So Far …

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities
- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code
- Test-Driven Development and Refactoring

Part: 4 Elaboration Iteration 2—More Patterns

- GRASP: More Objects with Responsibilities
- Applying GoF Design Patterns
- S.O.L.I.D

Applying GoF Design Patterns

Abdulkareem Alali

Based of Larman’s Applying UML and Patterns Book, 3d

The shift of focus (to patterns) will have a profound and enduring effect on the way we write programs.

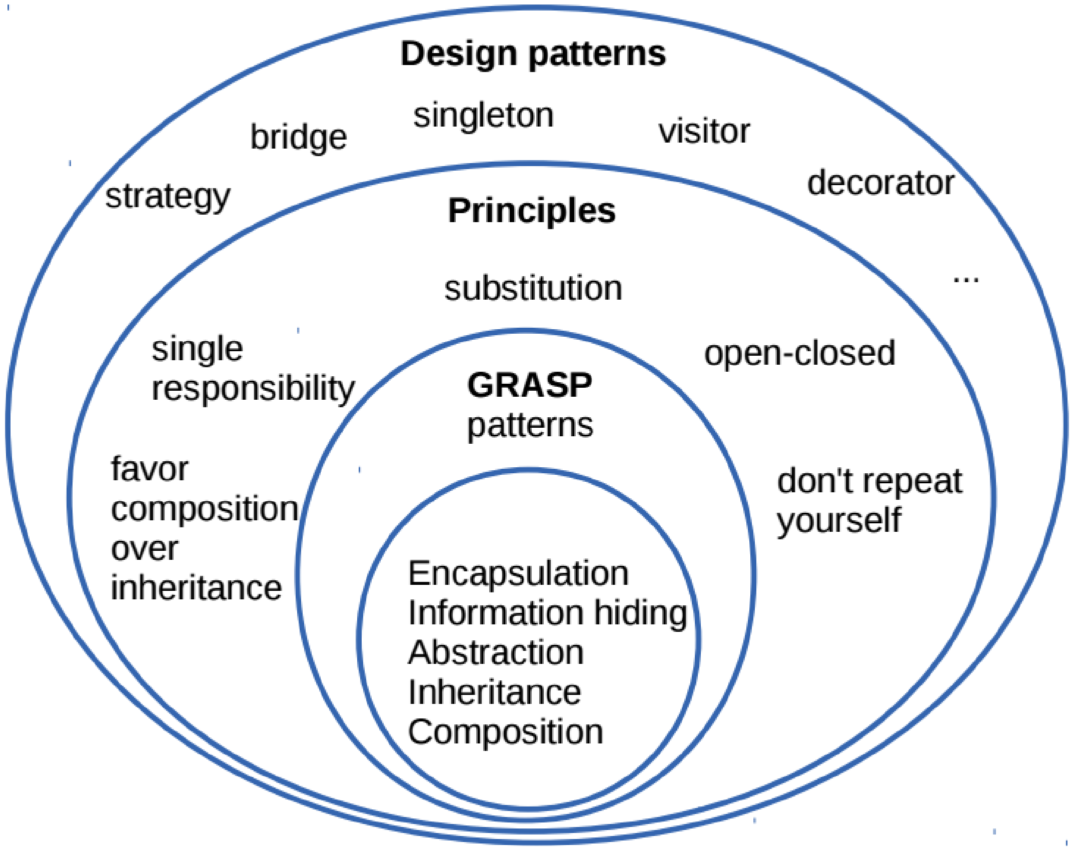
—Ward Cunningham and Ralph Johnson

How to Design Object-Oriented?

There's **no methodology** to get the best object-oriented design,

but there are

Principles, Patterns, Best Practices, and **heuristics**.



GoF

The idea of named patterns in software comes from Kent Beck (also of Extreme Programming fame) in the mid 1980s

1994 a major milestone in software design,
the Design Patterns Book was published:

- Authored by Gamma, Helm, Johnson, and Vlissides (**Gang of Four**)
- "**Bible**" of design pattern
- Describes **23** patterns for OO design

Not all of the **23** patterns are widely used; **15** are common and most useful

23 Gof, Foundation For All Other Patterns

- Creational Patterns (1-5)

- Structural Patterns (6-12)

- Behavioral Patterns (13-23)

23 Gof, Foundation For All Other Patterns

1. Abstract **Factory**
2. Builder
3. Factory Method
4. Prototype
5. **Singleton**
6. **Adapter**
7. Bridge
8. **Composite**
9. Decorator
10. **Façade**
11. Flyweight
12. Proxy
13. Chain of Resp.
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. **Observer**
20. State
21. **Strategy**
22. Template
23. Visitor

Adapter

Adapter

Problem

How to resolve incompatible interfaces, or provide

A **stable** interface (local) to similar components with different interfaces (APIs)?

Solution

Convert the original interface of a component into another interface, through an intermediate **adapter** object

POS Third-Party Services

POS needs to support several kinds of external third-party services, including:

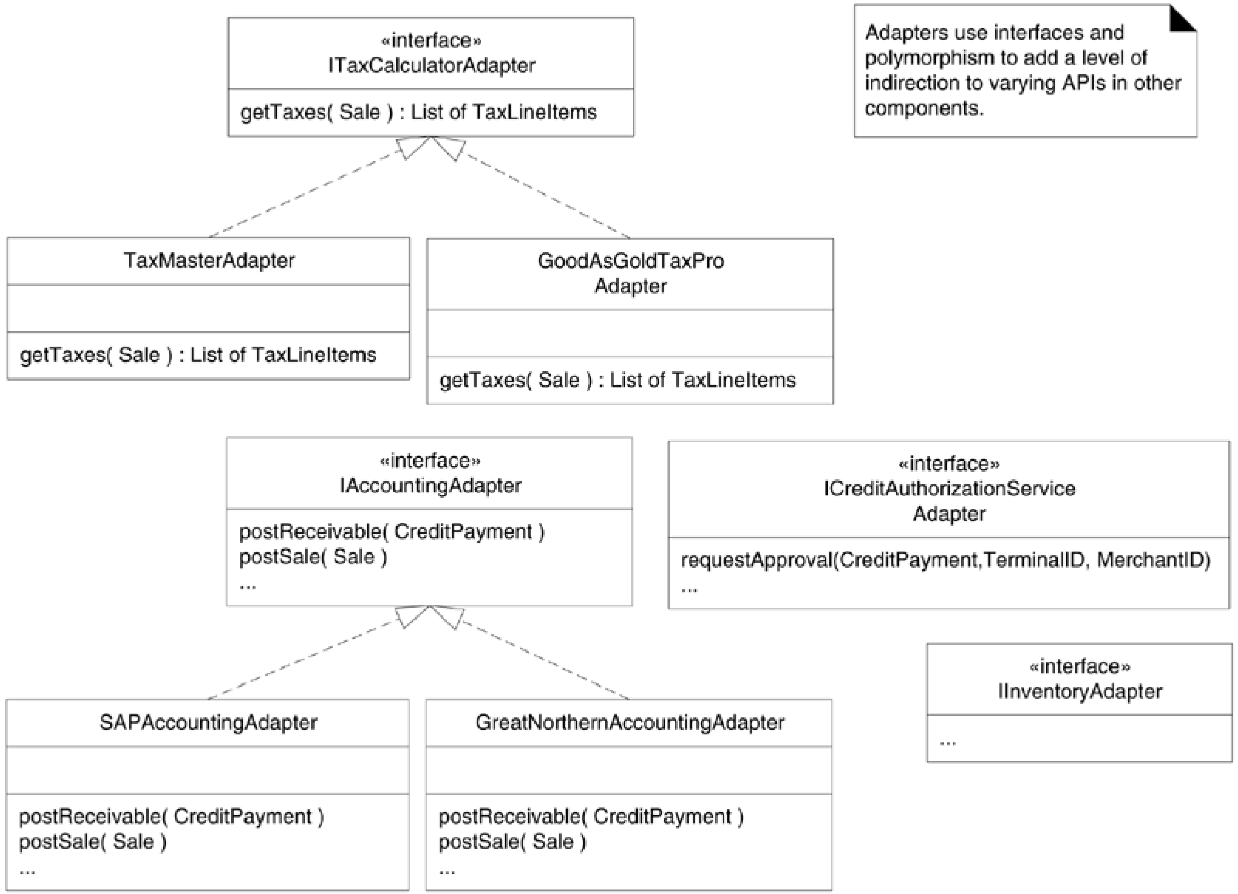
- Tax calculators,
- Credit authorization services,
- Inventory systems,
- Accounting systems, etc.

Each has a different API, different service, which can be changed

Solution: Add a level of **indirection** with objects that

Adapt varying external interfaces to a consistent local interface

Adapters For Third Party APIs



«interface»
ITaxCalculatorAdapter

getTaxes(Sale) : List of TaxLineItems

Adapters use interfaces and polymorphism to add a level of indirection to varying APIs in other components.

TaxMasterAdapter

getTaxes(Sale) : List of TaxLineItems

GoodAsGoldTaxPro Adapter

getTaxes(Sale) : List of TaxLineItems

«interface»
IAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

«interface»
ICreditAuthorizationService Adapter

requestApproval(CreditPayment,TerminalID, MerchantID)
...

SAPAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

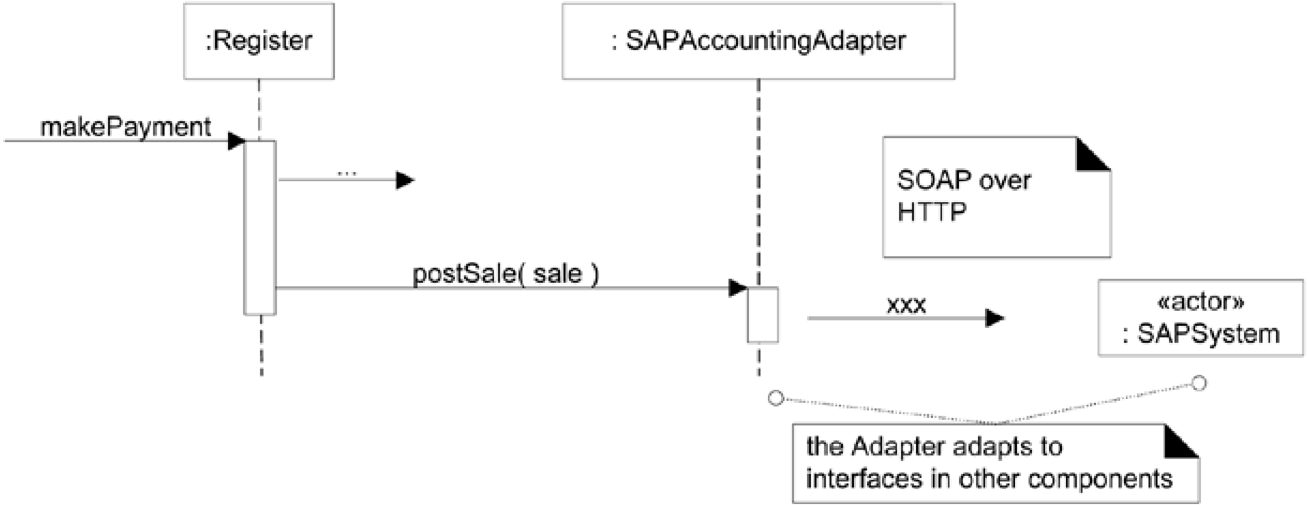
GreatNorthernAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

«interface»
IInventoryAdapter

...

