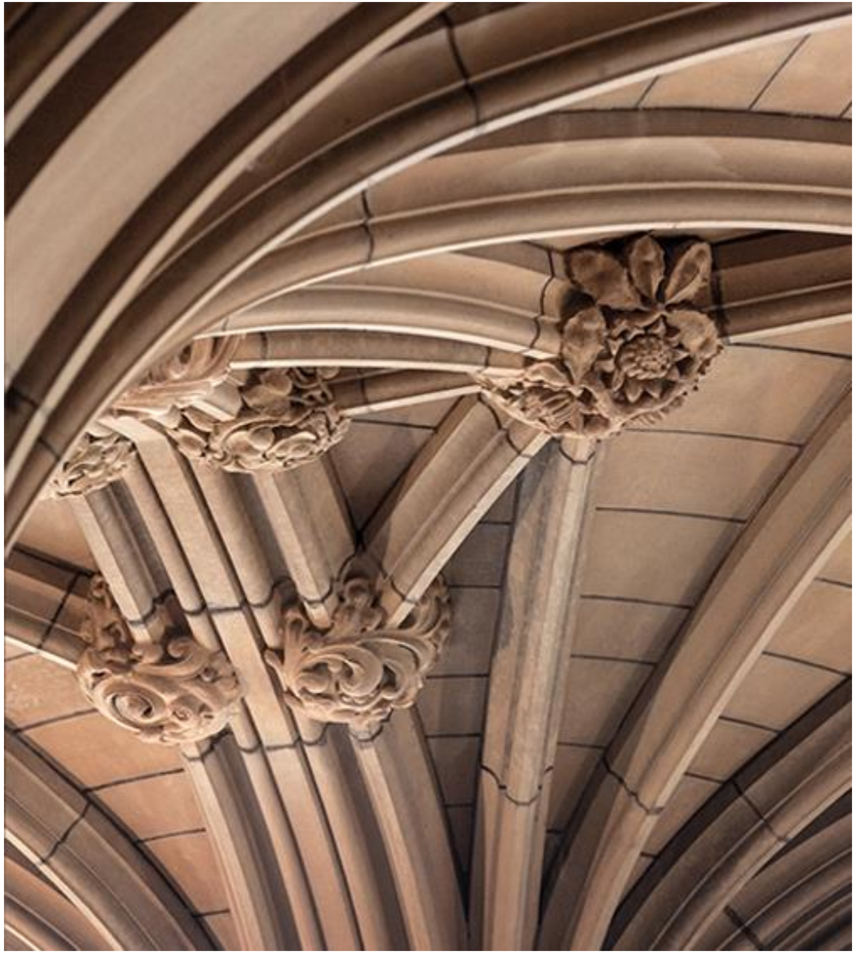
- Domain Logic
  - Business rules, validations and computations
  - Some systems intrinsically have complex domain logic
  - Regular changes as business conditions change

- EAA Patterns
  - organizing domain logic
- Data Model Patterns
  - Data modeling approach with examples of domains

SOFT3202 / COMP93202

- GoF Patterns
  - Flyweight, Bridge, Chain of Responsibility
- Concurrency
  - Lock, Thread Pool
- Enterprise
  - Lazy load, Value Object, Unit of Work (,MVC, SOA)

Review of GoF Design Patterns



Design Patterns

- Proven solutions to general design problems which can be applied to specific applications

- Not readily coded solution, but rather the solution path to a common programming problem

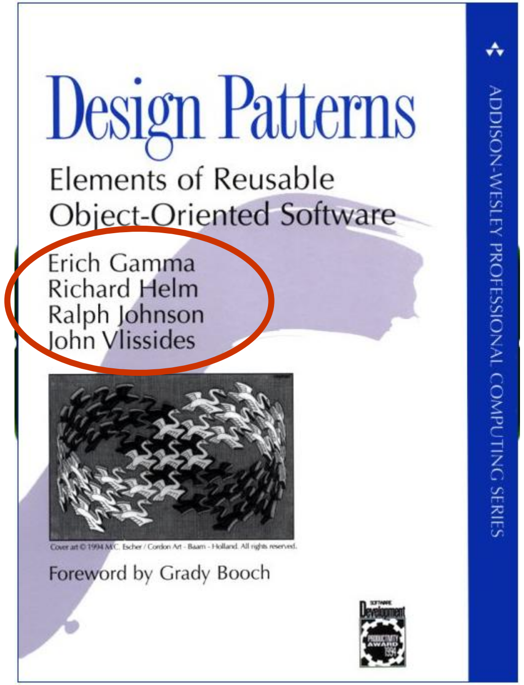
- Design or implementation **structure** that achieves a particular purpose
- Allow evolution and change
  - Vary independently of other components

- Provide a shared language among development communities – effective communication

Elements of a Pattern

- The **pattern name**
- The **problem**
  - When to apply the pattern
- The **solution**
  - The elements that make up the design
- **Consequence**
  - The results and trade-offs of applying the pattern

Gang of Four Patterns (GoF)


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Official design pattern reference
- Famous and influential book about design patterns

Recommended for students who wish to become experts

We will cover the most widely - used patterns from the book

- Many other patterns but not all so popular
- GoF Design Patterns → Design Patterns

Design Patterns – Classification

Scope / Purpose	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Design Patterns – Classification

Describes of 23 design patterns

- **Creational patterns**
  - Abstract the instantiation process
  - Make a system independent of how its objects are created, composed and represented
- **Structural patterns**
  - How classes and objects are composed to form larger structures
- **Behavioral patterns**
  - Concerns with algorithms and the assignment of responsibilities between objects

Selecting Appropriate Design Pattern

- Consider how design pattern solve a problem
- Read through Each pattern's intent to find relevant ones
- Study the relationship between design patterns
- Study patterns of like purpose (similarities and differences)
- Examine a cause of redesign (what might force a change to a design? Tight-coupling, difficult to change classes
- Consider what should be variable in your design (see table in next slides)

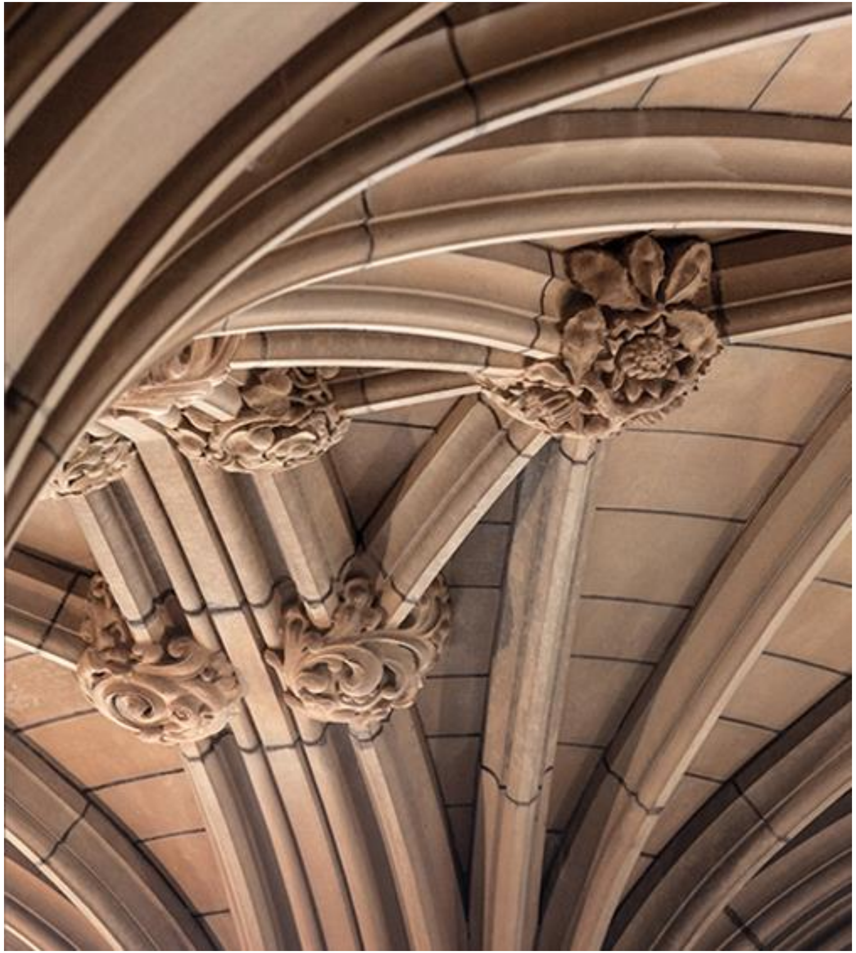
Design Aspects Can be Varied by Design Patterns

Purpose	Pattern	Aspects that can change
Creational	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Façade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location

Design Aspects Can be Varied by Design Patterns

Purpose	Pattern	Aspects that can change
Behavioral	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private info. is stored outside an object, & when
	Observer	number of objects that depend on another object; how the dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to object(s) without changing their class(es)

Creational Patterns



Creational Patterns

- Abstract the instantiation process
- Make a system independent of how its objects are created, composed and represented
  - Class creational pattern uses inheritance to vary the class that's instantiated
  - Object creational pattern delegates instantiation to another object
- Becomes more important as systems evolve to depend on object composition than class inheritance
- Provides flexibility in *what gets created*, *who creates it*, *how it gets created* and *when*
  - Let you configure a system with “product” objects that vary in structure and functionality

Creational Patterns

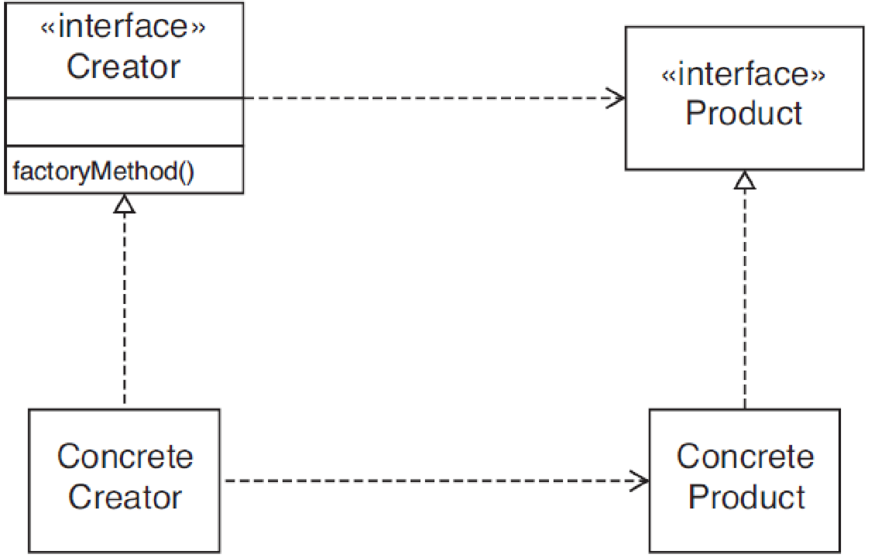
Pattern Name	Description
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes
Singleton	Ensure a class only has one instance, and provide global point of access to it
Factory Method	Define an interface for creating an object, but let sub-class decide which class to instantiate (class instantiation deferred to subclasses)
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Prototype	Specify the kinds of objects to create using a prototype instance, and create new objects by copying this prototype

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

Factory Method

- Intent
  - Define an interface for creating an object, but let *subclasses decide which class to instantiate*. Let a class defer instantiation to subclasses
- Also known as
  - Virtual Constructor
- Applicability
  - A class cannot anticipate the class 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 Pattern – Structure

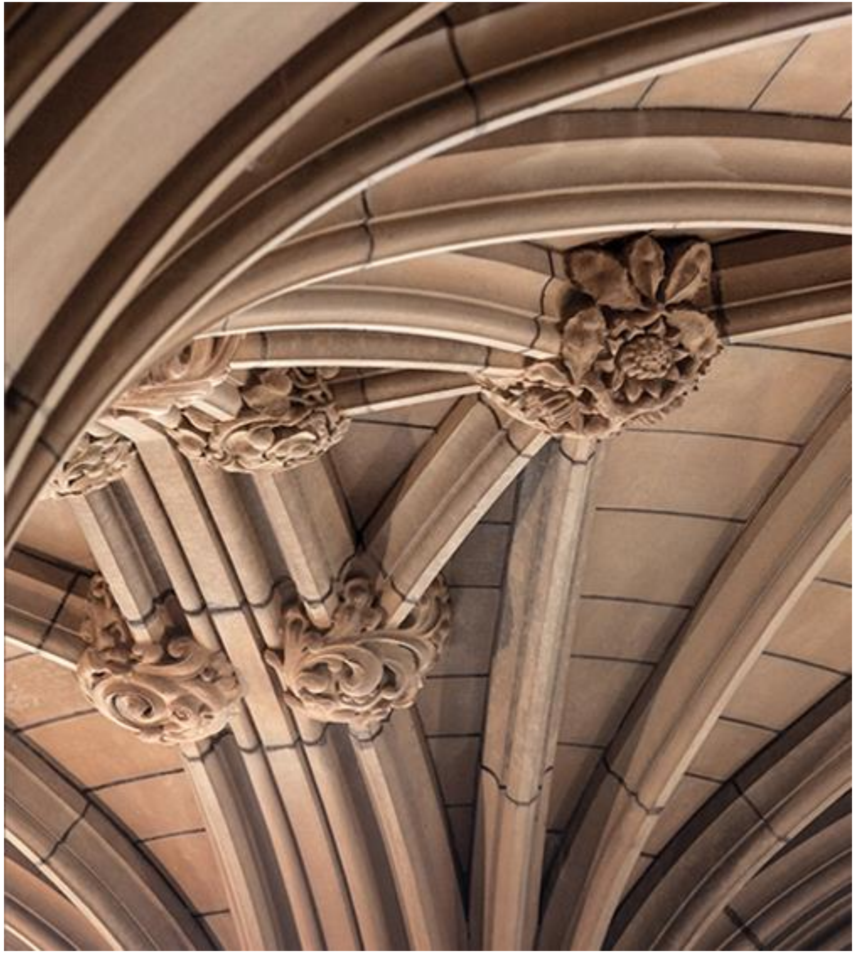


Factory Method Pattern – Participants

- **Product**
  - Defines the interface of objects the factory method creates
- **ConcreteProduct**
  - Implements the Product interface
- **Creator**
  - Declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default *ConcreteProduct* object
  - May call the factory method to create a Product object
- **ConcreteCreator**
  - Overrides the factory method to return an instance of a Concrete Product

Abstract Factory

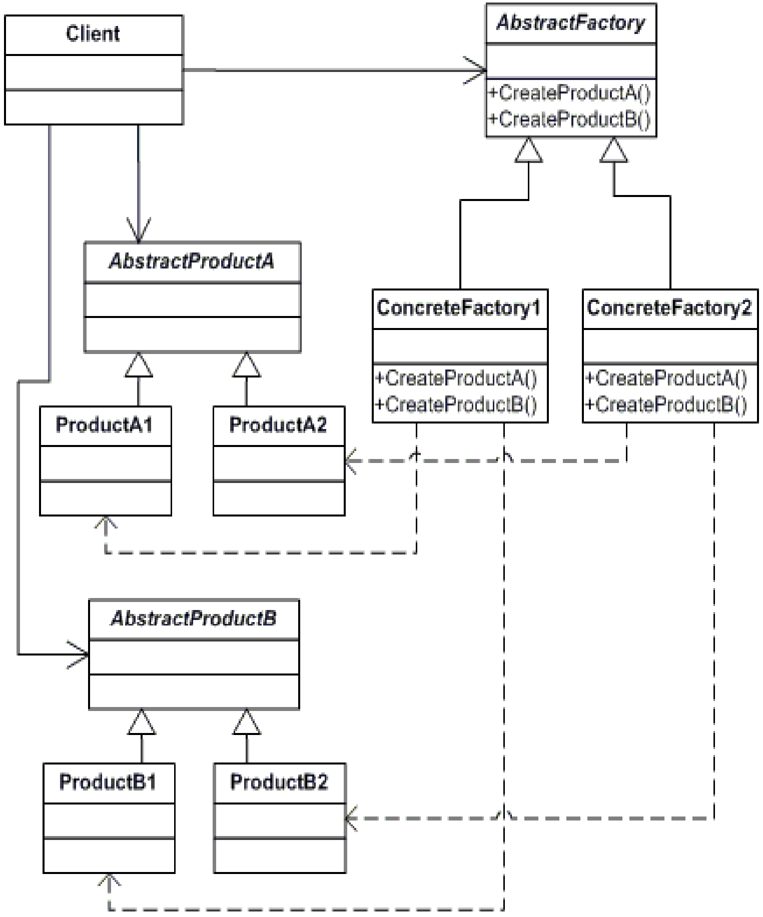
Object Creational



Abstract Factory Pattern

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

Abstract Factory – Structure

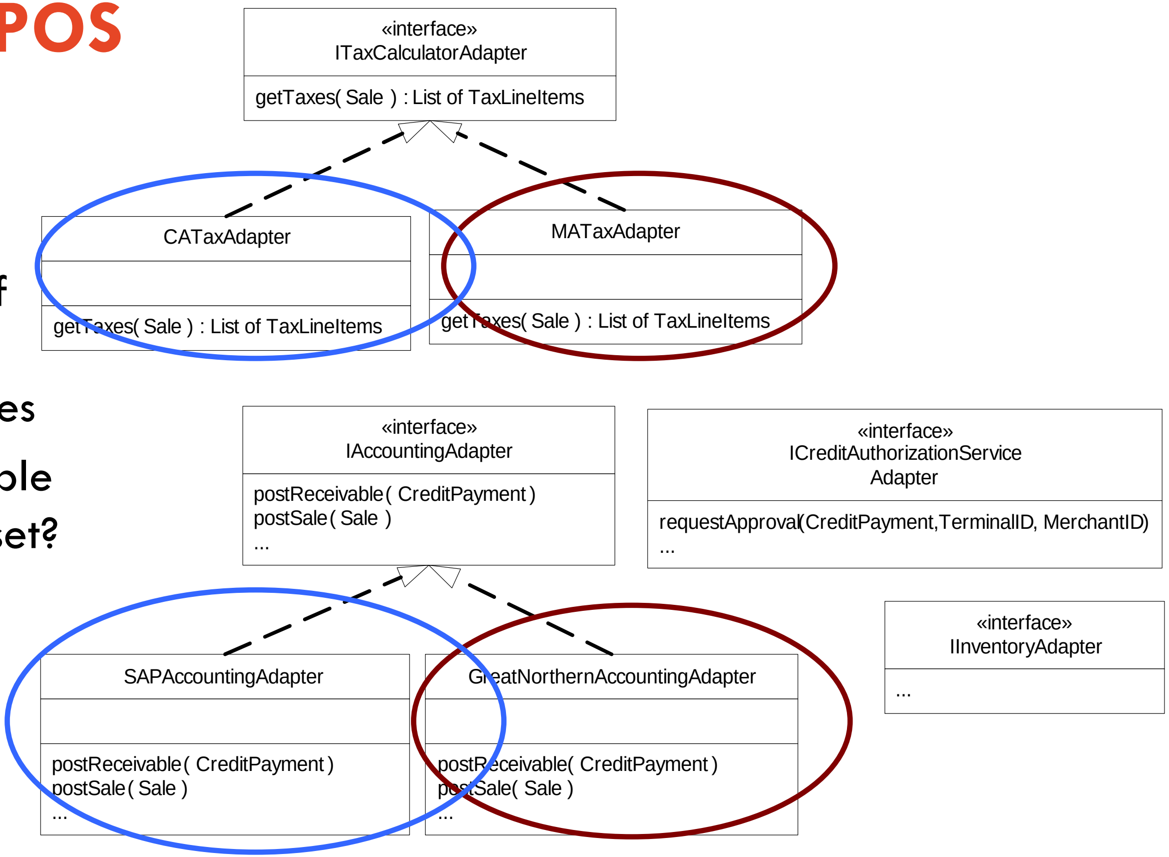


Abstract Factory Pattern – Participants

- **AbstractFactory**
  - Declares an interface for operations that create abstract product objects
- **ConcreteFactory**
  - Implements the operations to create concrete product objects
- **AbstractProduct**
  - declares an interface for a type of product object
- **ConcreteProduct**
  - defines a product object to be created by the corresponding concrete factory
  - Implements the AbstractProduct interface
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Factory – POS

- Problem
  - Now we have a series of adapters for all sorts of different external services
  - Who should be responsible for creating the correct set?



«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
CATaxAdapter
getTaxes(Sale) : List of TaxLineItems
MATaxAdapter
getTaxes(Sale) : List of TaxLineItems
«interface»
IAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
«interface»
ICreditAuthorizationService Adapter
requestApproval(CreditPayment,TerminalID, MerchantID)
...
SAPAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
GreatNorthernAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
«interface»
IInventoryAdapter
...

Abstract Factory – POS

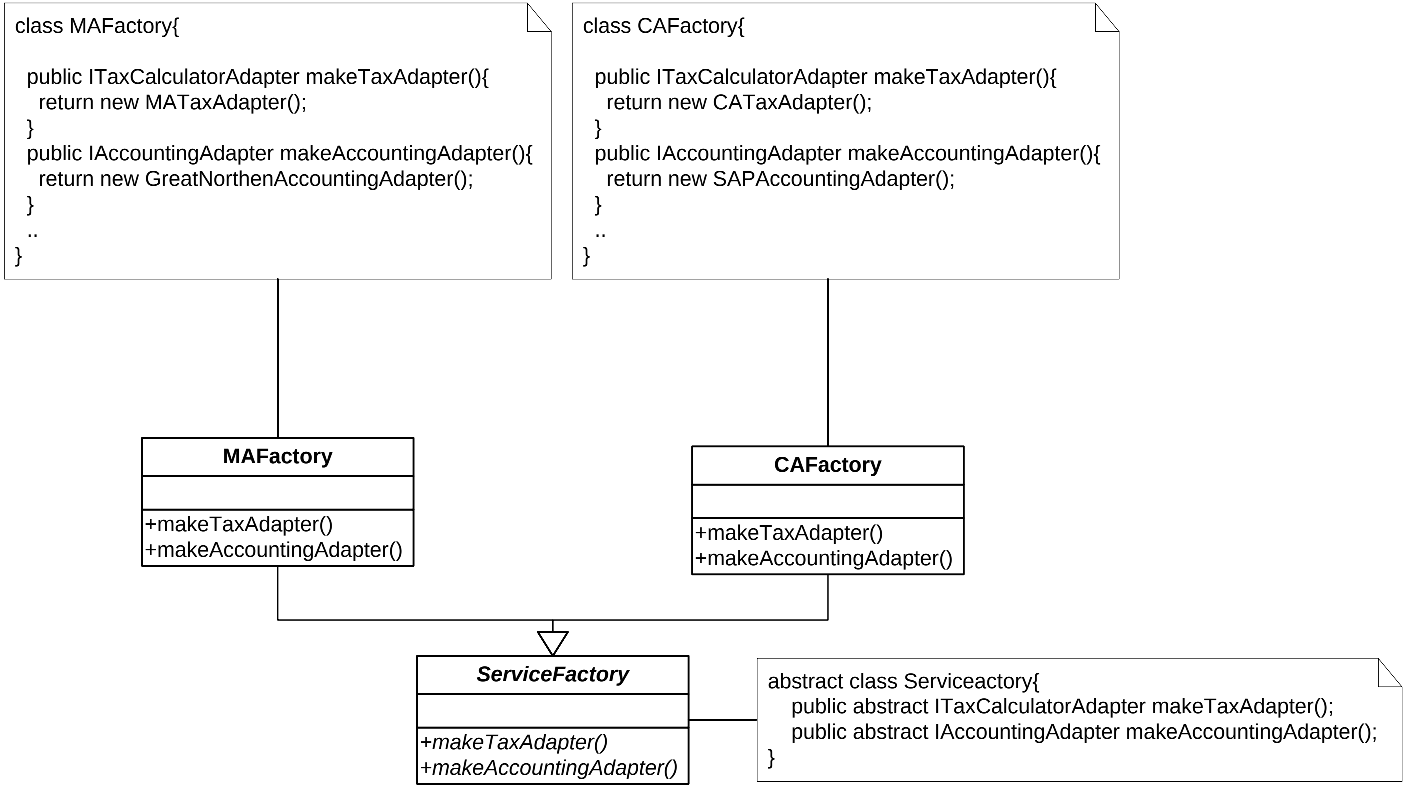
- Suppose the POS is deployed in some stores in MA, we'll need
  - `MATaxAdapter, GreatNorthenAccountingAdapter`, …

- If it is deployed in CA, we'll need
  - `CATaxAdapter, SAPAccountingAdapter`

Abstract Factory – POS

- We need several factory objects each will be responsible for creating a set of objects
  - A **`MAFactory`** which will create **`MATaxAdapter`**, **`GreatNorthemAccountingAdapter`** and so on
  - A **`CAFactory`** which will create **`CATaxAdapter`**, **`SAPAccountingAdapter`** and so on
  - Naturally we'll have an abstraction which is an Abstract Factory

Abstract Factory – POS



```
class MAFactory{

  public ITaxCalculatorAdapter makeTaxAdapter(){
    return new MATaxAdapter();
  }
  public IAccountingAdapter makeAccountingAdapter(){
    return new GreatNorthenAccountingAdapter();
  }
  ..
}
```

```
class CAFactory{

  public ITaxCalculatorAdapter makeTaxAdapter(){
    return new CATaxAdapter();
  }
  public IAccountingAdapter makeAccountingAdapter(){
    return new SAPAccountingAdapter();
  }
  ..
}
```

MAFactory

+makeTaxAdapter()
+makeAccountingAdapter()

CAFactory

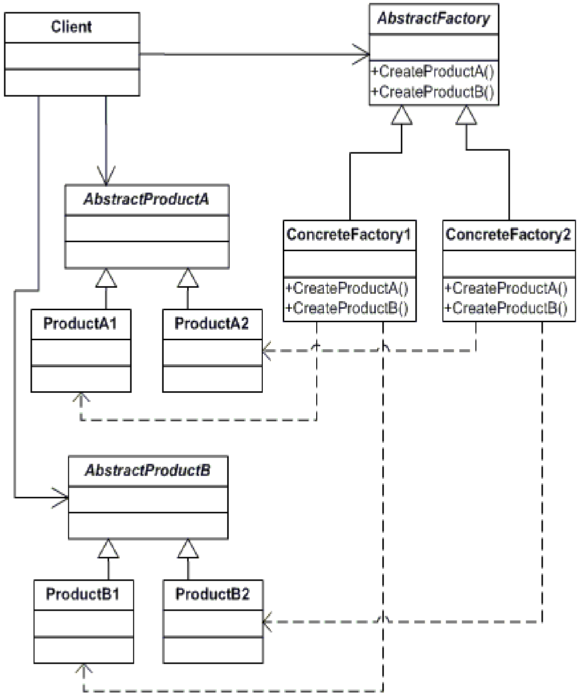
+makeTaxAdapter()
+makeAccountingAdapter()