SAPAccountingAdapter



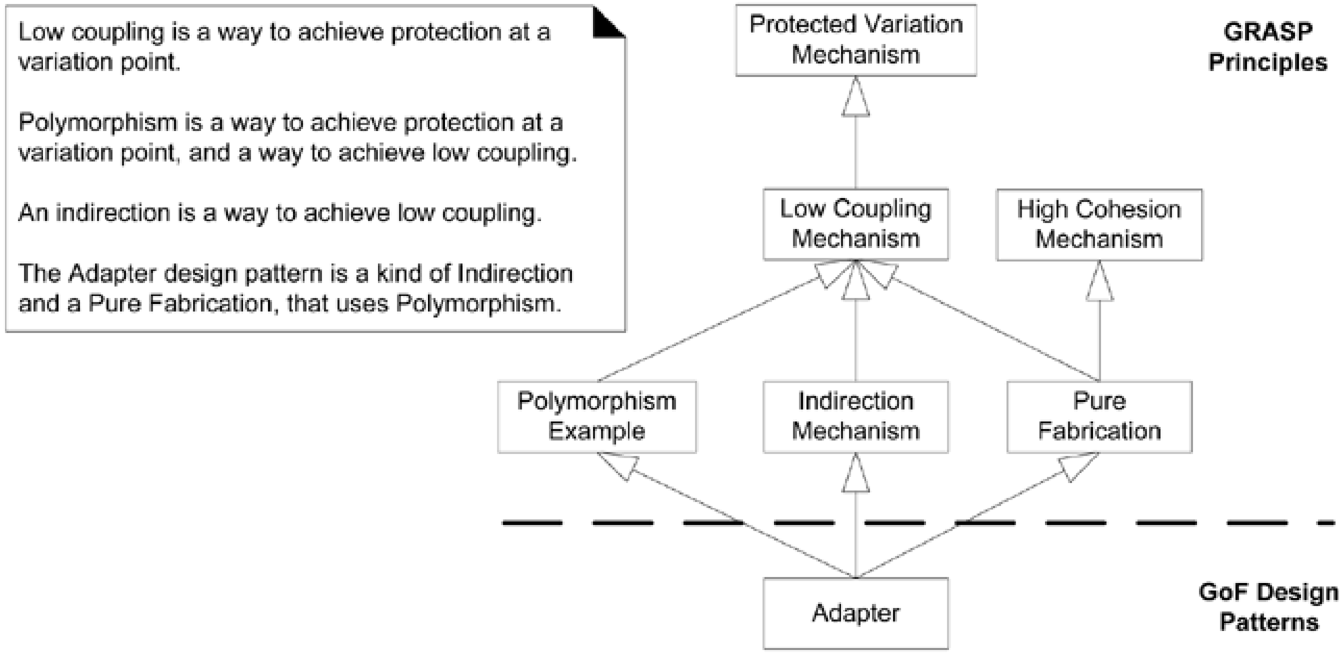
GRASP Principles As A Generalization Of Other Patterns

The Pattern Almanac 2000 book lists around **500** design patterns, + hundreds published since then **~1000**

Most design patterns can be seen as **specializations** of a few basic **GRASP** principles

Detailed Design Patterns To Accelerate Learning

Relating Adapter To Some Core GRASP Principles



Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.

Factory

Factory (GoF: Abstract Factory)

Adapter have solution for external services with varying interfaces,

but who creates the adapters?

e.g. **TaxMasterAdapter** or **GoodAsGoldTaxProAdapter**

Domain layer objects vs. Separation of Concerns Modularize or separate distinct concerns into different areas (GRASP High Cohesion principle)

Factory (GoF: Abstract Factory)

Domain layer of software objects pure application logic responsibilities

vs.

Other group of objects is responsible for connectivity to external systems (**Register** is a bad choice)

Solution: Factory!

Factory Advantages

Separate responsibility of complex creation into cohesive helper objects (**Pure Fabrication**)

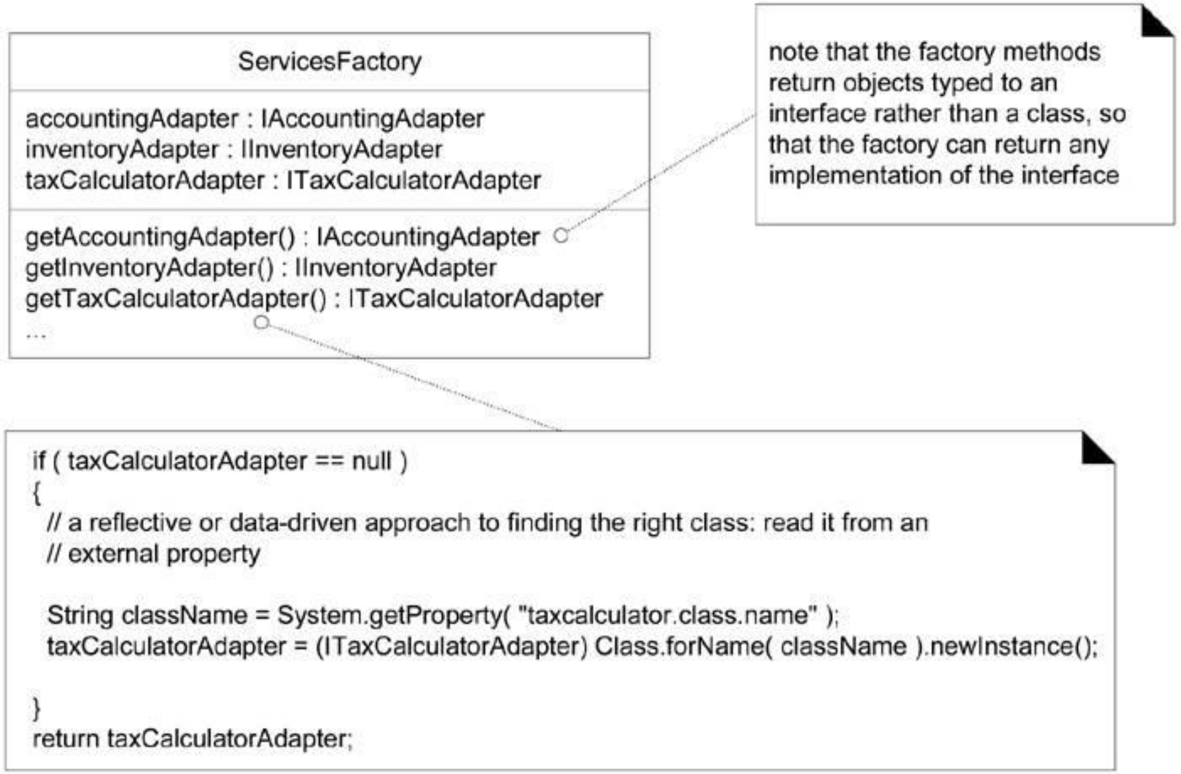
Hide potentially complex creation logic

Allow performance-enhancing memory management strategies, such as object caching or recycling

Factory Pattern

Name	**Factory**
Problem	Who should be responsible for creating objects when there are special considerations, such as 1. complex creation logic, 2. A desire to separate the creation responsibilities for better cohesion, and so forth?
Solution	Create a Pure Fabrication object called a **Factory** that handles the creation.

ServicesFactory


ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```
if ( taxCalculatorAdapter == null )
{
  // a reflective or data-driven approach to finding the right class: read it from an
  // external property

  String className = System.getProperty( "taxcalculator.class.name" );
  taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();

}
return taxCalculatorAdapter;
```

Singleton

Singleton

*Who creates the **ServicesFactory** itself, and how is it accessed?*

1. One instance the factory is needed
2. Methods of this factory may need to be called from different places who in need for an access to the adapters to call external services

*How to get visibility to this single **ServicesFactory** instance?*

Solutions:

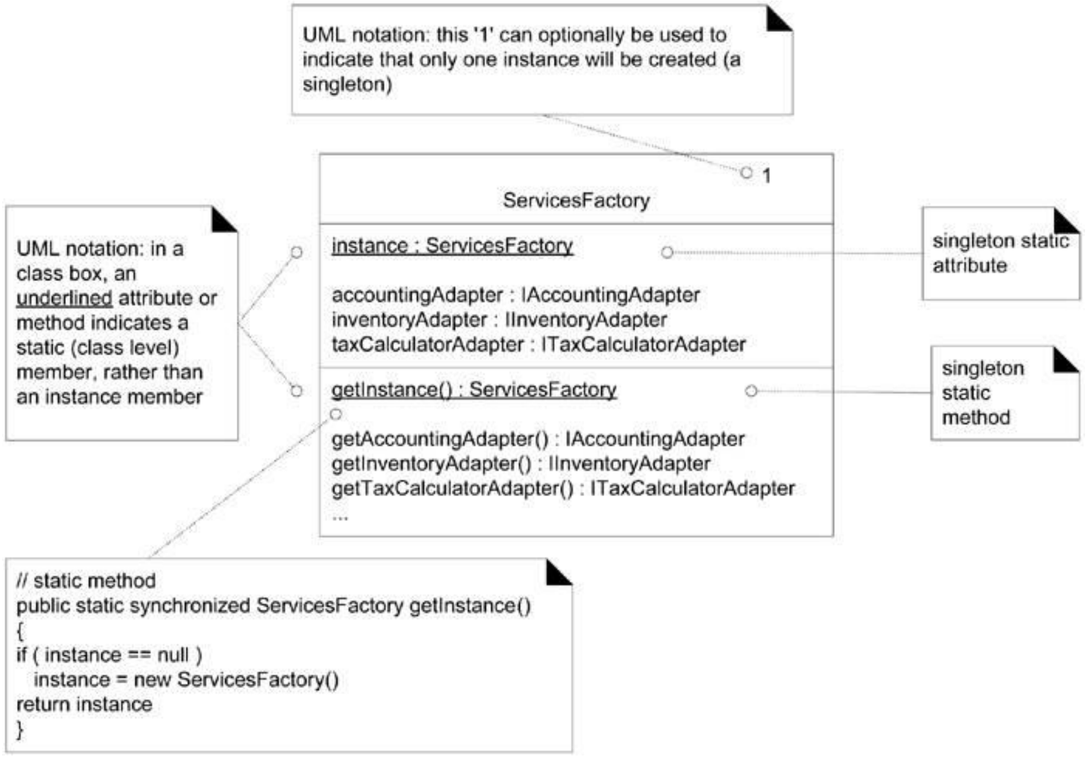
1. Pass the **ServicesFactory** instance around as a parameter to wherever a visibility needed, inconvenient!
2. **Singleton Pattern**:

   Global visibility or a Single access point to a single instance of a class

Singleton Pattern

Name	**Singleton**
Problem	Exactly one instance of a class is allowed—it is a “singleton.” Objects need a global and single point of access.
Solution	Define a **static method** of the class that **returns the singleton**

The Singleton Pattern in the ServicesFactory Class



UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)

1

ServicesFactory

instance : ServicesFactory

accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter

getInstance() : ServicesFactory

getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member

singleton static attribute

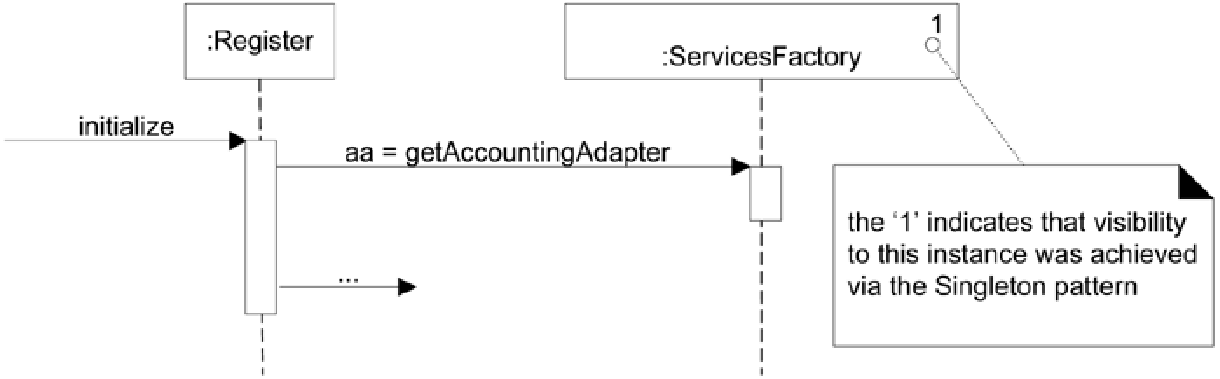
singleton static method