ServiceFactory

+makeTaxAdapter()
+makeAccountingAdapter()

```
abstract class Serviceactory{
    public abstract ITaxCalculatorAdapter makeTaxAdapter();
    public abstract IAccountingAdapter makeAccountingAdapter();
}
```

Abstract Factory – Structure (NextGen POS



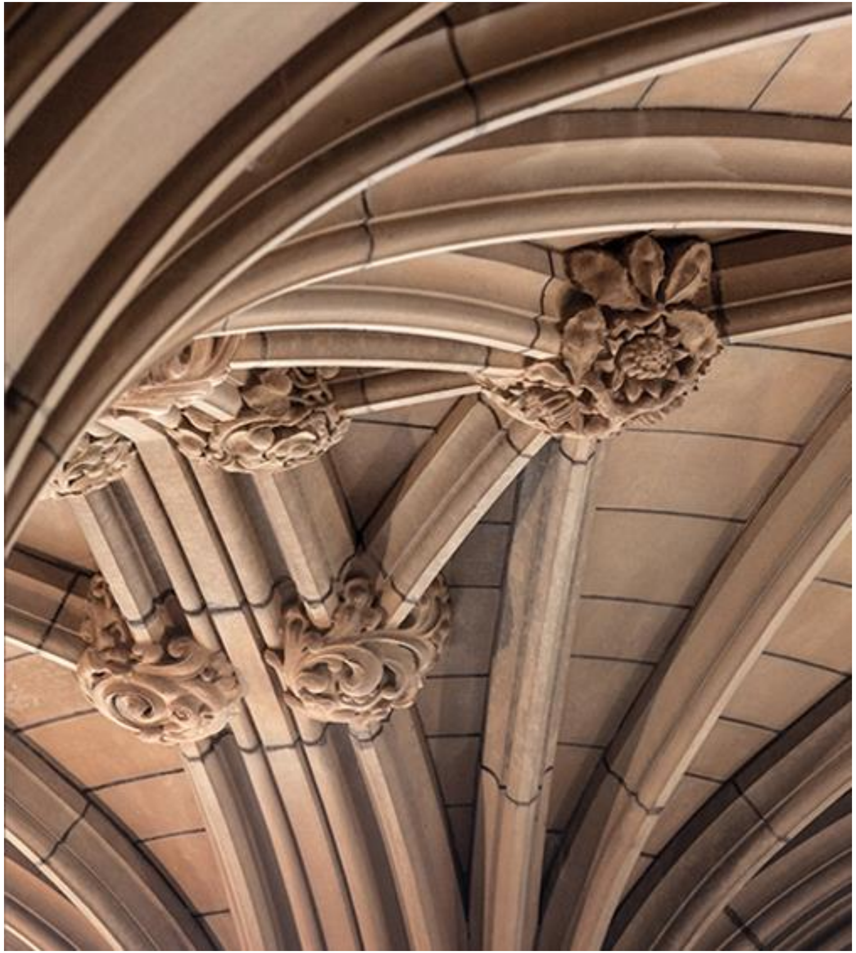
- **AbstractFactory** (**`ServiceFactory`**)
  - declares an interface for operations that create abstract products
- **ConcreteFactory** (**`MAFactory`**, **`CAFactory`**)
  - implements the operations to create concrete product objects
- **AbstractProduct** (**`IAccountingAdapter`**, **`ITaxCalculatorAdpter`**)
  - declares an interface for a type of product object
- **Product** (**`MATaxAdapter`**, **`CATaxAdapter`**, **`SAPAccountingAdapter`**, **`GreatNorthernAccountingAdapter`**)
  - defines a product object to be created by the corresponding concrete factory
  - implements the AbstractProduct interface
- **Client** (**`Store`**)
  - uses interfaces declared by AbstractFactory and AbstractProduct classes

Other Creational Patterns

Pattern Name	Description
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes
Singleton	Ensure a class only has one instance, and provide global point of access to it
Factory Method	Define an interface for creating an object, but let sub-class decide which class to instantiate (class instantiation deferred to subclasses)
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Prototype	Specify the kinds of objects to create using a prototype instance, and create new objects by copying this prototype

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

Structural Patterns



Structural Patterns

- How classes and objects are composed to form larger structures
- Structural *class* patterns use inheritance to compose interfaces or implementations
- Structural *object* patterns describe ways to compose objects to realize new functionality
  - The flexibility of object composition comes from the ability to change the composition at run-time

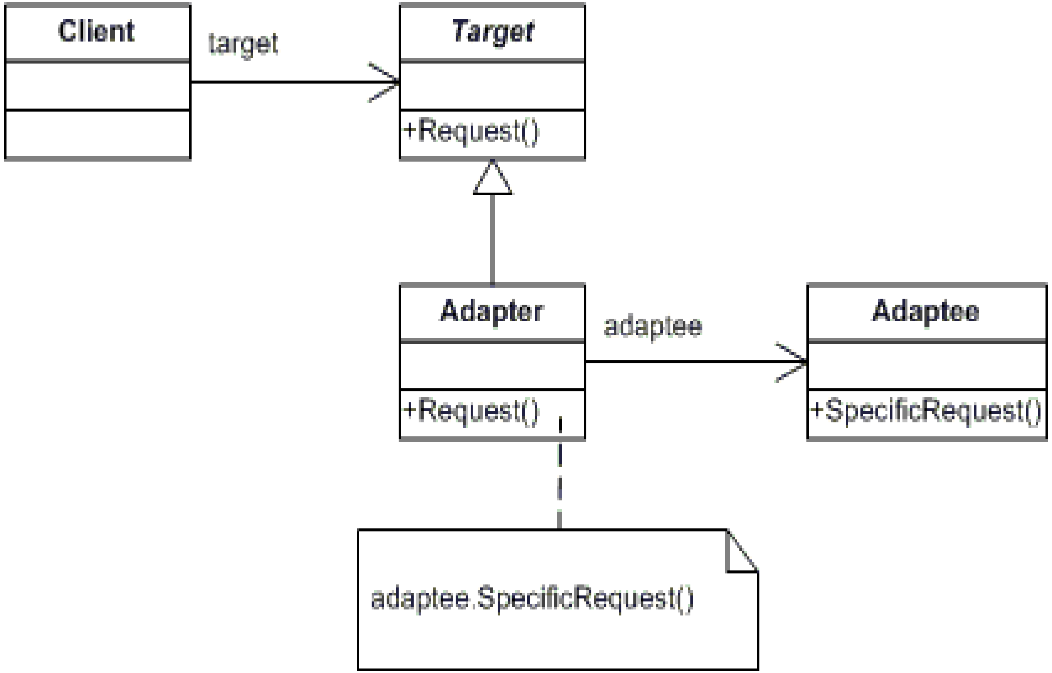
Structural Patterns (GoF)

Pattern Name	Description
Adapter	**Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface clients expect.**
Façade	**Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.**
Composite	Compose objects into tree structures to represents part-whole hierarchies. Let client treat individual objects and compositions of objects uniformly
Proxy	Provide a placeholder for another object to control access to it
Decorator	**Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)**
Bridge	Decouple an abstraction from its implementation so that the two can vary independently
Flight weight	Use sharing to support large numbers of fine-grained objects efficiently

Adapter

- Intent
  - Convert the interface of a class into another interface clients expect
  - Lets classes work together that couldn't otherwise because of incompatible interfaces
- Applicability
  - To use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, i.e., classes that don't necessarily have compatible interfaces
  - **Object adapter only** to use several existing subclasses, but it's unpractical to adapt their interface by sub-classing every one. An object adapter can adapt the interface of its parent class.

Object Adapter – Structure



Adapter – Participants

- **Target**
  - Defines the domain-specific interface that Client uses
- **Client**
  - Collaborates with objects conforming to the Target interface.
- **Adaptee**
  - Defines an existing interface t hat needs adapting.
- **Adapter**
  - Adapts the interface of Adaptee to the Target interface
- **Collaborations**
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptec operations that carry out the request

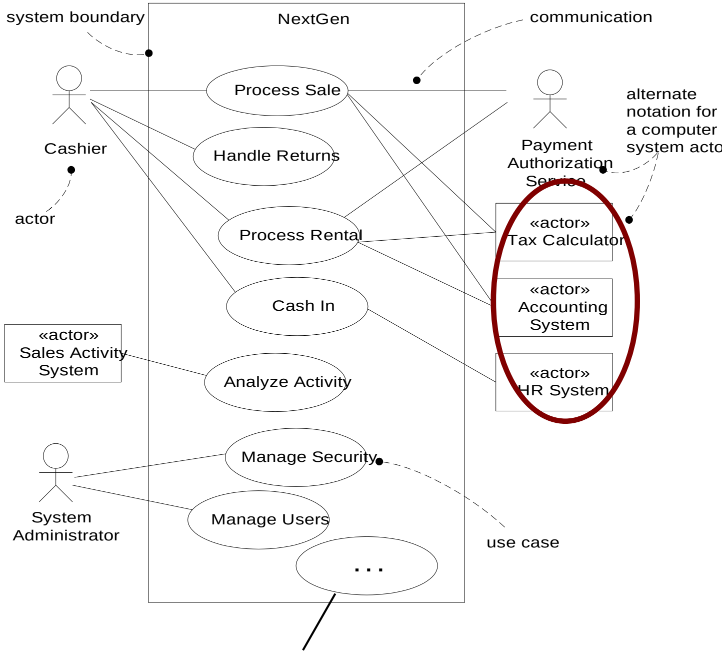
Object Adapter – Consequences

- Lets a single Adapter work with many Adaptees – i.e., the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once

- Makes it harder to override Adaptee behavior. It will require sub-classing adaptee and making Adapter refer to the subclass rather than the Adaptee itself

Adapter in POS – Requirements

- Next Gen PoS system needs to communicate with several external third-party services
  - Tax calculators, credit authorization services, inventory systems, accounting systems.
  - Each has a different API and can not be changed.



Outside the system boundary and developed and maintained by third parties

Adapter in POS – Reality

- Consider the *TaxCalculator* services

- Suppose the POS system will be installed through out the country
  - Each state has its own way of calculating and collecting tax
    - California: 8.5% on almost everything
    - Mass: 5% on most items except grocery

  - Each state has its own TaxCalculator service (as a jar perhaps)
    - California API: **`List getTaxes (List allItem)`**
    - Mass API: **`Set computeTaxes (Set allItem)`**

Requirements – Business Rules

- Business rule (domain rule)
  - Dictate how a domain or business may operate
  - Not restricted by a particular application
  - May apply to many applications in the same domain
  - Company policies, physical laws and government laws
- Example:

ID	Rule	Changeability	Source
RULE1	Tax rules. Sales require added taxes. (POS domain)	High. Tax laws change annually, at all government levels	law

Adapter in POS – First Attempt

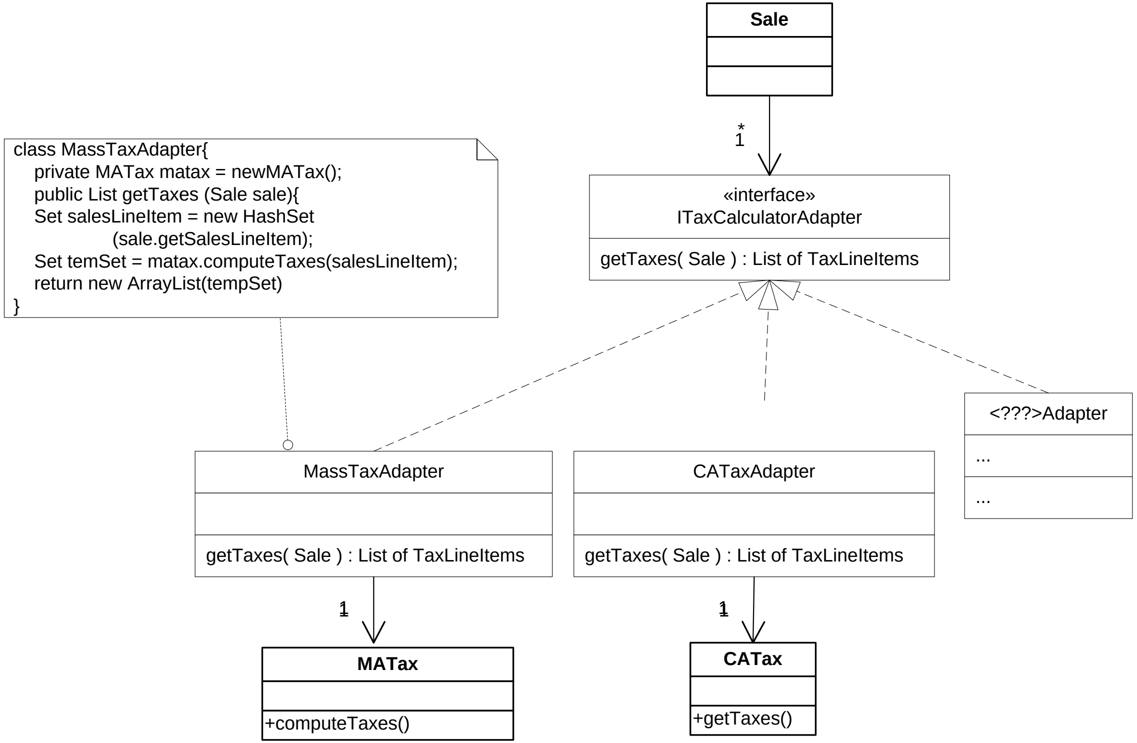
- Novice (straightforward) design of `Sale.getTaxes` method

```
List getTaxes (){
    …
    switch (state){
     case CA:
         CATax catax = new CATax();
         List taxlineitem = catax.getTaxes (lineItems);
         break;
     case MA:
         MATax matax = new MATax();
         List taxlineitem = new ArrayList(matax.computeTaxes (
          new HashSet(lineItems))); // type conversion required
         break;
     …
    }
```

Adapter in POS – Solution

- How to make the `Sale` object decoupled from detailed Tax Calculation services
  - Reason: `Sale` is interested in getting the tax for each taxable item, not how taxes are calculated in different states
  - Solution:
    - Sale defines an interface to get results on taxes
      - `ITaxCalculatorAdapter`
    - Any Tax Calculation Service that does not use this interface needs to find a interpreter (Adapter) to do the translation
      - `MassTaxAdapter, CATaxAdapter`

Adapter in POS – Solution



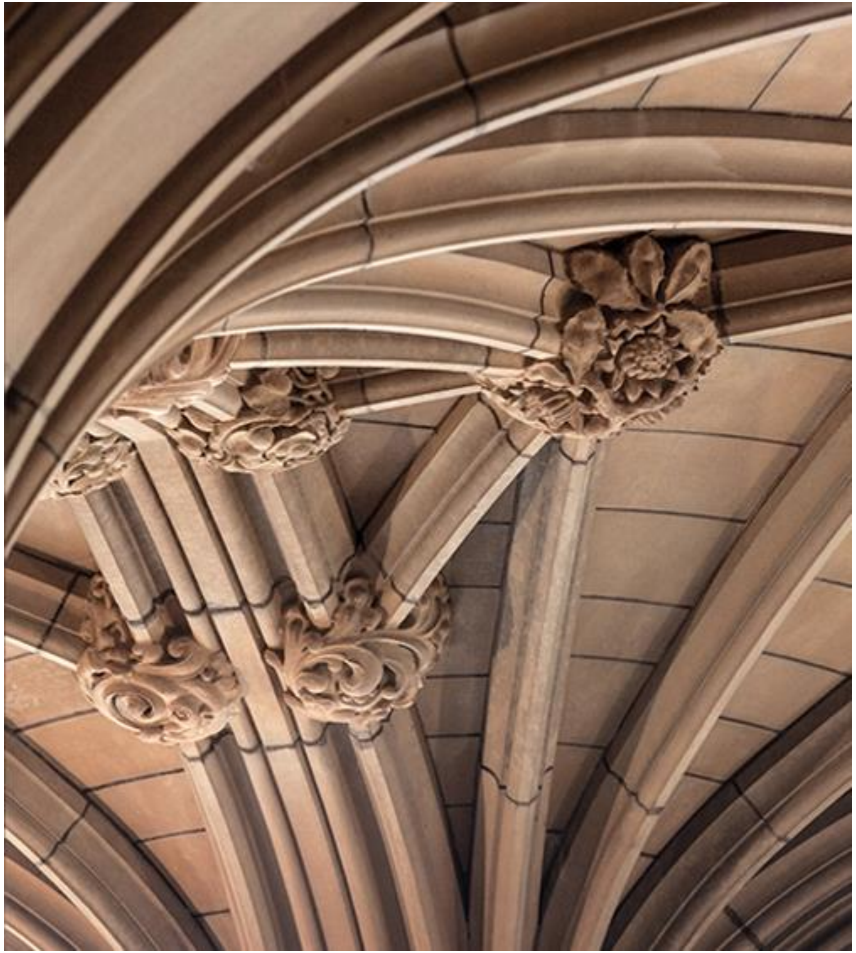
```
class MassTaxAdapter{
    private MATax matax = newMATax();
    public List getTaxes (Sale sale){
    Set salesLineItem = new HashSet
                (sale.getSalesLineItem);
    Set temSet = matax.computeTaxes(salesLineItem);
    return new ArrayList(tempSet)
}
```

The new Sale class

```
class Sale{
   List taxLineItem;
   …
   List<TaxLineItem> getTaxes (){
    ITaxCalculatorAdapter tca = …;
    return tca.getTaxes(this);
   }
   …
}
```

Façade Pattern

Object Structural

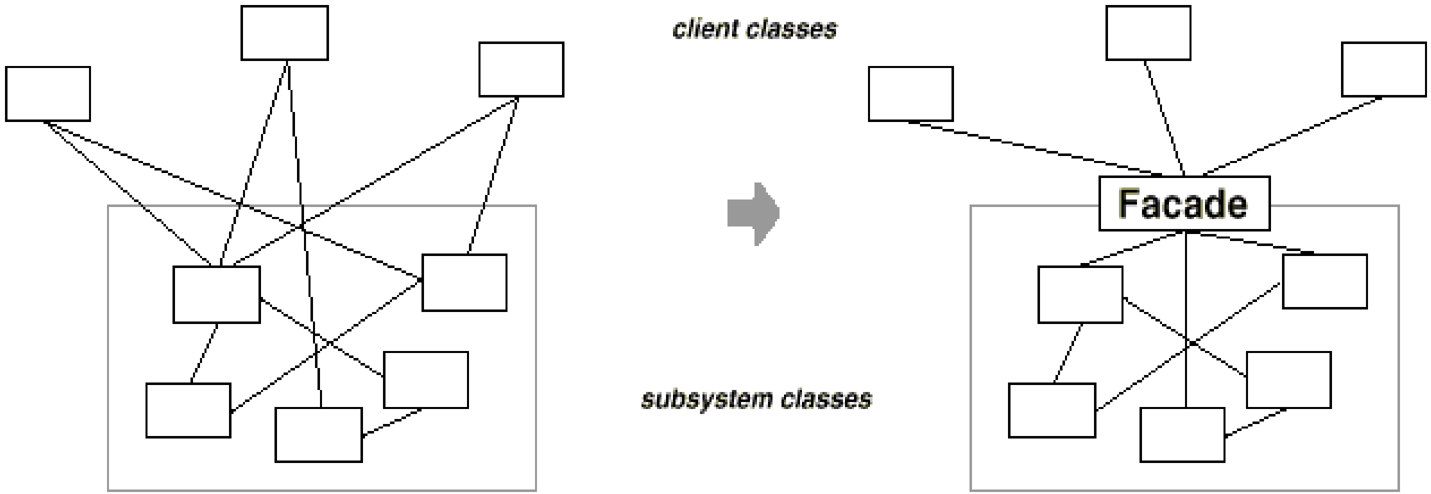


Façade Pattern

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

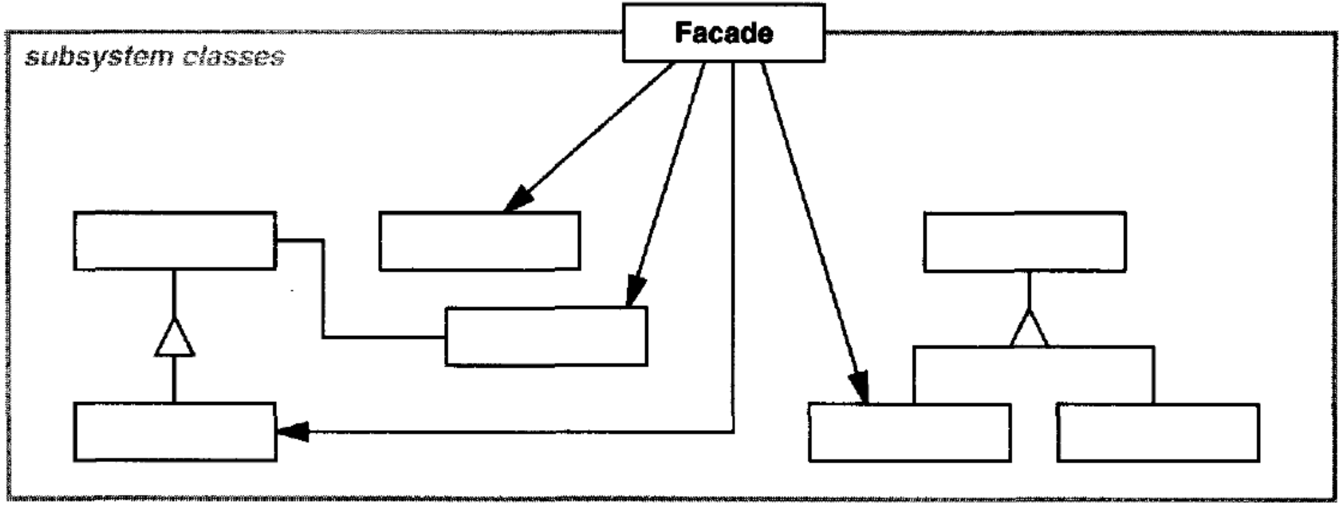
- Applicability
  - You want to provide a simple interface to a complex subsystem
  - There are many dependencies between clients and the implementation classes of an abstraction
  - You want to layer your subsystem. Façade would define an entry point to each subsystem level

Façade Motivation



A **facade** object provides a single, simplified interface to the more general facilities of a subsystem

Façade – Structure



Façade – Participants

- **Facade**
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- **Subsystem classes**
  - Implement subsystem functionality.
  - Handle work assigned by the Façade object
  - Have no knowledge of the facade; they keep no references to it.
- **Collaborations**
  - Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s).
    - Although the subsystem objects perform the actual work, the façade may have to do work of its own to translate its interface to subsystem interfaces
  - Clients that use the facade don't have to access its subsystem objects directly

Consequences

- Simplify the usage of an existing subsystem by defining your own interface

- Shields clients from subsystem components, reduce the number of objects that clients deal with and make the subsystem easier to use.

- Promote weak coupling between the subsystem and the clients
  - Vary the components of the subsystem without affecting its clients
  - Reduce compilation dependencies (esp. large systems) – when subsystem classes change

- Does not prevent applications from using subsystem classes if they need to. Choice between ease of use and flexibility.

Façade – NextGen POS

- *Pluggable business rules* in POS (iteration 2 requirements)

- Consider rules that might invalidate an action at certain point
  - When a new sales is created:
    - Business rule 1: if it will be paid by a gift card, only one item is allowed to be purchased. Invalidate all requests of entering another item.

  - When a payment is made by gift certificate:
    - Business rule 2: balance should due back in another gift certificate. Invalidate all requests of giving customer change either in cash or credit card.

Façade - NextGen POS Business Rules

- Each store that would deploy PoS system will have different business rules implemented differently

  - Not desirable to scatter the implementation of business rules all over the system
    - Consequence: for each installation of POS system, need to modify lots of classes

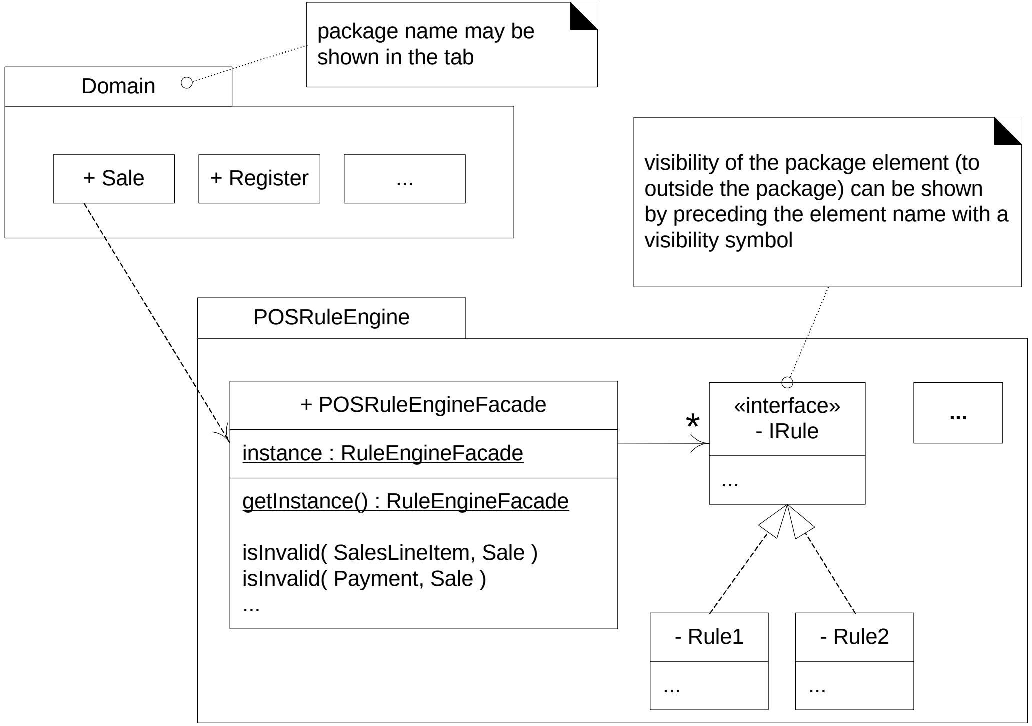
  - We want a design that has low impact on the existing software components.
    - Factor out all rules in a separate subsystem to localize the change.

Façade in POS system

- Use an façade object **POSRuleEngineFacade** to communicate with the business rule subsystem
- Façade object is usually implemented as Singleton

```
public class sale{
   public void makeLineItem(ProductDescription desc, int quantity){
    SalesLineItem sli = new SalesLineItem (desc, quantity);
    // call to the Façade
    if (POSRuleEngineFacade.getInstance().isInvalid(sli, this))
        return;
    lineItems.add(sli);
   }
   //..
}.. End of Sale
```

Façade – NextGen POS


package name may be shown in the tab
Domain
+ Sale
+ Register
...
visibility of the package element (to outside the package) can be shown by preceding the element name with a visibility symbol
POSRuleEngine
+ POSRuleEngineFacade
instance : RuleEngineFacade
getInstance() : RuleEngineFacade
isInvalid(SalesLineItem, Sale)
isInvalid(Payment, Sale)
...
*
«interface»
- IRule
...
...
- Rule1
...
- Rule2
...