```
// static method
public static synchronized ServicesFactory getInstance()
{
if ( instance == null )
   instance = new ServicesFactory()
return instance
}
```

Singleton Example

```
public class Register
{
   public void initialize()
   {
      ... do some work ...
      // accessing the singleton Factory via the getInstance call
      accountingAdapter =
         ServicesFactory.getInstance().getAccountingAdapter();
      ... do some work ...
   }
   // other methods...
} // end of class
```

Singleton Example



the '1' indicates that visibility to this instance was achieved via the Singleton pattern

Implementation and Design Issues

Lazy Initialization

```
public static synchronized ServicesFactory getInstance()
{
   if ( instance == null )
   {
      // critical section if multithreaded application
      instance = new ServicesFactory();
   }
   return instance;
}
```

Implementation and Design Issues

Eager initialization

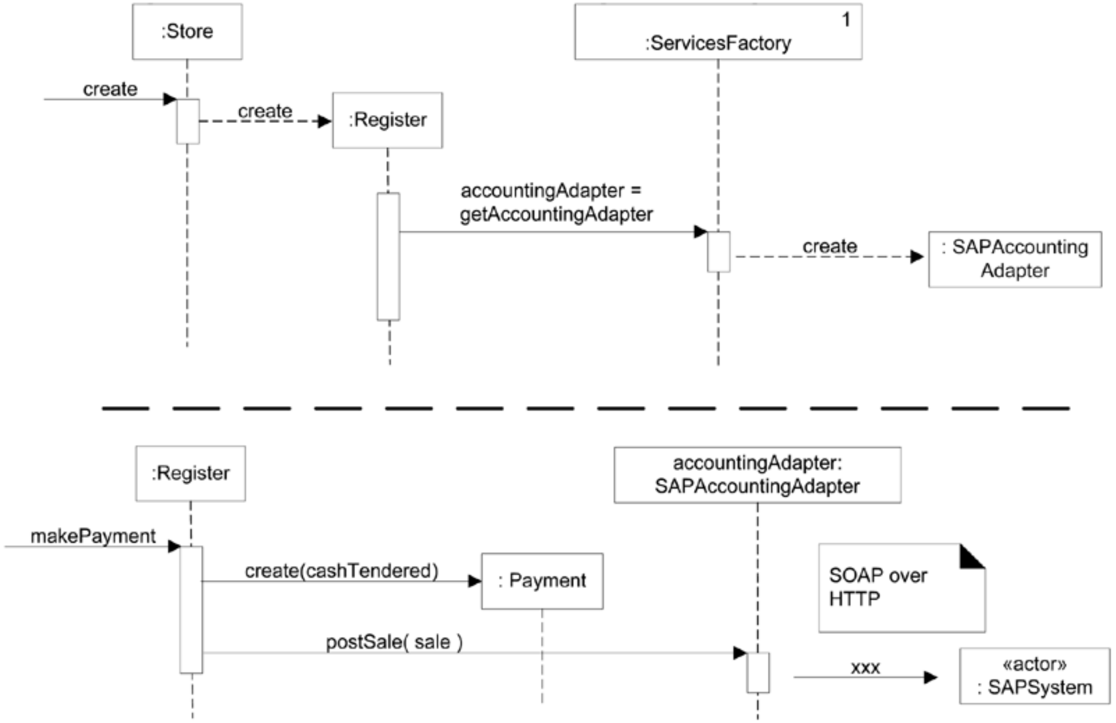
```
public class ServicesFactory
{
   private static ServicesFactory instance = new ServicesFactory();
   public static ServicesFactory getInstance()
   {
       return instance;
   }
}
// other methods.
```

Lazy Initialization Is Usually Preferred

- Creation work (and perhaps holding on to “expensive” resources) is avoided, if the instance is never actually accessed

- The ***getInstance*** lazy initialization sometimes contains complex and conditional creation logic

How Many Patterns?



Patterns As Building Blocks

A combination of **Adapter**, **Factory**, and **Singleton** patterns have been used to provide **Protected Variations** from the varying interfaces of external tax calculators, accounting systems, etc.

Actually it is: **Controller**, **Creator**, **Protected Variations**, **Low Coupling**, **High Cohesion**, **Indirection**, **Polymorphism**, **Adapter**, **Factory**, and **Singleton**

I can say, “To handle the problem of varying interfaces for external services, let's use Adapters generated from a Singleton Factory.”

Object designers really do have conversations that sound like this; using patterns and pattern names supports

Raising The Level Of Abstraction In Design Communication

Strategy (Best!)

Strategy

Problem: Complex pricing logic:

store-wide discount for the day,
senior citizen discounts, etc.

Pricing Strategy (A rule, policy, or algorithm) for a sale can vary

e.g.: During one period it may be 10% off all sales,
later it may be $10 off if the sale total is greater than $200,
and myriad other **variations**

How do we design for these varying pricing algorithms?

Strategy Pattern

Name	**Strategy**
Problem	How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?
Solution	Define each algorithm/policy/ **strategy in a separate class, with a common interface.**

Pricing Strategy, Solution

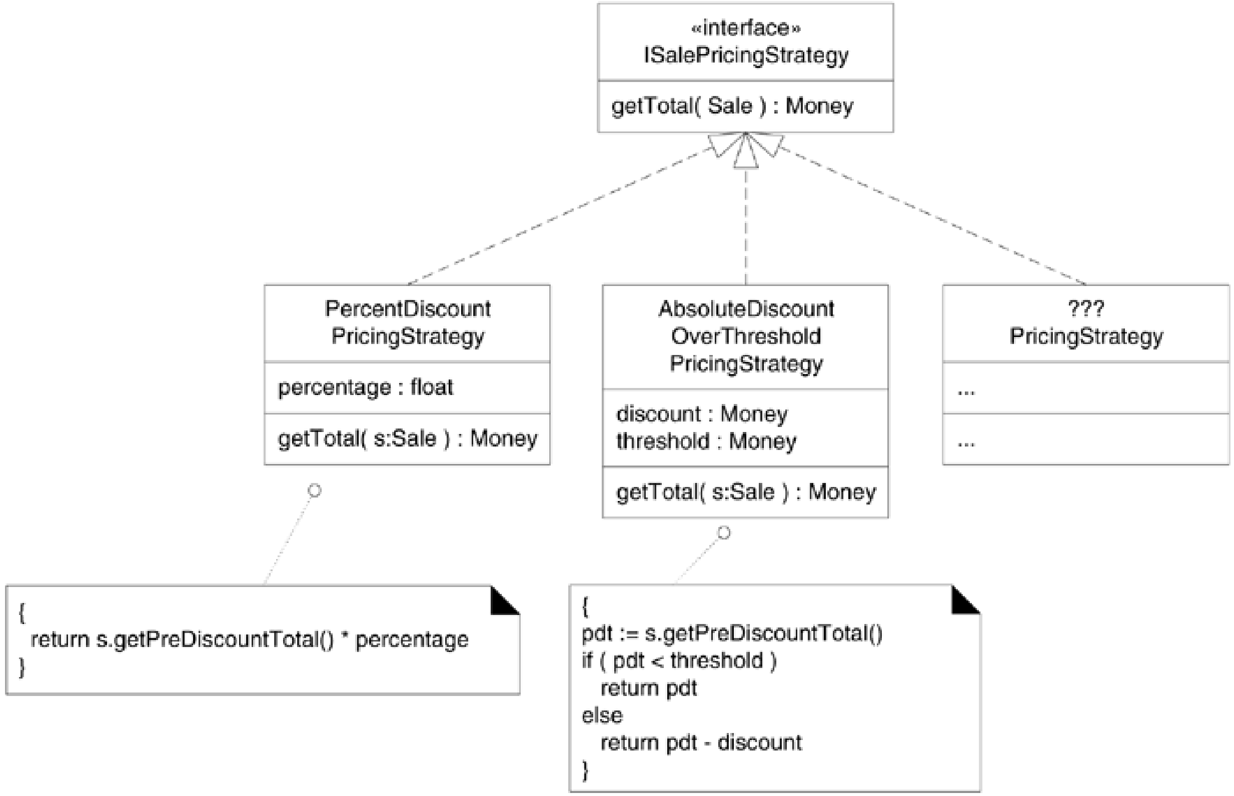
Behavior of pricing varies by strategy (or algorithm)

- Create multiple **SalePricingStrategy** classes
  - each with a polymorphic ***getTotal*** method

- Each ***getTotal*** takes **Sale** object as a parameter
  - Pricing strategy object can find the pre-discount price (total) from **Sale**

- Then apply the discounting rule
  - Implementation of each ***getTotal*** will be different **PercentDiscountPricingStrategy** will discount by a percentage, etc.

Pricing Strategy Classes



«interface»
ISalePricingStrategy

getTotal(Sale) : Money

PercentDiscount
PricingStrategy

percentage : float

getTotal(s:Sale) : Money

AbsoluteDiscount
OverThreshold
PricingStrategy

discount : Money
threshold : Money

getTotal(s:Sale) : Money

???
PricingStrategy

...

...

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

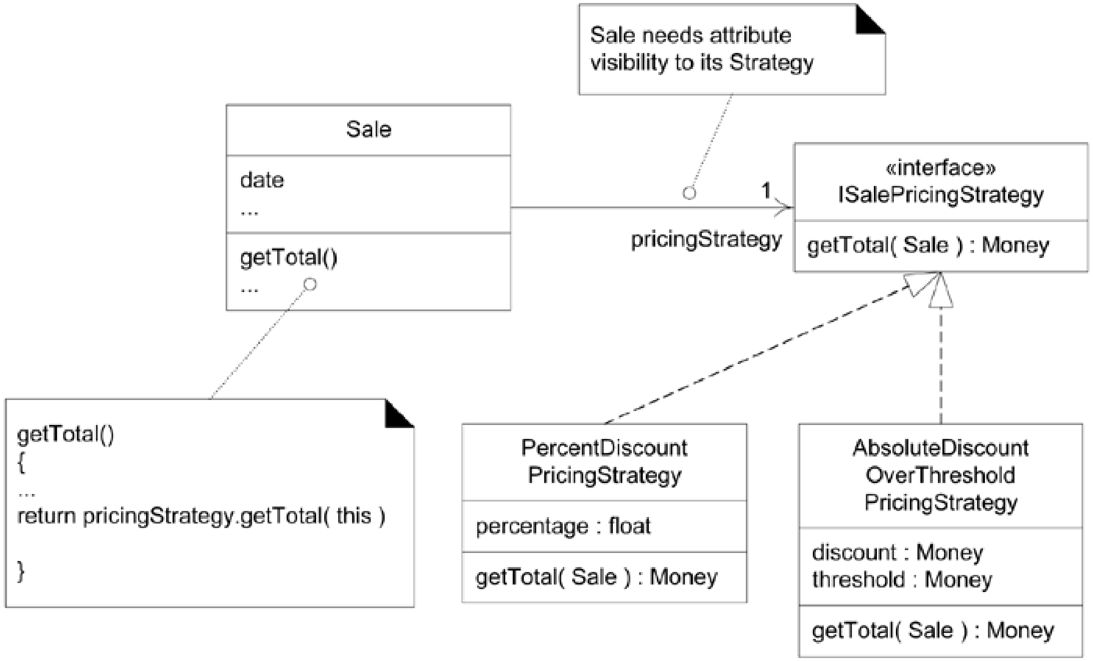
```
{
pdt := s.getPreDiscountTotal()
if ( pdt < threshold )
   return pdt
else
   return pdt - discount
}
```

Context Object Attached To Strategy Object

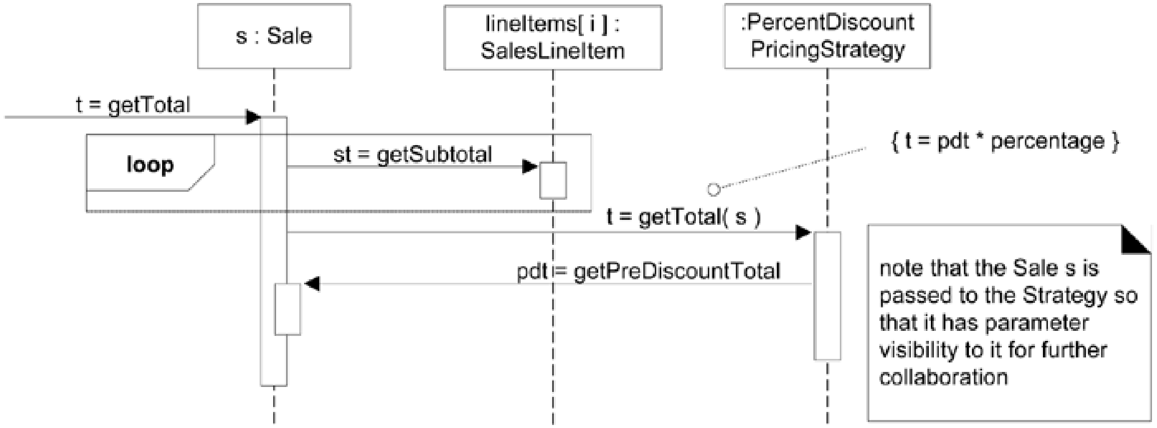
A context object—the object to which it applies the algorithm, **Sale**

- ***getTotal*** message is sent to a **Sale**
- **Sale** delegates some of the work to its strategy object
- Context object pass a reference of self (**this**) to strategy object
- Strategy has parameter visibility to Context object, for further collaboration

Context Object Needs Attribute Visibility To Its Strategy



Strategy In 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

Creating a Strategy Using Factory

Different pricing algorithms or strategies, change over time

Who should create the strategy? Separation of Concerns

PricingStrategyFactory creates all strategies, new factory was used for the strategies;

that is, different than the **ServicesFactory**

High Cohesion—each factory is cohesively focused on creating a related family of objects

Sale Associated with Interface

Sale has association to the interface **ISalePricingStrategy**, not to a **concrete** class

Sale declared in terms of the interface, not a class,
so any implementation of the interface can be **bound** (polymorphism)
to the attribute inside **Sale** (pricingStrategy)

When a **Sale** instance is created, it asks the **factory** for its pricing strategy

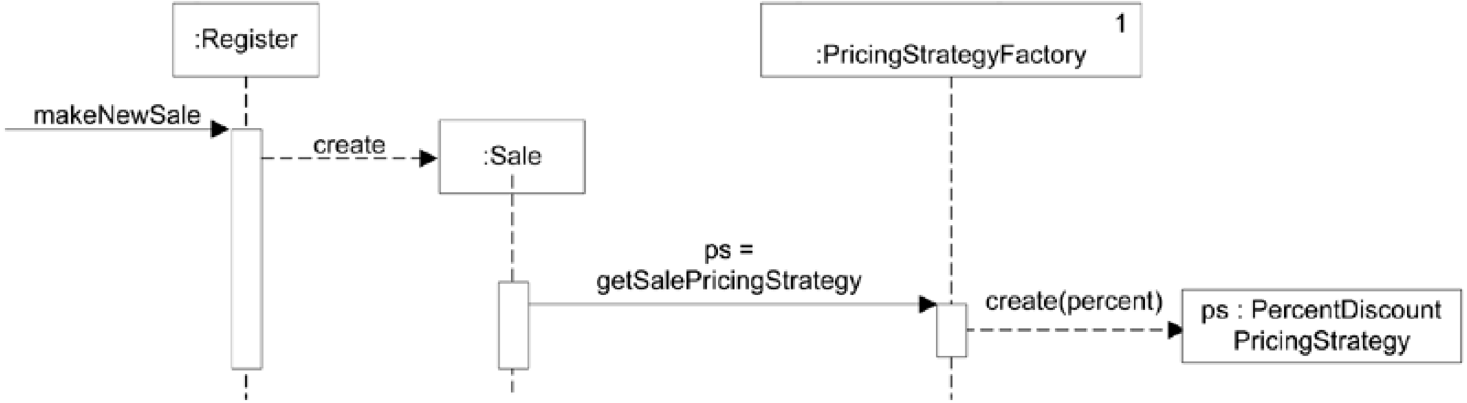
Creating a Strategy Using Factory

Due to frequently changing pricing policy (it could be every hour),

Not desirable to cache the created strategy instance in a field of the **PricingStrategyFactory**, but re-create one each time, by reading configurations

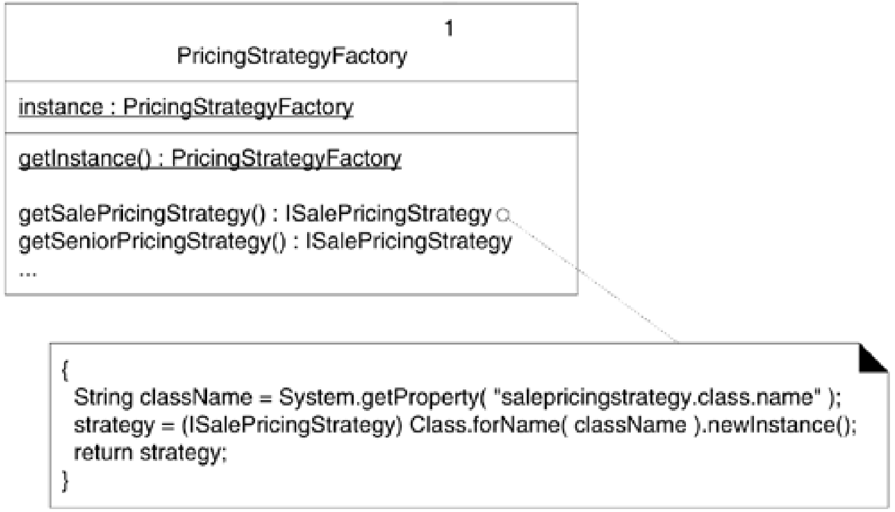
As with most factories, the **PricingStrategyFactory** will be a Singleton

Creating a Strategy Using Factory



Creating a strategy

Singleton Factory for strategies


1
PricingStrategyFactory
instance : PricingStrategyFactory
getInstance() : PricingStrategyFactory
getSalePricingStrategy() : ISalePricingStrategy
getSeniorPricingStrategy() : ISalePricingStrategy
...