Façade – NextGen POS

- Façade is always used to separate different tiers of a system
  - The Façade controller acts as the single point of entry from UI (presentation) layer to Domain layer
  - We also use Façade to control the communication between domain layer and data layer.

Structural Patterns (GoF)

Pattern Name	Description
Adapter	**Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface clients expect.**
Façade	**Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.**
Composite	Compose objects into tree structures to represents part-whole hierarchies. Let client treat individual objects and compositions of objects uniformly
Proxy	Provide a placeholder for another object to control access to it
Decorator	**Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)**
Bridge	Decouple an abstraction from its implementation so that the two can vary independently
Flight weight	Use sharing to support large numbers of fine-grained objects efficiently

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

Behavioural Design Patterns

Behavioural Patterns

- Concerned with algorithms and the assignment of responsibilities between objects
- Describe patterns of objects and class, and communication between them
- Simplify complex control flow that's difficult to follow at run-time
  - Concentrate on the ways objects are interconnected
- **Behavioural Class Patterns**
  - Use *inheritance* to distribute behavior between classes (algorithms and computation)
- **Behavioural Object Patterns**
  - Use *object composition*, rather inheritance. E.g., describing how group of peer objects cooperate to perform a task that no single object can carry out by itself

Behavioural Patterns (GoF)

Pattern Name	Description
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable (let algorithm vary independently from clients that use it)
Observer	Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
State	Allow an object to alter its behaviour when its internal state changes. The object will appear to change to its class
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates

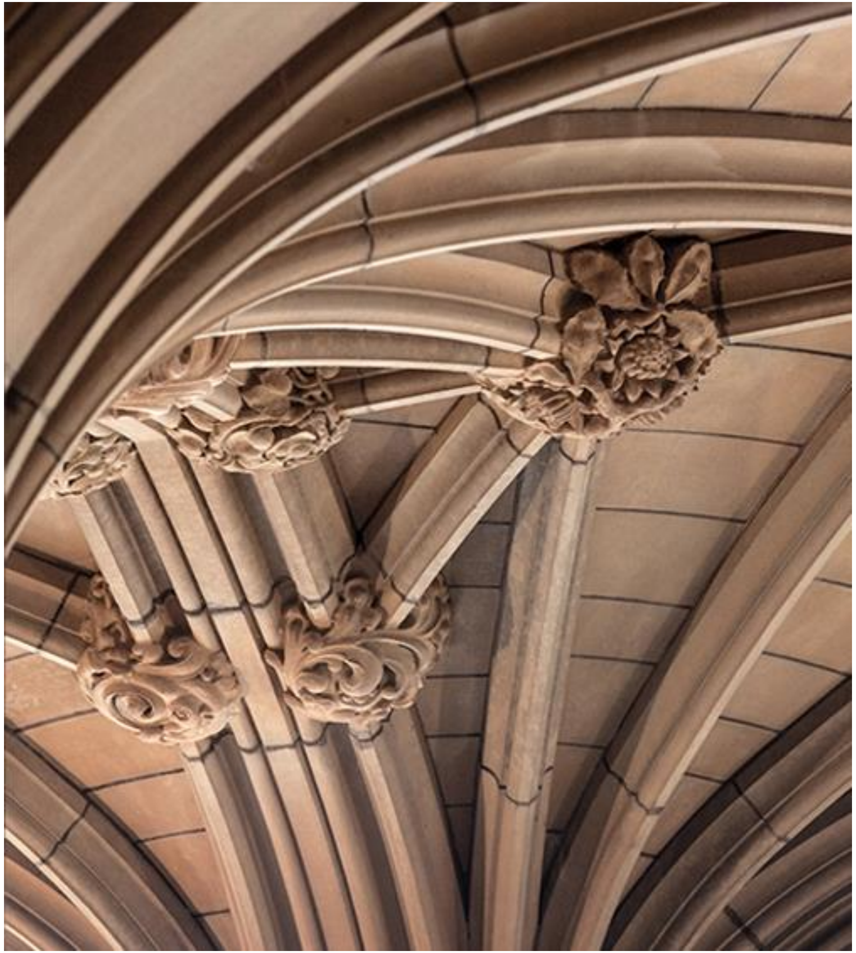
Behavioural Patterns (GoF)

Pattern Name	Description
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
State	Allow an object to alter its behaviour when its internal state changes. The object will appear to change to its class
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates
Other patterns; Chain of responsibility, Command, Mediator, Template Method	

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

Strategy Pattern

Object behavioural



Strategy

- **Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Let the algorithm vary independently from clients that use it
- **Known as**
  - Policy
- **Motivation**
  - Design for varying but related algorithms
  - Ability to change these algorithms

Strategy – Example (Text Viewer)

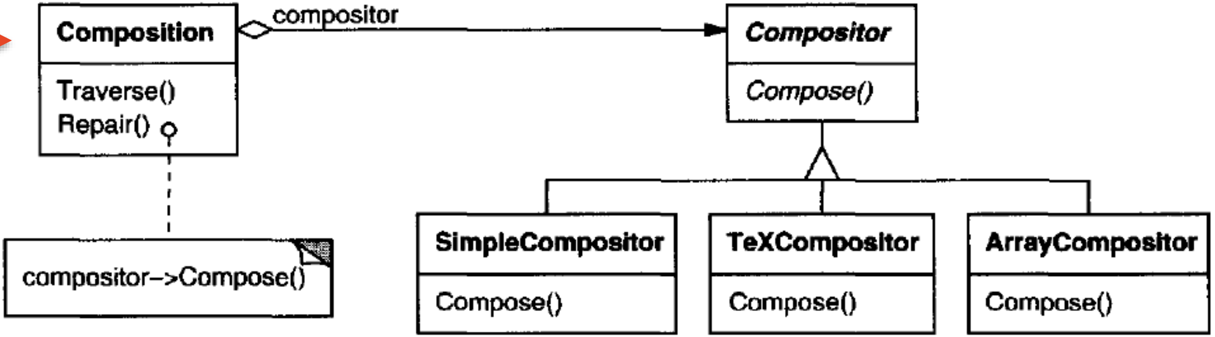
- Many algorithms for breaking a stream of text into lines

- Problem: hard-wiring all such algorithms into the classes that require them

  - More complex and harder to maintain clients (more line breaking algorithms)
  - Not all algorithms will be needed at all times

Strategy – Example (Text Viewer)

Maintain & update the line breaks of text



Different line breaking algorithms (strategies)

Strategy – Applicability

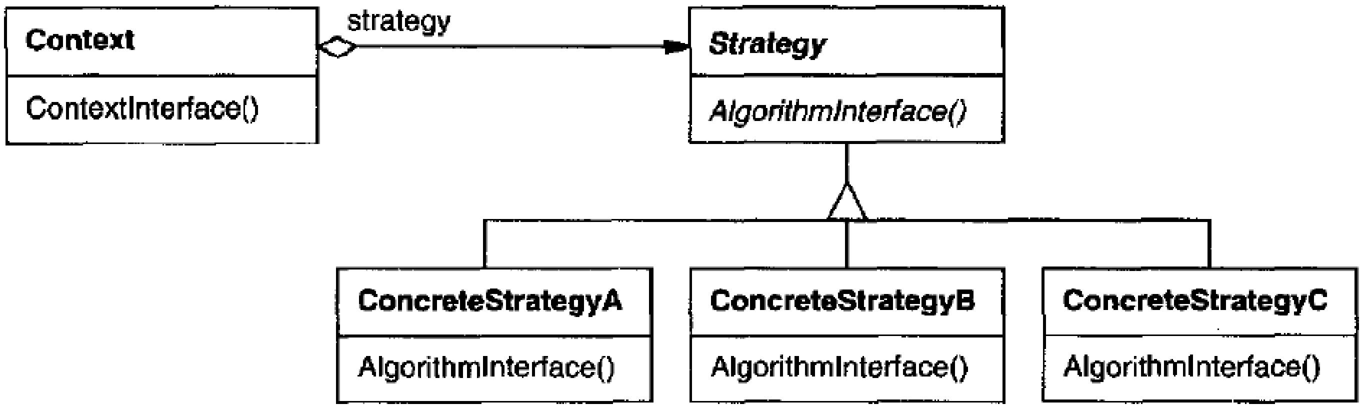
- Many related classes differ only in their *behavior*

- You need different *variant of an algorithm*

- An algorithm uses *data that should be hidden from its clients*

- A class defines many behaviors that appear as multiple statements in its operations

Strategy – Structure



Strategy – Participants

Participant	Goals
Strategy (Compositor)	Declares an interface common to all supported algorithms Used by context to call the algorithm defined by ConcereteStrategy
ConcereteStrategy (SimpleCompositor, TeXCompositor, etc)	Implements the algorithm using the Strategy interface
Context (Compositoion)	Is configured with a ConcereteStrategy object Maintains a reference to a Strategy object May define an interface that lets Strategy access its data

Strategy – Collaborations

- Strategy and Context interact to implement the chosen algorithm
  - A context may pass all data required by the algorithm to the Strategy
  - The context can pass itself as an argument to Strategy operations

- A context forwards requests from its clients to its strategy
  - Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively

Strategy – Consequences (Benefits)

- Family of related algorithms (behaviors) for context to reuse

- Alternative to sub-classing
  - Why not sub-classing a Context class directly to give it different behaviors?

- Strategies eliminate conditional statements

- Provide choice of different implementation of the same behavior

Strategy – Consequences (Drawbacks)

- Clients must be aware of different strategies
  - Understand how strategies differ

- Communicate overhead between Strategy and Context
  - Strategy interface is shared by all ConcereteStrategy classes whether the algorithms they implement are trivial or complex

- Increased number of objects in an application
  - Can be reduced by implementing strategies as stateless objects that context can share
  - Strategy objects often make good flyweight (sharing strategies)

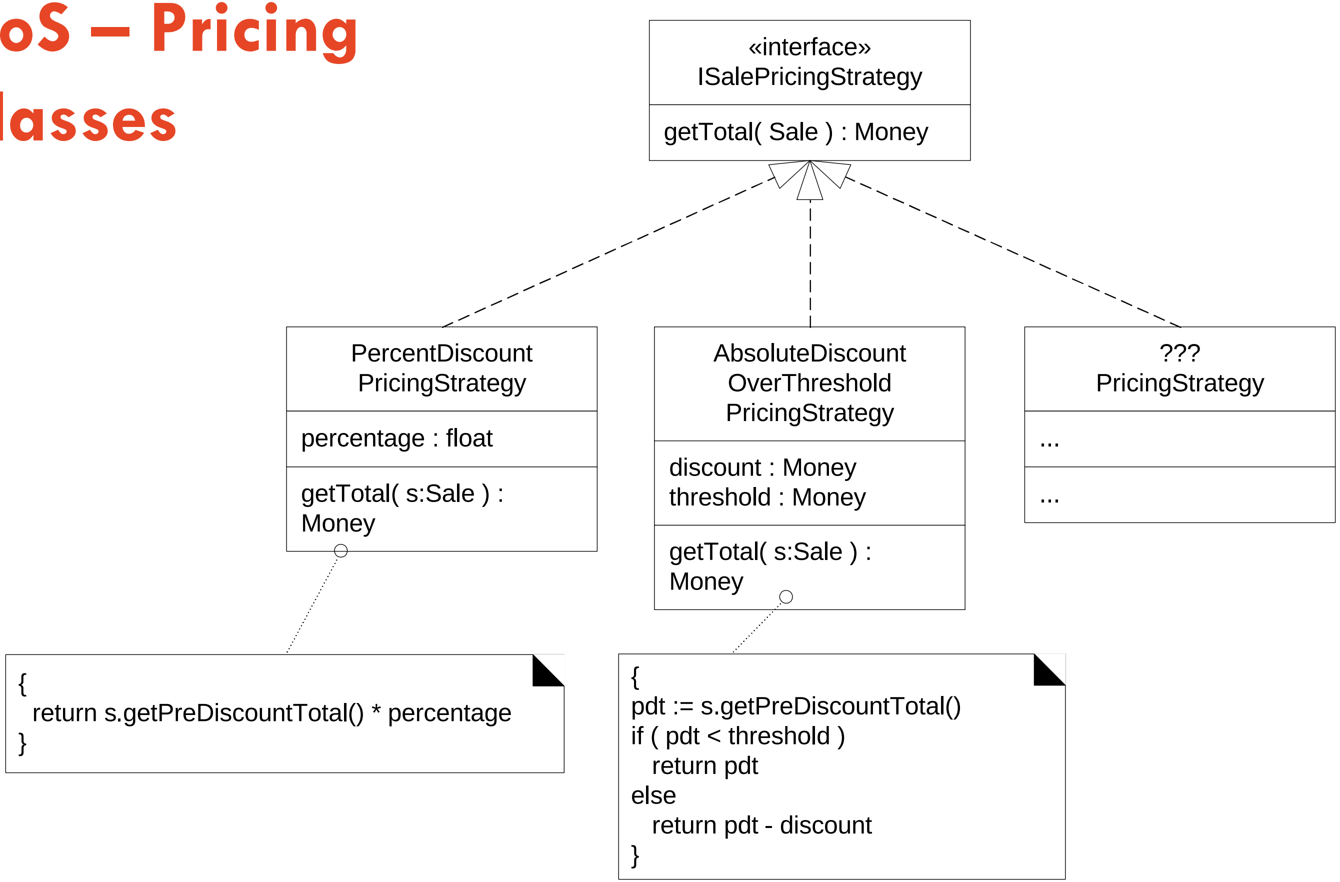
Strategy – NextGen PoS System

- **Design problem:** how to provide more complex pricing logic, e.g., store-wide discount for the day, senior citizen discounts

- The pricing strategy (or policy) for a sale can vary:
  - 10% of all sales during a specific period
  - $10 off if the total sale is greater than $200
  - Other variations

- How do we design our system to accommodate such varying pricing policies?