```
{
  String className = System.getProperty( "salepricingstrategy.class.name" );
  strategy = (ISalePricingStrategy) Class.forName( className ).newInstance();
  return strategy;
}
```

Reading and Initializing the Percentage Value

How to find the different numbers for the percentage or absolute discounts

For example, on Monday, the **PercentageDiscountPricingStrategy** may have a percentage value of **10%**, but **20%** on Tuesday

A percentage discount may be related to the type of buyer, such as a senior citizen, rather than to a time period

Numbers stored in external **DB**,

what object will read them and ensure they are assigned to the strategy?

Reading and Initializing the Percentage Value

PricingStrategyFactory itself, it is creating the pricing strategy, and can know which percentage to read from a data store

(current store discount, senior discount, Mondays, etc)

Plain SQL call, **indirection**, data query language, or type of data store

Summary

Protected Variations with respect to dynamically changing pricing policies has been achieved

Using Strategy and Factory patterns

Strategy builds on Polymorphism and interfaces to allow **pluggable algorithms** in an object design

Composite

Composite

How to design for **Multiple, conflicting** pricing policies?

A store has the following policies in effect today (Monday):

- 20% senior discount policy
- Preferred customer discount of 15% off sales over $400
- There is $50 off purchases over $500
- Buy 1 case of Darjeeling tea, get 15% discount off of everything

A senior and a preferred customer buys 1 case of Darjeeling tea, and $600 of burgers. *What pricing policy should be applied?*

Pricing Strategies, Problem

Multiple co-existing strategies, one sale may have several pricing strategies

1. Related to time (e.g. Monday)
2. Related to type of customer (e.g. senior)
3. Related to type of product being bought (e.g., Darjeeling tea)

Type must be known by **StrategyFactory** at the time of creation of a pricing strategy

Is there a way to change the design so, **Sale** object **does not know** if it is dealing with

- **One** or **many pricing strategies** +
- Design for the **conflict resolution**

Composite!

Composite Pattern

Name	**Composite**
Problem	How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?
Solution	Define classes for composite and atomic objects so that they implement the same interface.

Pricing Strategies, Solution

Solution:

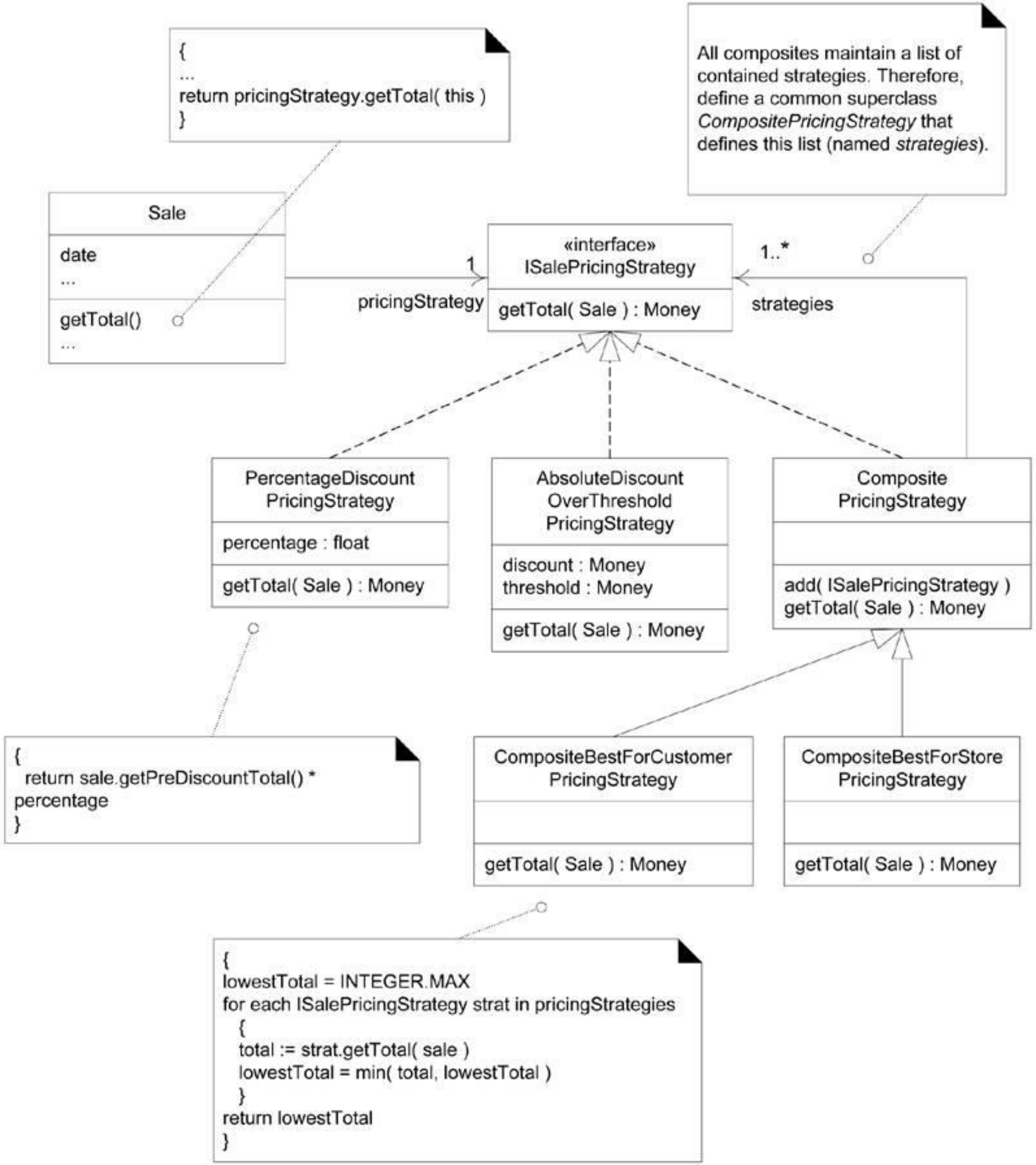
Store's conflict resolution strategy

- Best for the customer
- Highest price, best for store
- else?

CompositeBestForCustomerPricingStrategy can implement the **ISalesPricingStrategy**

and itself contain other **ISalesPricingStrategy** objects

The Composite Pattern



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

All composites maintain a list of contained strategies. Therefore, define a common superclass *CompositePricingStrategy* that defines this list (named *strategies*).

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

```
{
lowestTotal = INTEGER.MAX
for each ISalePricingStrategy strat in pricingStrategies
  {
  total := strat.getTotal( sale )
  lowestTotal = min( total, lowestTotal )
  }
return lowestTotal
}
```

As if it is Atomic!

Composite classes like **CompositeBestForCustomerPricingStrategy** inherit an attribute strategies that contains a list of more **ISalePricingStrategy** objects

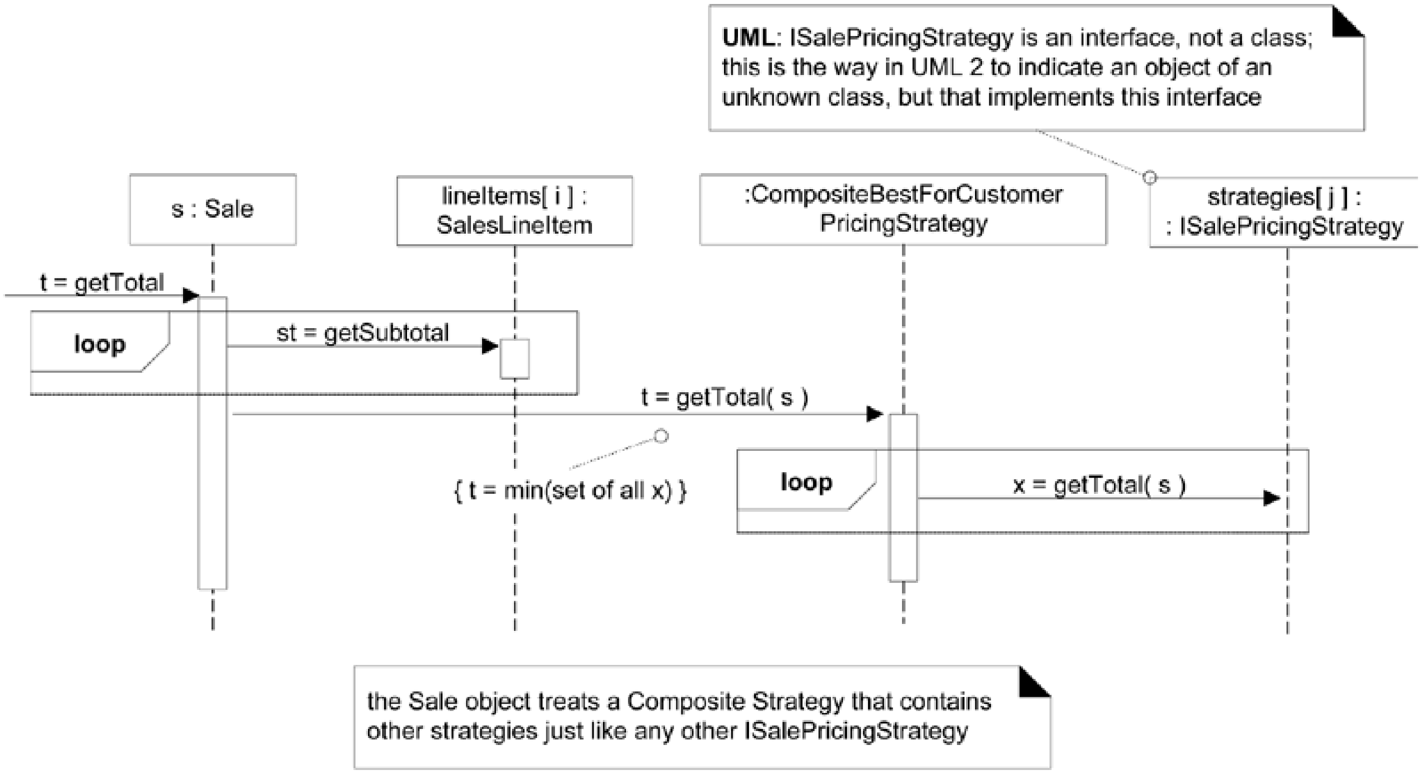
Outer composite object contains a list of inner objects, and both the outer and inner objects implement the same interface

Attach **CompositeBestForCustomerPricingStrategy** object or an atomic **PercentDiscountPricingStrategy** object to the **Sale** object

Sale does not know or care if its pricing strategy is **atomic or composite strategy**,

It is just an object that implements **ISalePricingStrategy** interface and understands the ***getTotal*** message, like a single pricing strategy

Collaboration with a Composite



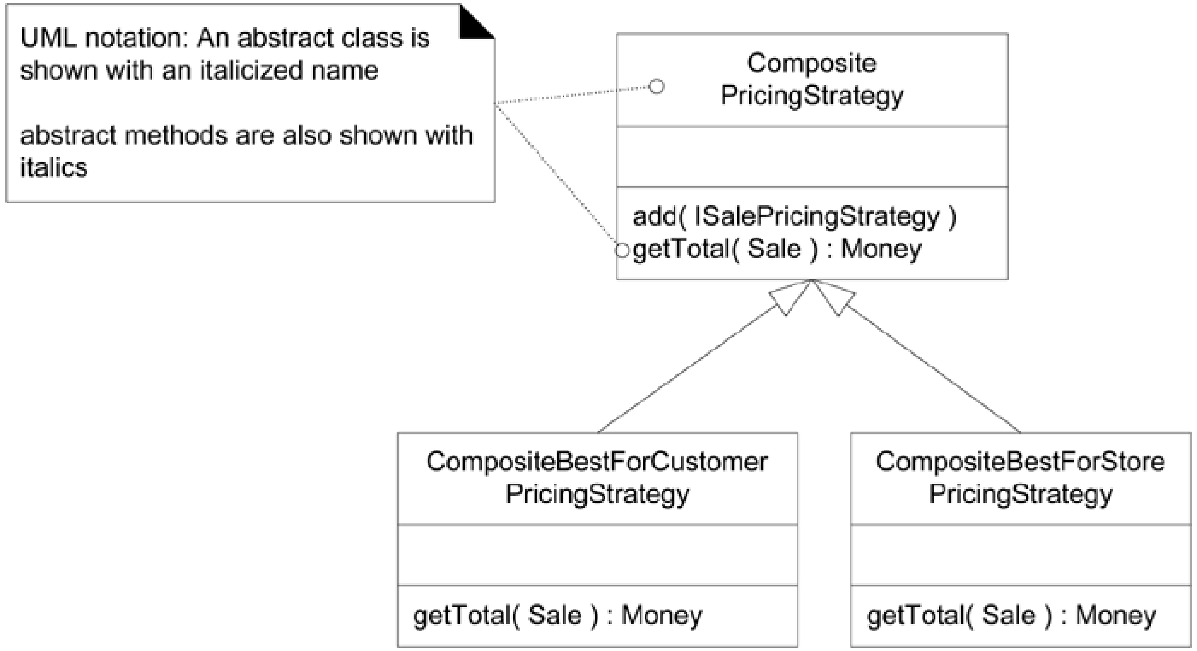
UML: ISalePricingStrategy is an interface, not a class; this is the way in UML 2 to indicate an object of an unknown class, but that implements this interface

the Sale object treats a Composite Strategy that contains other strategies just like any other ISalePricingStrategy

```
// superclass so all subclasses can inherit a List of strategies
public abstract class CompositePricingStrategy
   implements ISalePricingStrategy
{
   protected List strategies = new ArrayList();
   public add( ISalePricingStrategy s )
   {
      strategies.add( s );
   }
   public abstract Money getTotal( Sale sale );
} // end of class
```