NextGen PoS – Pricing Strategy Classes



«interface»
ISalePricingStrategy

getTotal(Sale) : Money

PercentDiscount
PricingStrategy

percentage : float

getTotal(s:Sale) :
Money

```
{
  return s.getPreDiscountTotal() * percentage
}
```

AbsoluteDiscount
OverThreshold
PricingStrategy

discount : Money
threshold : Money

getTotal(s:Sale) :
Money

```
{
pdt := s.getPreDiscountTotal()
if ( pdt < threshold )
   return pdt
else
   return pdt - discount
}
```

???
PricingStrategy

...

...

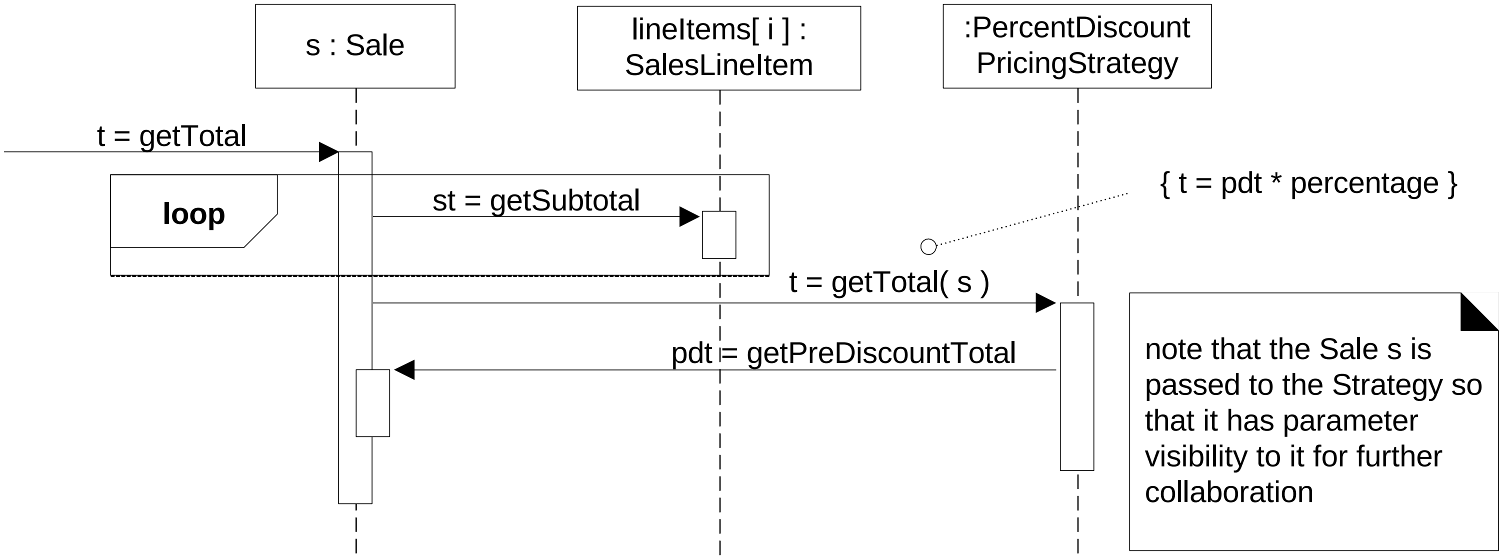
Strategy Pattern – NextGen PoS System

- Create multiple *SalesePricingStrategy* classes, each with a polymorphic *getTotal* method

- Each *getTotal* method takes the *Sale* object as a parameter
  - The pricing strategy object can find the pre-discount price from the Sale, and they apply the discounting policy

- The implementation of each getTotal method will be different
  - E.g., *PercentDiscountPricingStrategy* will discount by a percentage

PoS System – Strategy POS Collaboration

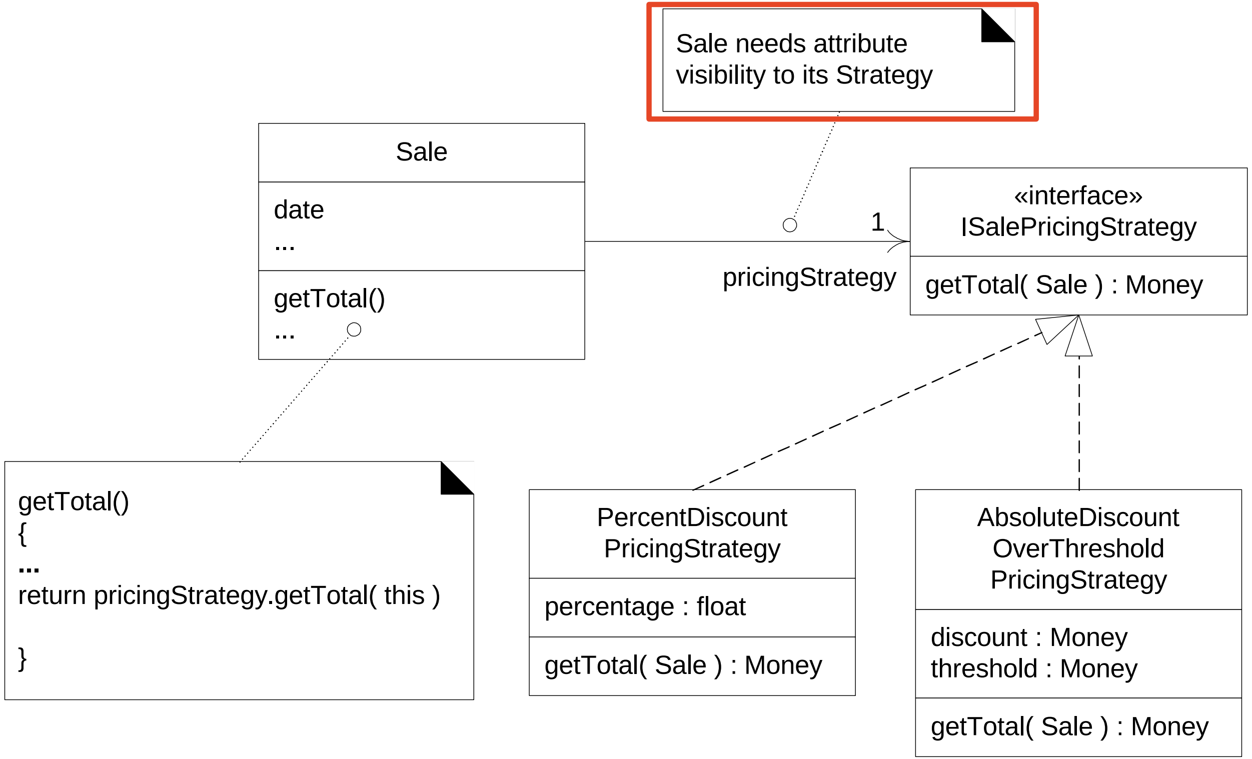

s : Sale
lineItems[i] : SalesLineItem
:PercentDiscount PricingStrategy
t = getTotal
loop
st = getSubtotal
{ t = pdt * percentage }
t = getTotal(s)
pdt = getPreDiscountTotal
note that the Sale s is passed to the Strategy so that it has parameter visibility to it for further collaboration

PoS System – Strategy POS Collaboration

- A strategy is attached to the *Sale* object (context object)

- The *Sale* object delegates some of the work to its strategy object
  - The message to the context and strategy objects is not required to be the same (e.g., getTotal)
  - The Sale object passes a reference to itself on to the strategy object

PoS System – Attributes Visibility

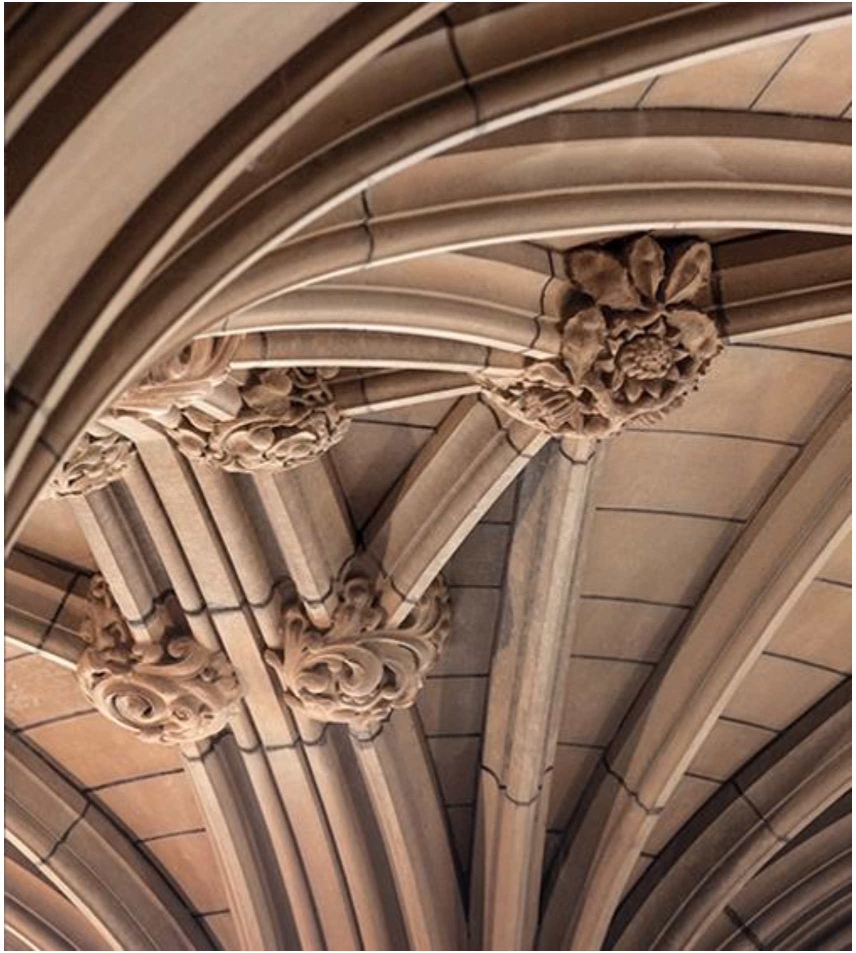


Sale needs attribute visibility to its Strategy

```
getTotal()
{
...
return pricingStrategy.getTotal( this )

}
```

Observer

Object Behavioural



Observer Pattern

- **Intent**: define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Known as**: Dependents, Publish-Subscribe
- **Motivation**:
  - Partitioning a system into a collection of cooperating classes help to maintain consistency between related objects
  - How to achieve consistency while maintaining classes loosely-coupled, and highly reusable?

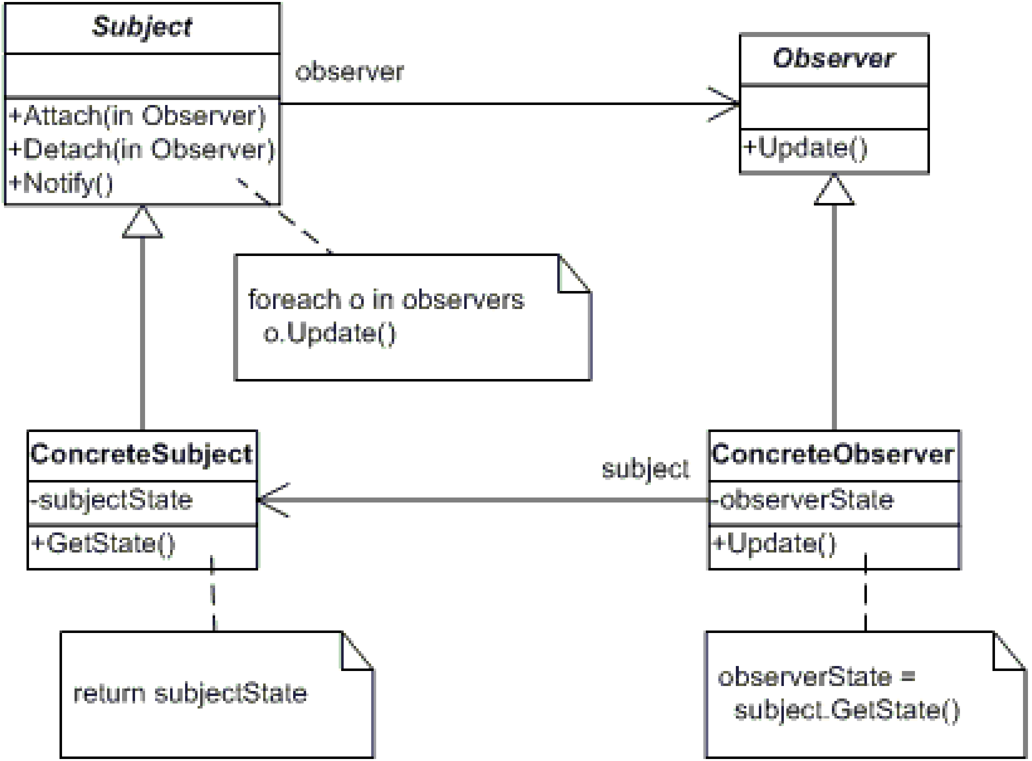
Observer Pattern – Applicability

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets varying and using them independently