```
// a Composite Strategy that returns the lowest total of its inner SalePricingStrategies
public class CompositeBestForCustomerPricingStrategy
   extends CompositePricingStrategy
{
    public Money getTotal( Sale sale )
    {
       Money lowestTotal = new Money( Integer.MAX_VALUE );
       // iterate over all the inner strategies
       for( Iterator i = strategies.iterator(); i.hasNext(); )
       {
          ISalePricingStrategy strategy =
             (ISalePricingStrategy)i.next();
          Money total = strategy.getTotal( sale );
          lowestTotal = total.min( lowestTotal );
       }
       return lowestTotal;
    }
} // end of class
```

Abstract Superclasses, Abstract Methods, And Inheritance In The UML



UML notation: An abstract class is shown with an italicized name

abstract methods are also shown with italics

Composite PricingStrategy

add(ISalePricingStrategy)
getTotal(Sale) : Money

CompositeBestForCustomer PricingStrategy

getTotal(Sale) : Money

CompositeBestForStore PricingStrategy

getTotal(Sale) : Money

When Do We Create These Strategies?

Create default, 0%, 10%, etc.
PercentageDiscountPricingStrategy

Then, if at a later step in the scenario, another pricing strategy is discovered (e.g. senior discount),

Add it to the composite, using the inherited **CompositePricingStrategy**.***add*** method

Scenarios

Three pricing strategies may be added to the composite:

1. Current store-defined discount, added when the sale is created

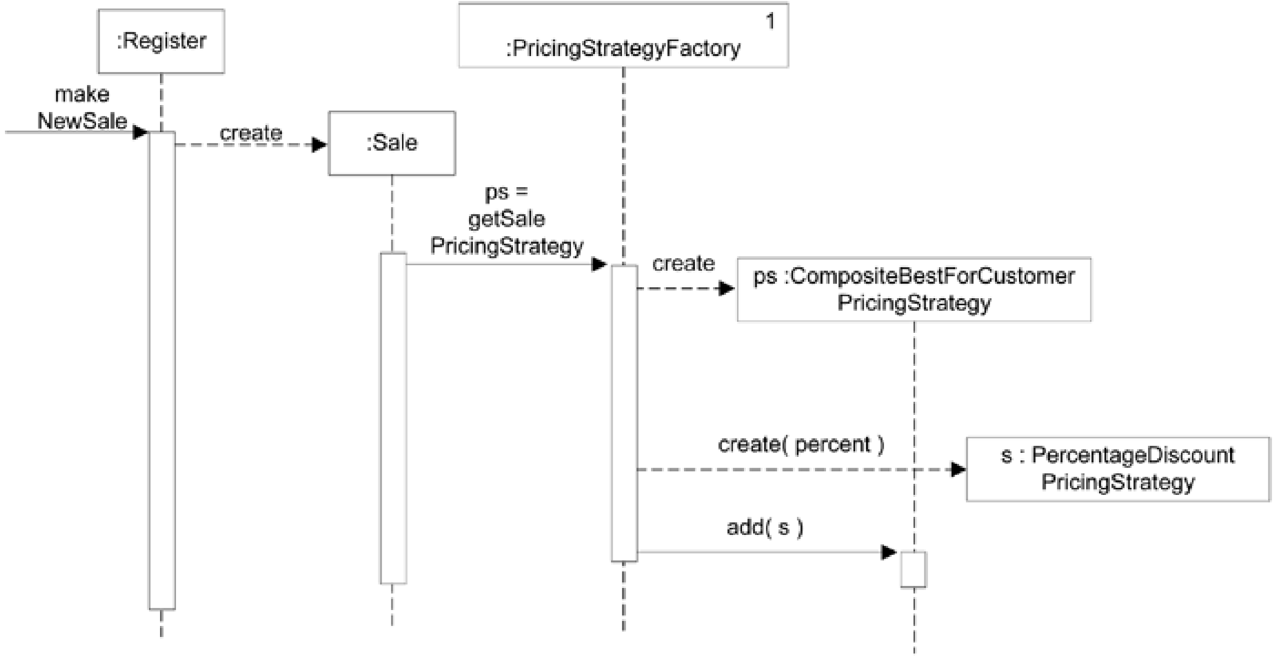
2. Customer type discount, added when the customer type is communicated to POS

3. Product type discount, added when the product is entered to the sale

First Case—Current Store-defined Discount

- Strategy class name to instantiate could be read as a system property (confg.),

- And a percentage value could be read from an external data store



:Register
:PricingStrategyFactory
1
make NewSale
create
:Sale
ps = getSale PricingStrategy
create
ps :CompositeBestForCustomer PricingStrategy
create(percent)
s : PercentageDiscount PricingStrategy
add(s)

Second Case—Customer type discount

Use Case UC1: Process Sale

… **Extensions (or Alternative Flows):**

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer)

1. Cashier signals discount request.

2. Cashier enters Customer identification.

3. System presents discount total, based on discount rules.

enterCustomerForDiscount

A new system operation on the POS system, ***makeNewSale***, ***enterItem***, ***endSale***, and ***makePayment***.

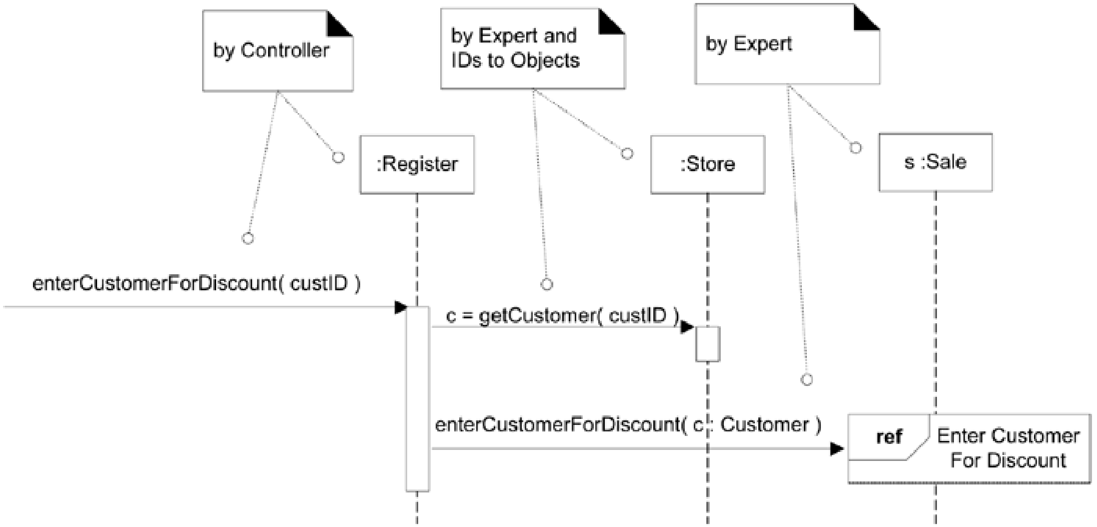
enterCustomerForDiscount

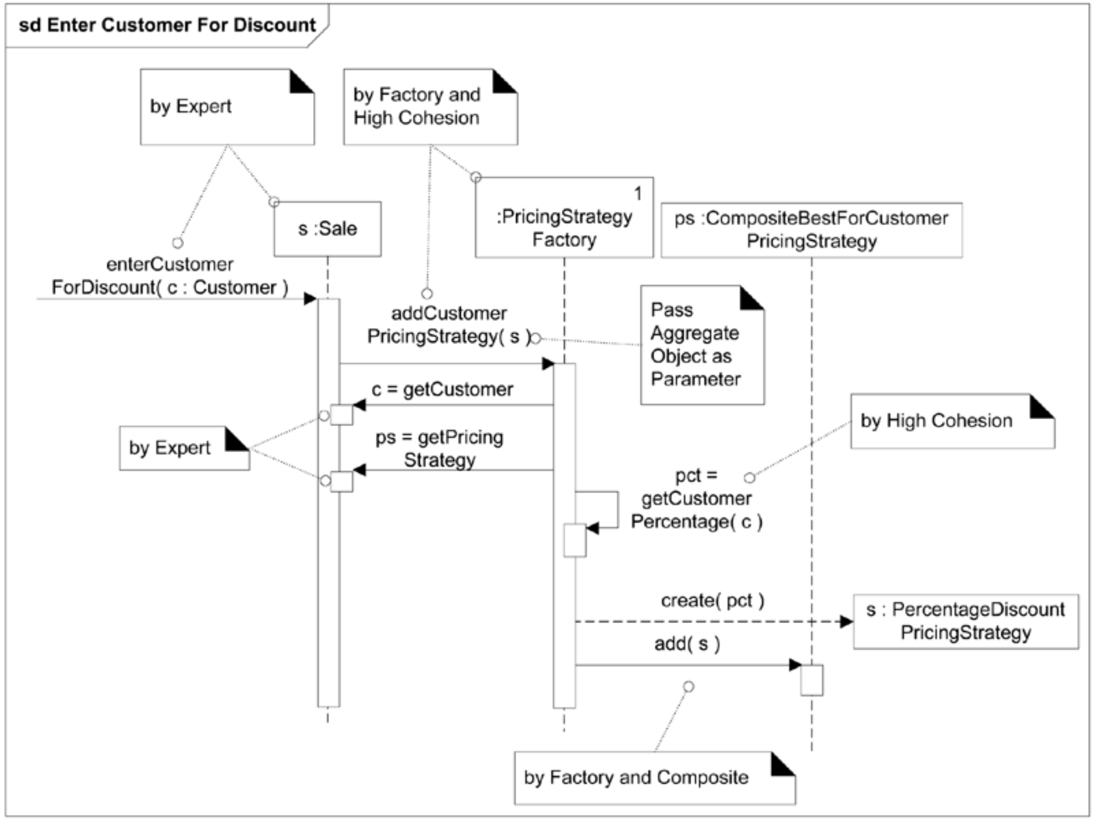
Optionally occur after the *endSale* operation

Customer identification have to come in through the user interface, the customerID

- Captured from a card reader, or via the keyboard

Creating The Pricing Strategy For A Customer Discount


by Controller
by Expert and IDs to Objects
by Expert
:Register
:Store
s :Sale
enterCustomerForDiscount(custID)
c = getCustomer(custID)
enterCustomerForDiscount(c : Customer)
ref
Enter Customer For Discount


sd Enter Customer For Discount
by Expert
by Factory and High Cohesion
s :Sale
1
:PricingStrategy Factory
ps :CompositeBestForCustomer PricingStrategy
enterCustomer ForDiscount(c : Customer)
addCustomer PricingStrategy(s)
Pass Aggregate Object as Parameter
c = getCustomer
by Expert
ps = getPricing Strategy
by High Cohesion
pct = getCustomer Percentage(c)
create(pct)
s : PercentageDiscount PricingStrategy
add(s)
by Factory and Composite

Second Case Design

Factory object create additional pricing strategy

- e.g. Another **PercentageDiscountPricingStrategy** represents a senior discount

Choice of class will be pulled from confg. And the percentage for the customer type from a data store

Solution provides Protected Variations with respect to changing the class or values

Sale may have many conflicting pricing strategies attached to it, but it continues to look like a single strategy to the **Sale** object

GRASP + ODD Principles

GRASP Creator and Information Expert

Why not **Register** send a message to **PricingStrategyFactory**, to create new pricing strategy and then pass it to **Sale**?

Low Coupling! Why?

- **Sale** is already coupled to the **factory**
- **Sale** is the Expert
  - Knows its current pricing strategy, so **Register** delegates to **Sale**

GRASP Creator and Information Expert

customerID is transformed into a **Customer** object via the **Register** asking the **Store** for a **Customer**, given an ID

getCustomer responsibility of **Store**

- By Information Expert and LRG, **Store** can know all the **Customers**

Register asks **Store**, **Register** already has attribute visibility to **Store**

if **Sale** had to ask **Store**, **Sale** would need a reference to the **Store,** that increases coupling!

IDs to Objects

Why transform the customerID (ID—perhaps a number) into a Customer object?

Common practice in OOD—to transform keys and IDs for things into true objects, takes place shortly after an ID or key enters the domain layer of the Design Model from the UI layer

Not a pattern, but a candidate—perhaps ***IDs to Objects***

IDs to Objects, WHY?

A true **Customer** object encapsulates a set of information about the customer

Can have behavior (being Expert)

Beneficial and flexible as the design grows

Designers may not feel a need for a true object (plain number) is sufficient

e.g. Transformation of the itemID into a **ProductDescription** object is another example of this ***IDs to Objects*** pattern

Pass Aggregate Object as Parameter

addCustomerPricingStrategy(**<u>s</u>**:Sale) message passed a **Sale** to the factory, factory turns around and asks for **Customer** and **PricingStrategy** from **Sale**

*Why not extract these two objects from the **Sale**, and instead pass in the **Customer** and **PricingStrategy** to the factory?*

Pass Aggregate Object as Parameter, WHY?

Answer: Avoid extracting child objects out of parent or aggregate objects, and then passing around the child objects. **Pass Aggregate Object**

Increases **flexibility**, factory can collaborate with entire **Sale** in ways we may not have previously anticipated as necessary (is very common)

Supports Low Coupling and Protected Variations, ***Pass Aggregate Object as Parameter*** is a candidate pattern.

Summary

Composite was applied to a Strategy family

Composite pattern can be applied to other kinds of objects

Façade

Facade

Pluggable business rules at predictable points in the scenarios

e.g. when ***makeNewSale*** or ***enterItem*** occurs in the Process Sale use case different customers who wish to purchase the NextGen POS would like to customize its behavior slightly

Example, Problem

Rules are desired that invalidate an action:

- New sale is created, will be paid by a gift certificate. Then, a store may have a rule to only allow one item to be purchased, subsequent ***enterItem*** should be **invalidated**
- If sale is paid by a gift certificate, if cashier requested change in the form of cash, or as a credit to the customer's store account, **invalidate** those requests
- A new sale is a charitable donation (from the store to the charity). A store may also have a rule to only allow item entries less than $250 each, and also to only add items to the sale if the currently logged in "cashier" is a manager. *What does that even mean?*

Solution

Software Architect design with low impact on the existing software components

Wants to factor out this rule handling into a separate concern

Architect is unsure of the best implementation for this pluggable rule handling, for example:

- Rules can be implemented with the Strategy pattern
- Free open-source rule interpreters that read and interpret a set of IF-THEN rules
- Or with commercial, purchased rule interpreters, among other solutions

Façade!

Façade Pattern

Name	**Façade**
Problem	A common, unified interface to a disparate set of implementations or interfaces—such as within a subsystem—is required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What to do?
Solution	Define a **single point of contact** to the **subsystem**— a facade object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.

Single Point of Entry

A Facade is a **“front-end”** object that is the single point of entry for the services of a subsystem

Implementation and other components of the subsystem are **private** and can't be seen by external components

Facade provides **Protected Variations** from changes in the implementation of a subsystem

A “Rule Engine” subsystem, whose specific implementation is not yet known

POSRuleEngineFacade

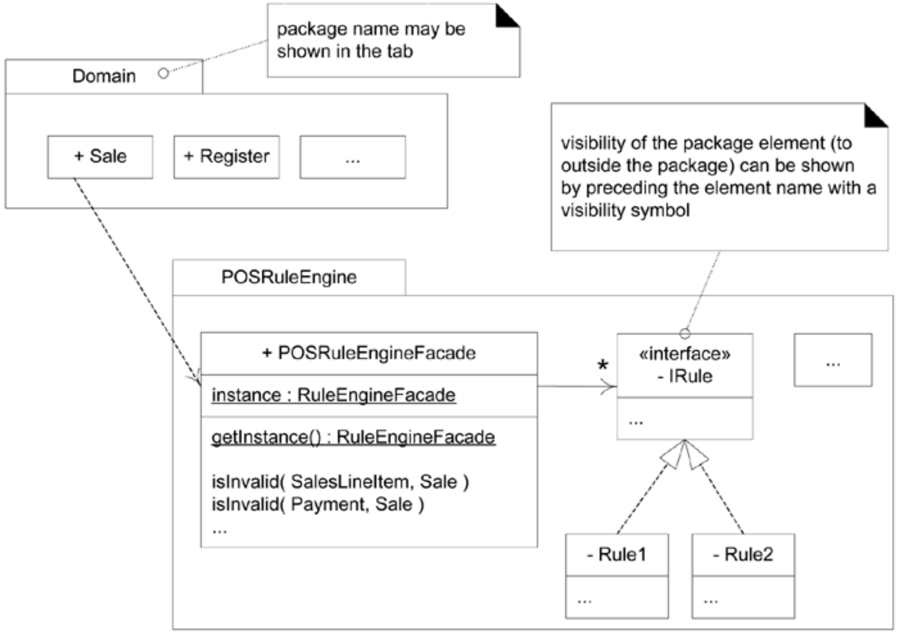
Facade object to this subsystem will be called **POSRuleEngineFacade**

Designer decides to place calls to this facade near the start of the methods that have been defined as the points for pluggable rules

Implementation

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

UML Package Diagram With A Facade


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
...

Subsystem Hidden

Note the use of the Singleton pattern. Facades are often accessed via Singleton

Subsystem hidden by the facade object could contain **dozens or hundreds of classes of objects, or even a non-object-oriented solution, yet as a client to the subsystem, we see only its one public access point**

And a separation of concerns has been achieved to some degree—all the rule-handling concerns have been delegated to another subsystem

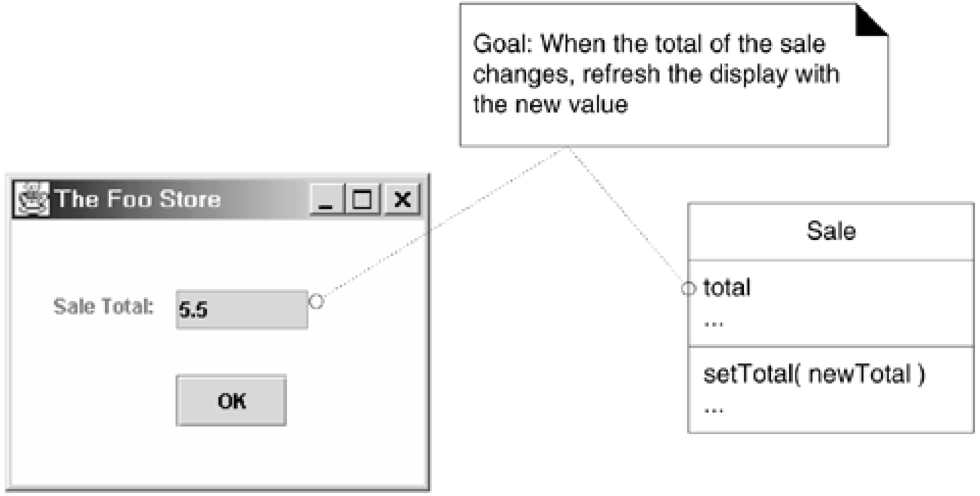
Subsystem Hidden

The Facade Pattern Is Simple And Widely Used. It Hides A Subsystem Behind An Object!

Observer/ Publish-Subscribe/ Delegation Event Model

Observer, Problem?

Adding the ability for a GUI window to refresh its display of the **sale total** when the total changes



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

Observer, Solution?

Sale changes its total, **Sale** object sends a message to a window, asking it to refresh its display. *What is Wrong with that?*

Model-View Separation Principle: Model objects (non-UI) should not know about view or presentation objects like window, **Protected Variations** with respect to a changing user interface

Why? Such low coupling allows replacement of the view or a window, without impacting the non-UI objects

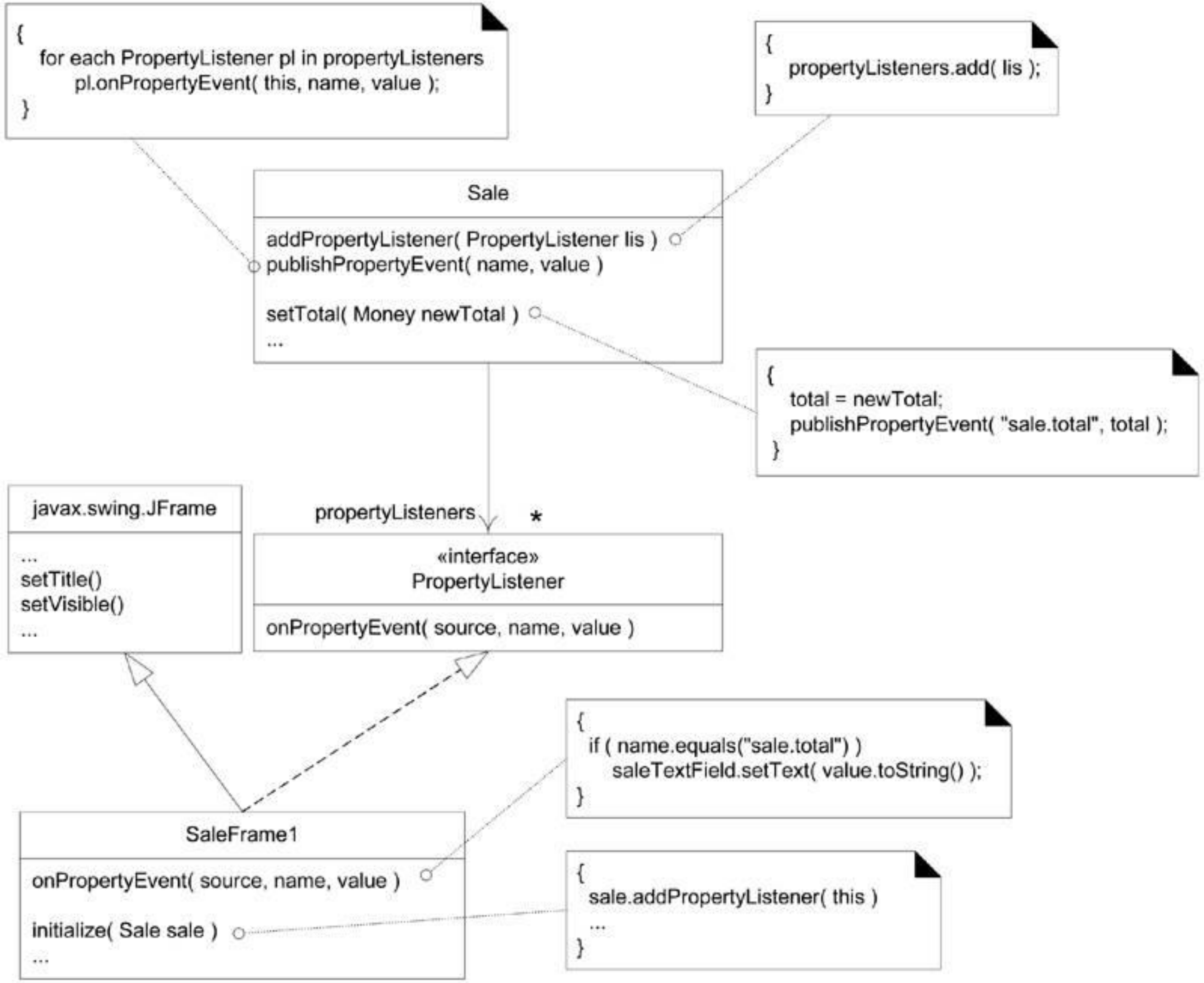
If model objects do not know about Java Swing objects (for example), then it is possible to unplug a Swing interface, or unplug a particular window, and plug in something else.

Observer!

Observer Pattern

Name	**Observer (Publish-Subscribe)**
Problem	Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?
Solution	Define a "subscriber" or "listener" **interface**. 1. Subscribers implement this interface. 2. The **publisher can dynamically register subscribers** who are interested in an event and **3. Notify** them when an event occurs

Observer Pattern



```
{
  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 );
}
```

propertyListeners *

javax.swing.JFrame

...
setTitle()
setVisible()
...

«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 )
 ...
}
```

Sale as a Publisher

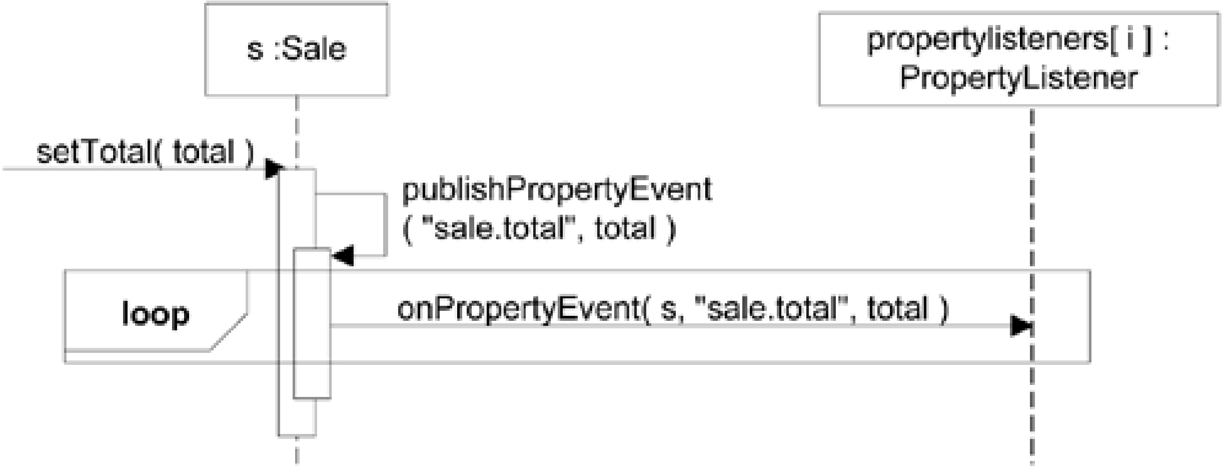
PropertyListener interface with operation ***onPropertyEvent***
(window) **SaleFrame1** will implement the method ***onPropertyEvent***

SaleFrame1 gets **Sale** to display total

SaleFrame1 registers/subscribes to **Sale** instance for notification of "property events," propertyListener,
via the ***addPropertyListener*** message

Total changes, window wants to be notified

The Sale Publishes A Property Event To All Its Subscribers



SaleFrame1 as a Subscriber

Sale does not know about **SaleFrame1** objects; it knows objects implement **PropertyListener**

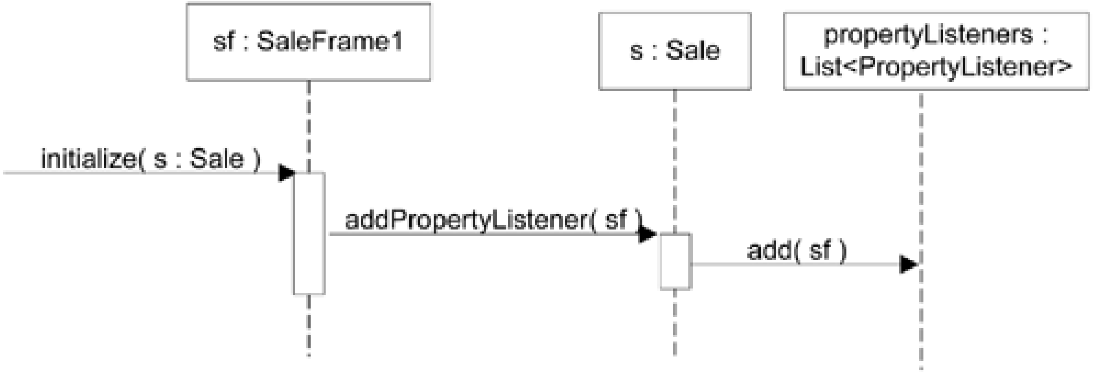
- Lowers coupling, **Sale** to window—Coupling is only to an interface, and not to a GUI class

Sale instance is a publisher of “property events.”