- When a change to one object requires changing others, and it’s not clear how many objects need to be changed

- When an object should be able to notify other objects without making assumptions about who these objects are (keep these objects loosely-coupled)

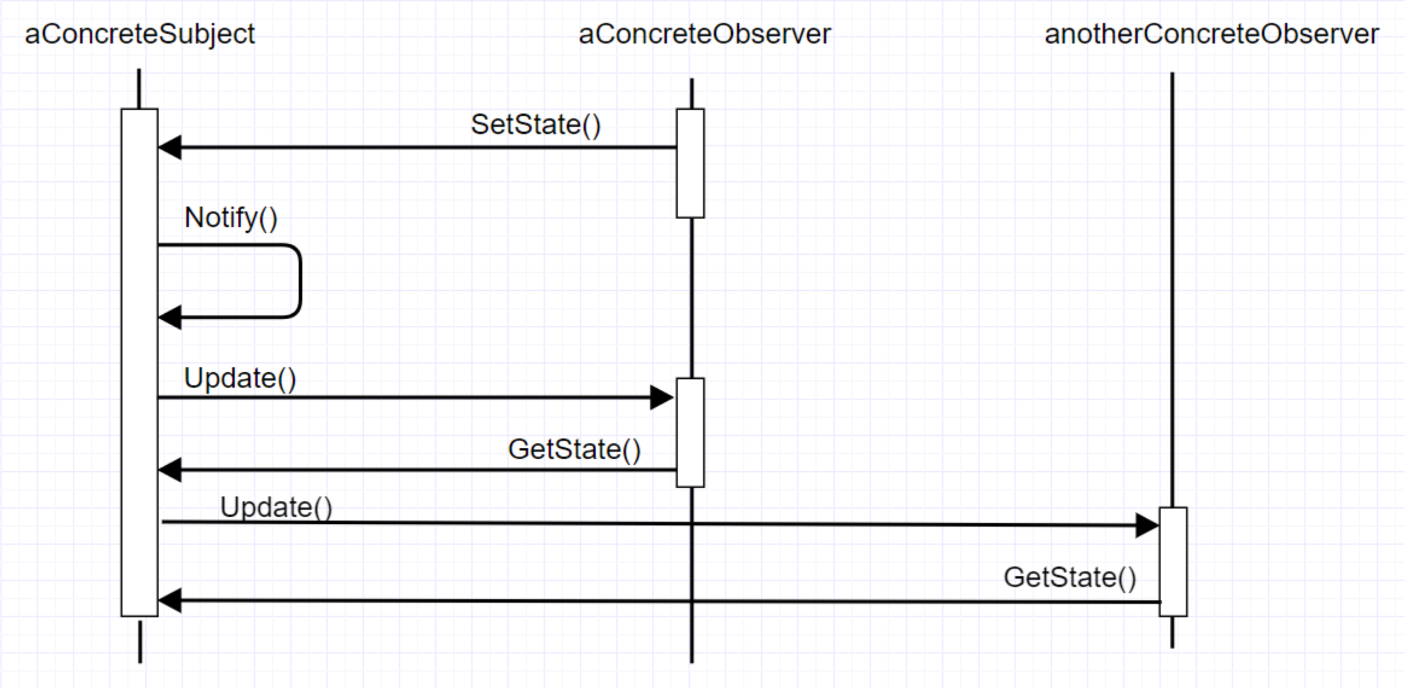
Observer Pattern – Structure



Observer Pattern – Participants

Participant	Goals
Subject	knows its observers, any number of Observer objects may observe a subject Provides an interface for attaching and detaching Observer objects
Observer	defines an updating interface for objects that should be notified of changes in a subject
ConcereteSubject	Stores state of interest to ConcereteObserver objects Sends notifications to its observers when its state changes
ConcreteObserver	Maintains a reference to a ConcereteSubject object Stores state that should stay consistent with the subject's Implemenets the Observers updating interface to keep its state consistent

Observer Pattern – Collaborations



Observer Pattern – Consequences/Benefits

- Abstract coupling between Subject and Observer
  - *Subject* only knows its *Observers* through the abstract *Observer* class (it doesn't know the concrete class of any observer)

- Support for broadcast communication
  - Notifications are broadcasted automatically to all interested objects that subscribe to the *Subject*
  - Add/remove Observers anytime

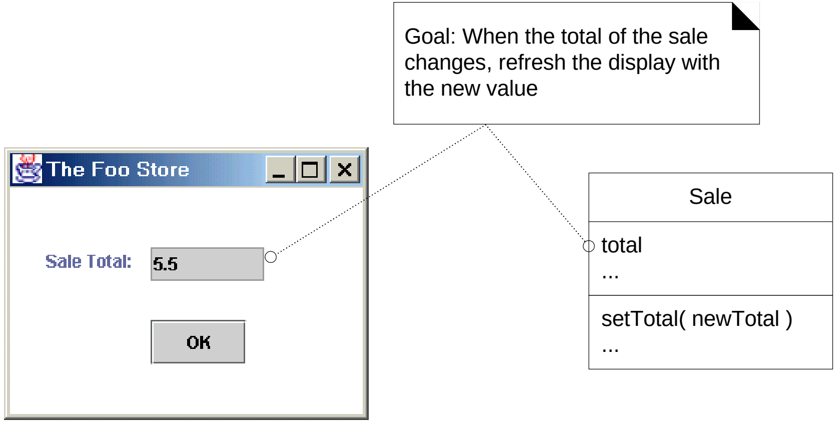
- Unexpected updates
  - Observers have no knowledge of each other's presence, so they can be blind to the cost of changing the subject
  - An innocent operation on the subject may cause a cascade of updates to Observers and their dependents

Observer Pattern – PoS System

- PoS system requirement (iteration 2):
  - A GUI window to refresh its display of the Sale total when the total changes
  - In next iterations; extend the solution to refreshing the GUI display for other changing data

Discuss the following solution:

When the *Sale* object changes its total, the Sale object sends a message to a window (GUI), asking it to refresh its display

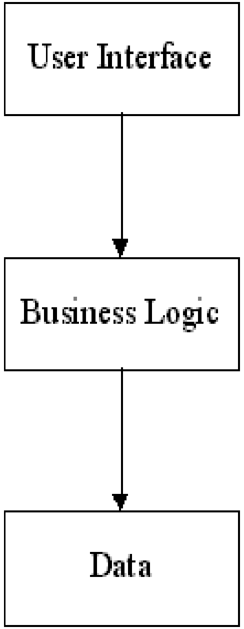

Goal: When the total of the sale changes, refresh the display with the new value

Observer in POS

- Problem of the naïve solution
  - Violation of layer dependency principle
  - Specially, violate the <u>Model-View Separation</u> principle

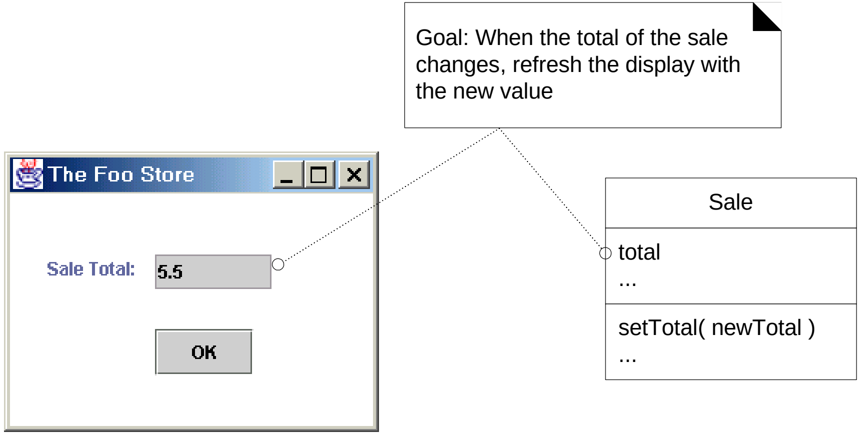
- Model-View Separation principle
  - Do not connect or couple non-UI objects directly to UI objects
  - Do not put application logic in the UI object methods.

- We also want it flexible to plug/unplug certain UI objects.



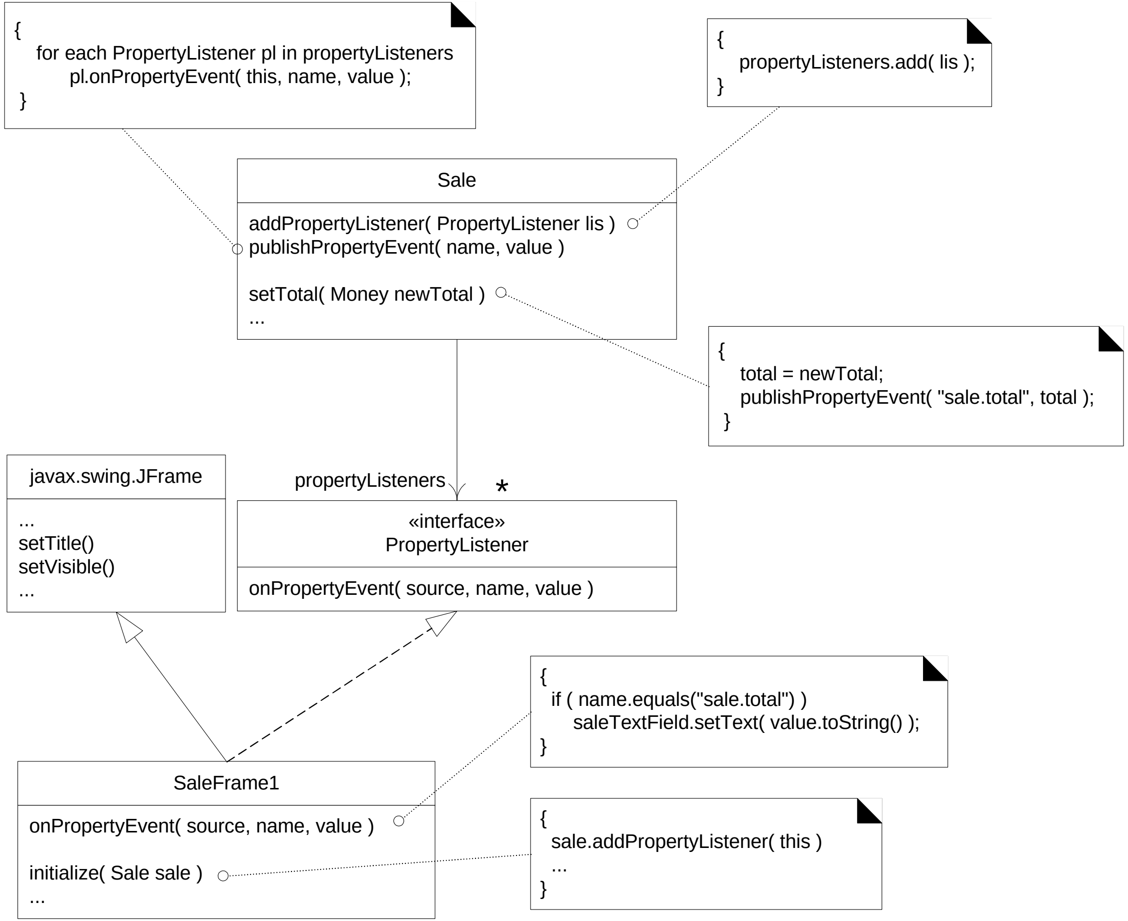
Observer Pattern – PoS System

- Model-view separation principle (low-coupling of model and UI layers)
  - “model” (e.g., Sale object) should not know/update “view” presentation objects (e.g., window)
  - Allows replacing (or changing) the UI without influencing the model (Sale object)



Goal: When the total of the sale changes, refresh the display with the new value

Observer in POS – Solution

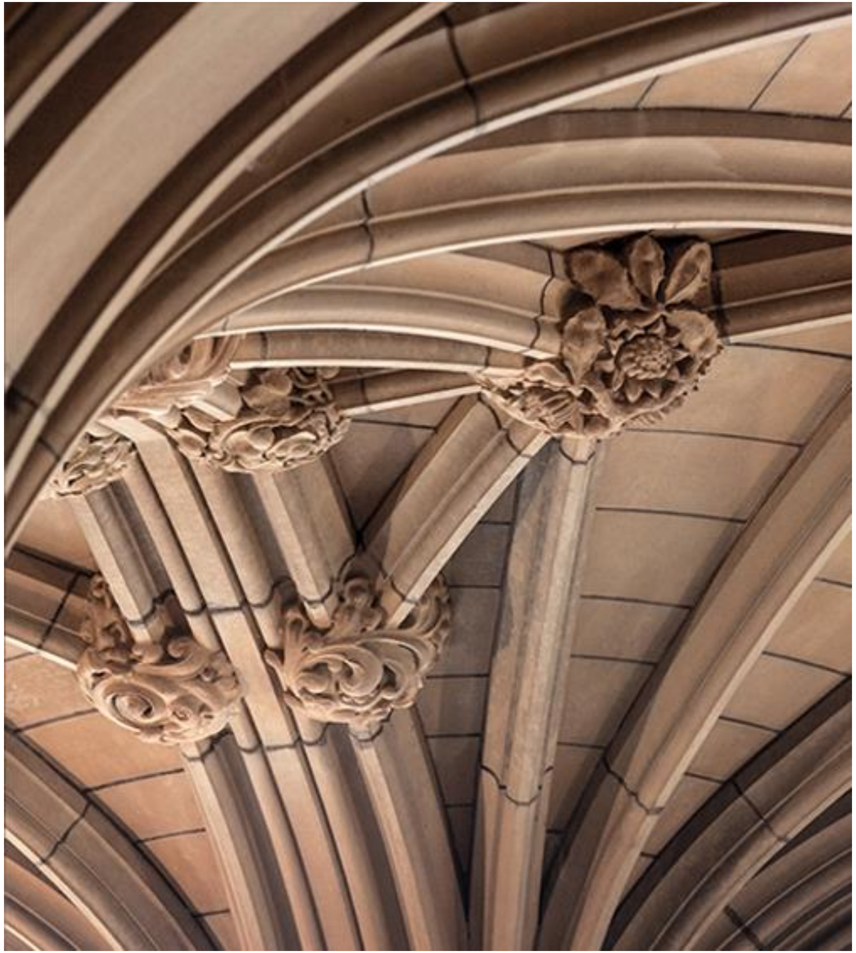

{
for each PropertyListener pl in propertyListeners
pl.onPropertyEvent(this, name, value);
}
{
propertyListeners.add(lis);
}
Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
{
total = newTotal;
publishPropertyEvent("sale.total", total);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface»
PropertyListener
onPropertyEvent(source, name, value)
{
if (name.equals("sale.total"))
saleTextField.setText(value.toString());
}
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
{
sale.addPropertyListener(this)
...
}

Observer in POS – Solution

- An interface is defined – **`PropertyListener`** with the operation **`onPropertyEvent`**
- Define an UI object to implement the interface -- **`SalesFrame1`**
- When the SalesFrame1 window is initialized, pass it the `Sale` instance from which it is displaying the total
- The **`SaleFrame1`** window registers or subscribes to the **`Sale`** instance for notification of "property events", via the **`addPropertyListener`** message.
- The **`Sale`** instance, once its total changes, iterates across all subscribing **`PropertyListeners`**, notifying each

Command Design Pattern

Object Behavioural



Command Pattern

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- Applicability
  - To parameterize objects by an action to perform – like callback functions
  - To specify, queue and execute requests at different times
    - A command object can have a lifetime independent of the original request
  - To support undo
    - The Command's Execute operation can store state for reversing its effects in the command itself

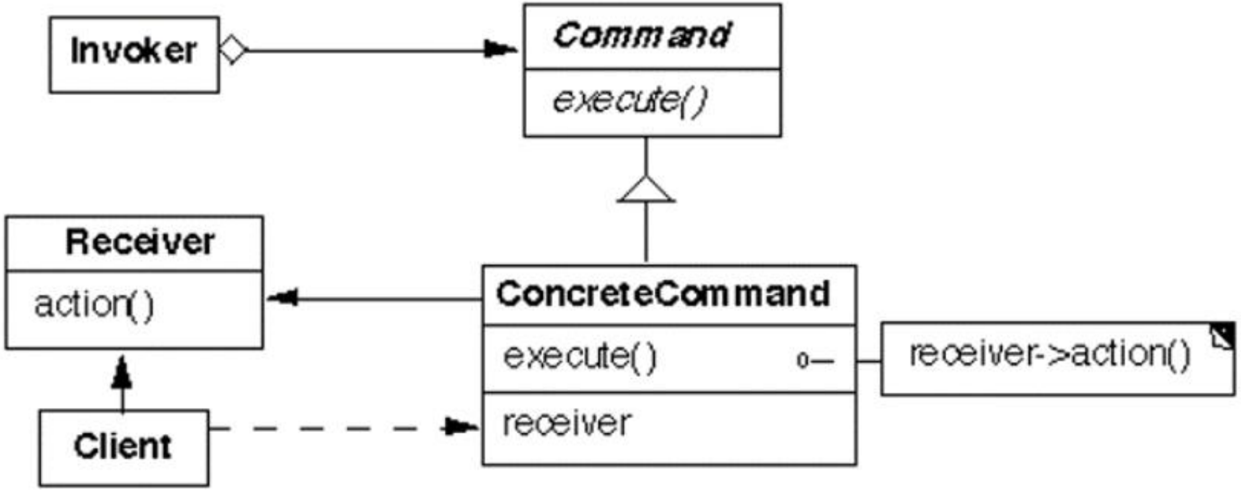
Command Pattern – Applicability

- To support logging changes so it can be applied in case of a system crash
  - Load and Store operations in the Command interface to keep a persistent log of change

- To structure a system around high-level operations built on primitive operations
  - E.g., transaction systems maintain set of changes to data
  - Commands have a common interface, you can invoke all transactions the same way
  - Also can extend the system with new transactions

Command Pattern – Structure

– •



Command – Structure (Participants)

- **Command**
  - Declares an interface for executing an operation
- **ConcreteCommand** (PasteCommand, OpenCommand)
  - Defines a binding between a Receiver object and an action
  - Implements Execute by invoking the corresponding operation(s) on Receiver
- **Client (Application)**
  - Creates a ConcreteCommand object and sets its receiver
- **Invoker** (MenuItem)
  - Asks the command to carry out the request
- **Receiver**
  - Knows how to perform the operations associated with carrying out a request

Command – Toolkits User Interface Example

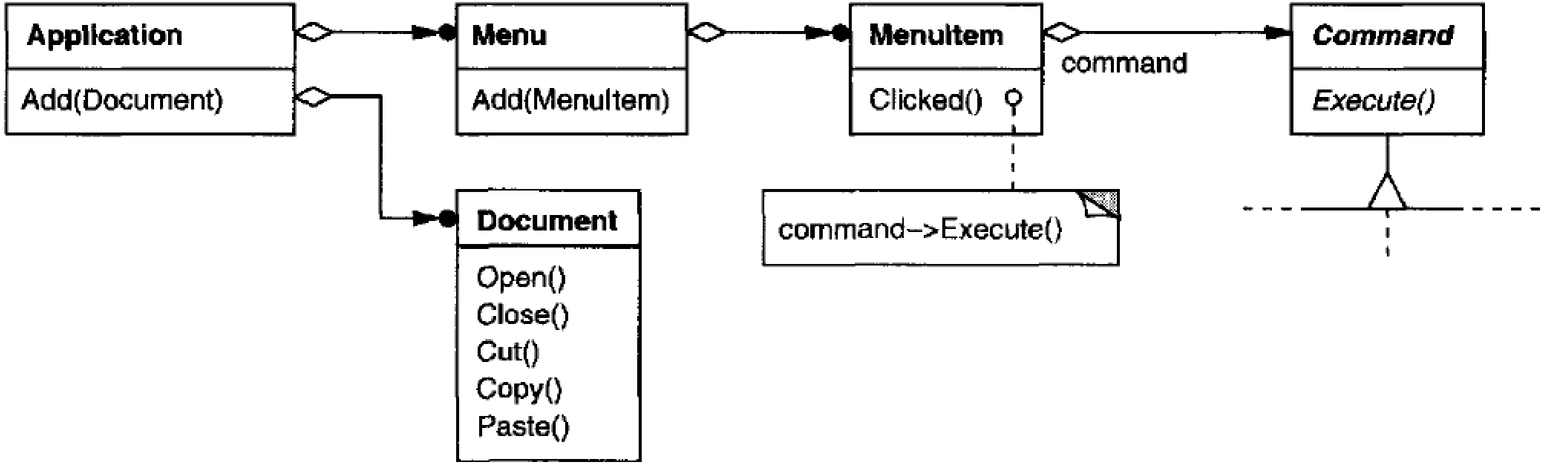
- Consider a user interface toolkits that include objects like buttons and menus that carry out a request in response to user input

- The toolkit cannot implement the request in the button or menu objects; applications that use the toolkit know what should be done on which object

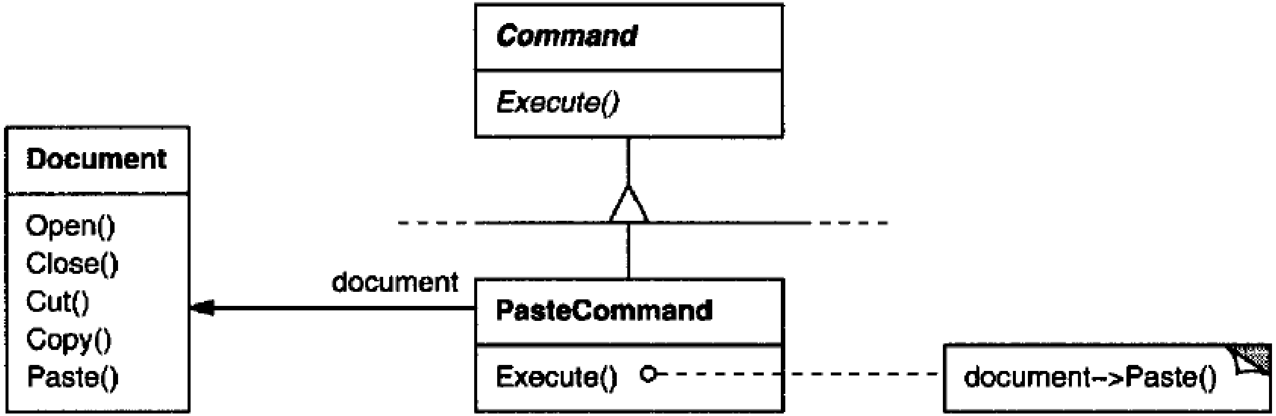
- Requests will be issued to objects without knowing anything about the operation being requested or the receiver of the request

Command – Example

– •

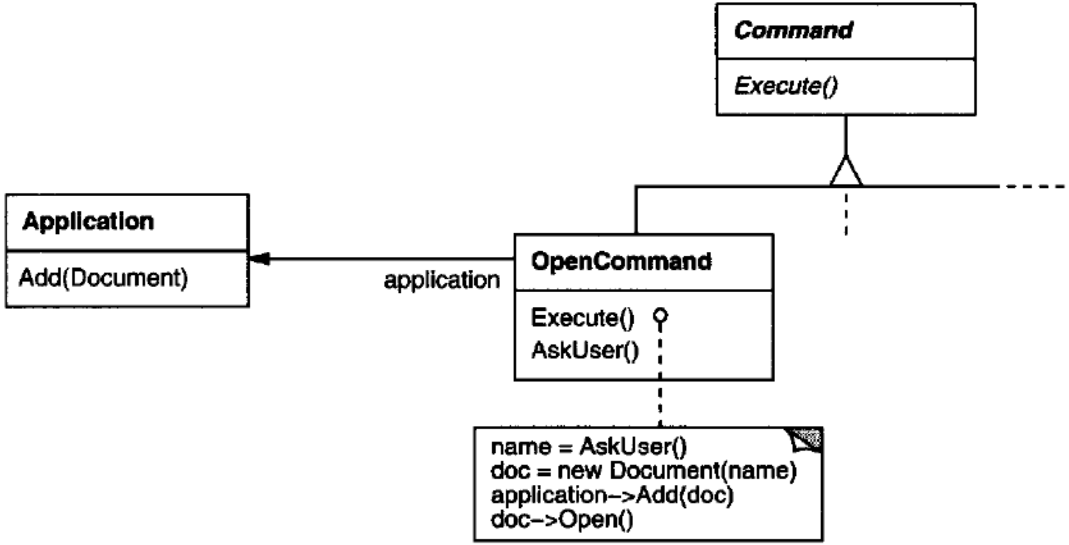


Command – Toolkit (Paste Command)



PasteCommand allows pasting text from the clipboard into a document

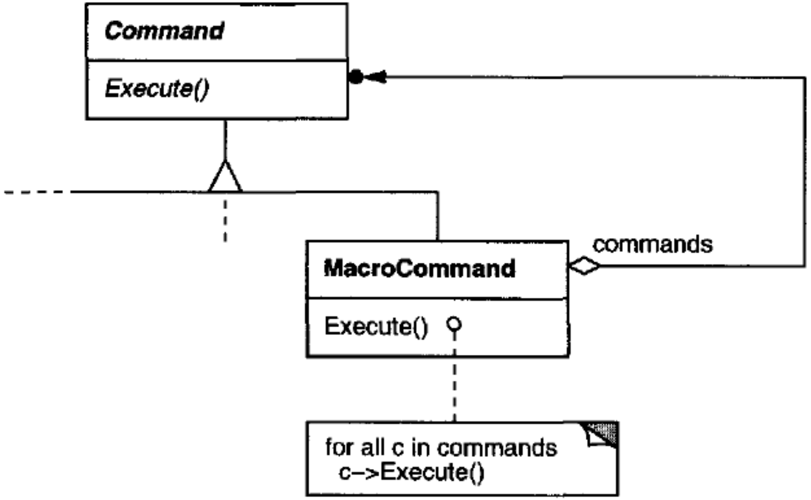
Command – Toolkit (Open Command)



```
name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()
```

OpenCommand’s Execute operation

Command – Toolkit (Sequence of Commands)



- MenuItem needs to execute a sequence of commands
  - E.g., MenuItem for centering a page at normal size constructed from CenterDocCommand and NormalSizeCommand

- MacroCommand class allow menuItem to execute sequence of commands

Behavioural Patterns (GoF)

Pattern Name	Description
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
State	Allow an object to alter its behaviour when its internal state changes. The object will appear to change to its class
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates
Other patterns; Chain of responsibility, Command, Mediator, Template Method	

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

- Martin Folwer, Patterns In Enterprise Software, [https://martinfowler.com/articles/enterprisePatterns.html]