When the total changes, it iterates across all subscribing **PropertyListeners**, notifying each

SaleFrame1 object is the observer/subscriber/listener

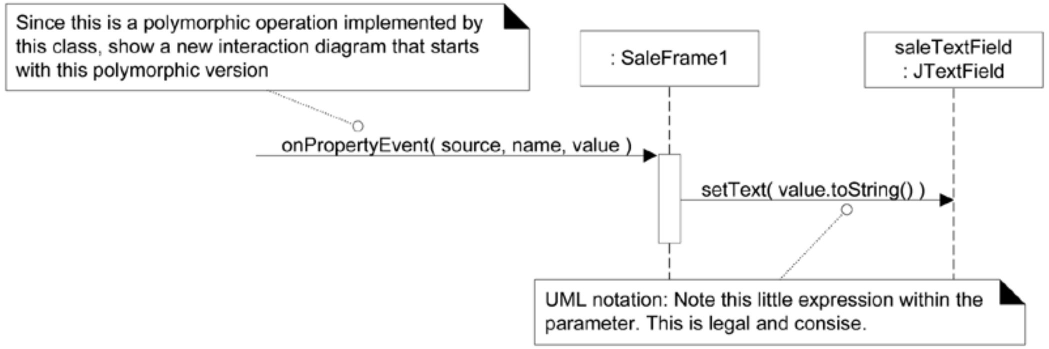
The Observer Saleframe1 Subscribes To The Publisher Sale



The Subscriber Saleframe1 Receives Notification Of A Published Event

SaleFrame1, implements **PropertyListener** interface, thus implements
an ***onPropertyEvent*** method

SaleFrame1 receives the message, it sends a message to its GUI textbox object to refresh with the new sale total



: SaleFrame1
saleTextField
: JTextField
onPropertyEvent(source, name, value)
setText(value.toString())

Since this is a polymorphic operation implemented by this class, show a new interaction diagram that starts with this polymorphic version

UML notation: Note this little expression within the parameter. This is legal and consise.

Model-View Separation Principle

Coupling from model object (**Sale**) to the view object (**SaleFrame1**)??!!!

A **loose coupling** to an interface independent of the presentation layer—the **PropertyListener** interface

Design does not require any subscriber objects to actually be registered with the publisher (no objects have to be listening)

List of registered **PropertyListeners Sale** can be **empty!**

Model-View Separation Principle

Coupling to a generic interface of objects that do not need to be present,

and which can be dynamically added (or removed), supports low coupling

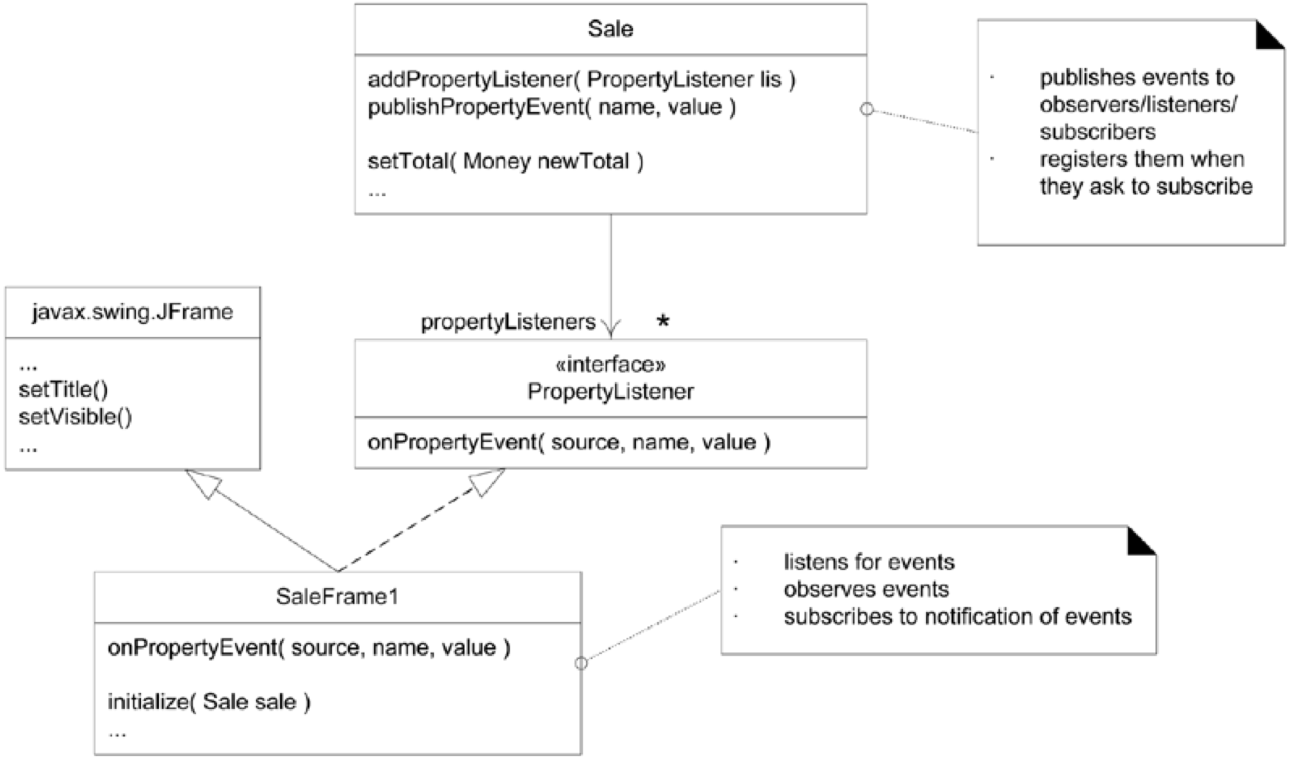
Protected Variations with respect to a changing user interface has been achieved via interface and polymorphism

Why Called Observer, Publish-Subscribe, or Delegation Event Model?

It has been called Observer because the listener or subscriber is observing the event; that term was popularized in Smalltalk in the early 1980s.

It has also been called the Delegation Event Model because the publisher delegates handling of events to “listeners”

Who Is The Observer, Listener, Subscriber, And Publisher?


Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
publishes events to observers/listeners/ subscribers
registers them when they ask to subscribe
propertyListeners
*
javax.swing.JFrame
...
setTitle()
setVisible()
...
«interface»
PropertyListener
onPropertyEvent(source, name, value)
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
listens for events
observes events
subscribes to notification of events

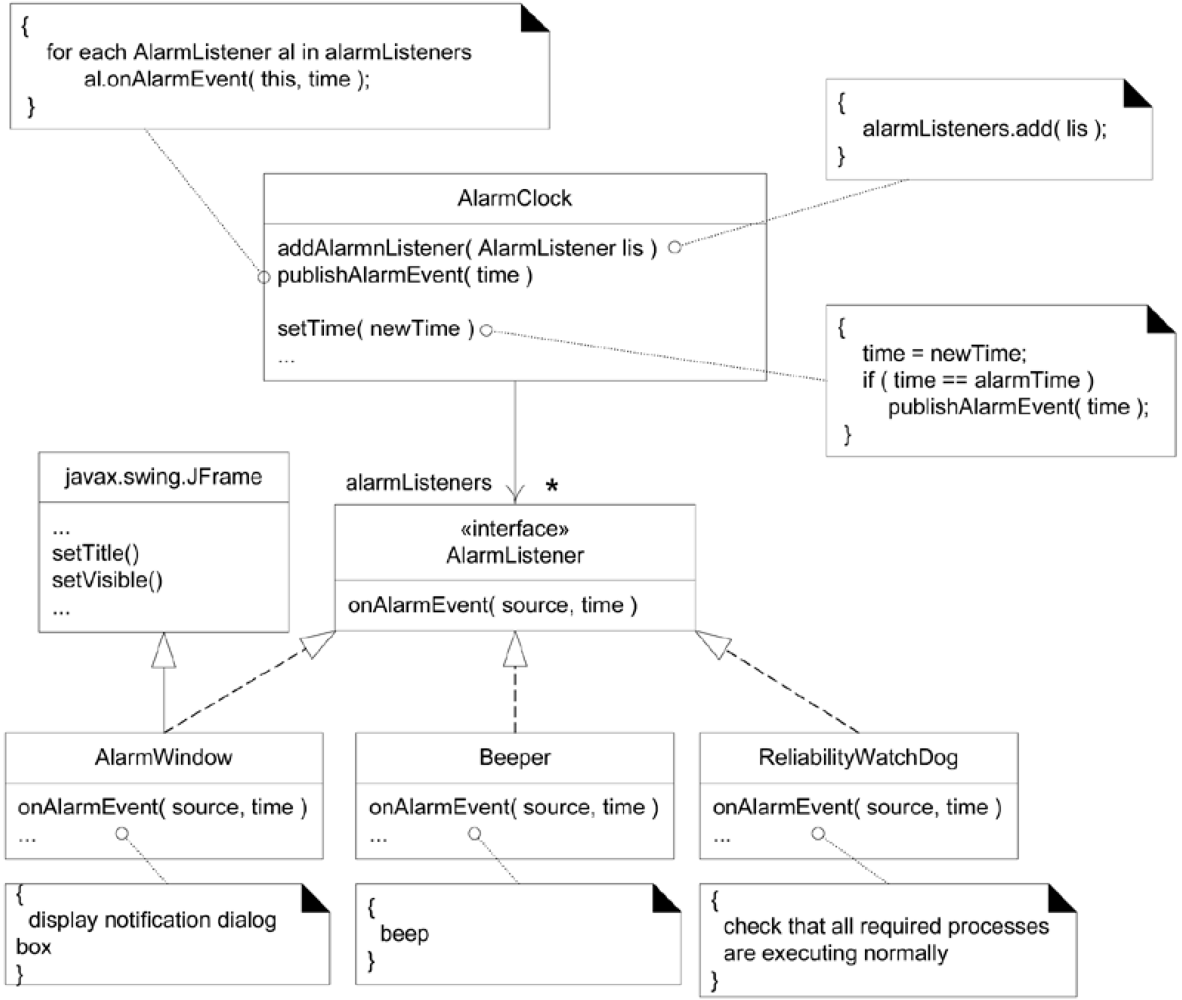
Observer Is Not Only for Connecting UIs and Model Objects

A **Button** publishes an “action event” when it is pressed. Another object will register with the button so that when it is pressed, the object is sent a message and can take some action

e.g. **AlarmClock** A publisher of alarm events and various subscribers

- Many classes can implement **AlarmListener** interface, many objects registered as listeners, and all react to “*alarm event*” in their own unique way

Observer Applied To Alarm Events, With Different Subscribers



```
{
   for each AlarmListener al in alarmListeners
       al.onAlarmEvent( this, time );
}
```

```
{
   alarmListeners.add( lis );
}
```

```
{
   time = newTime;
   if ( time == alarmTime )
       publishAlarmEvent( time );
}
```

```
{
  display notification dialog
box
}
```

```
{
  beep
}
```

```
{
  check that all required processes
  are executing normally
}
```

AlarmClock
addAlarmnListener(AlarmListener lis)
publishAlarmEvent(time)
setTime(newTime)
...
javax.swing.JFrame
...
setTitle()
setVisible()
...
alarmListeners
*
«interface»
AlarmListener
onAlarmEvent(source, time)
AlarmWindow
onAlarmEvent(source, time)
...
Beeper
onAlarmEvent(source, time)
...
ReliabilityWatchDog
onAlarmEvent(source, time)
...

One Publisher Can Have Many Subscribers for an Event

One publisher instance could have from zero to many registered subscribers

One instance of an **AlarmClock** could have three registered **AlarmWindows**, four **Beepers**, and one **ReliabilityWatchDog**

When an alarm event happens, all eight of these **AlarmListeners** are notified via an ***onAlarmEvent***

Summary

Observer provides a way to **loosely couple** objects in terms of communication

Publishers know about subscribers only through an **interface**

And subscribers can register (or de-register) dynamically with the publisher

S.O.L.I.D

Abdulkareem Alali

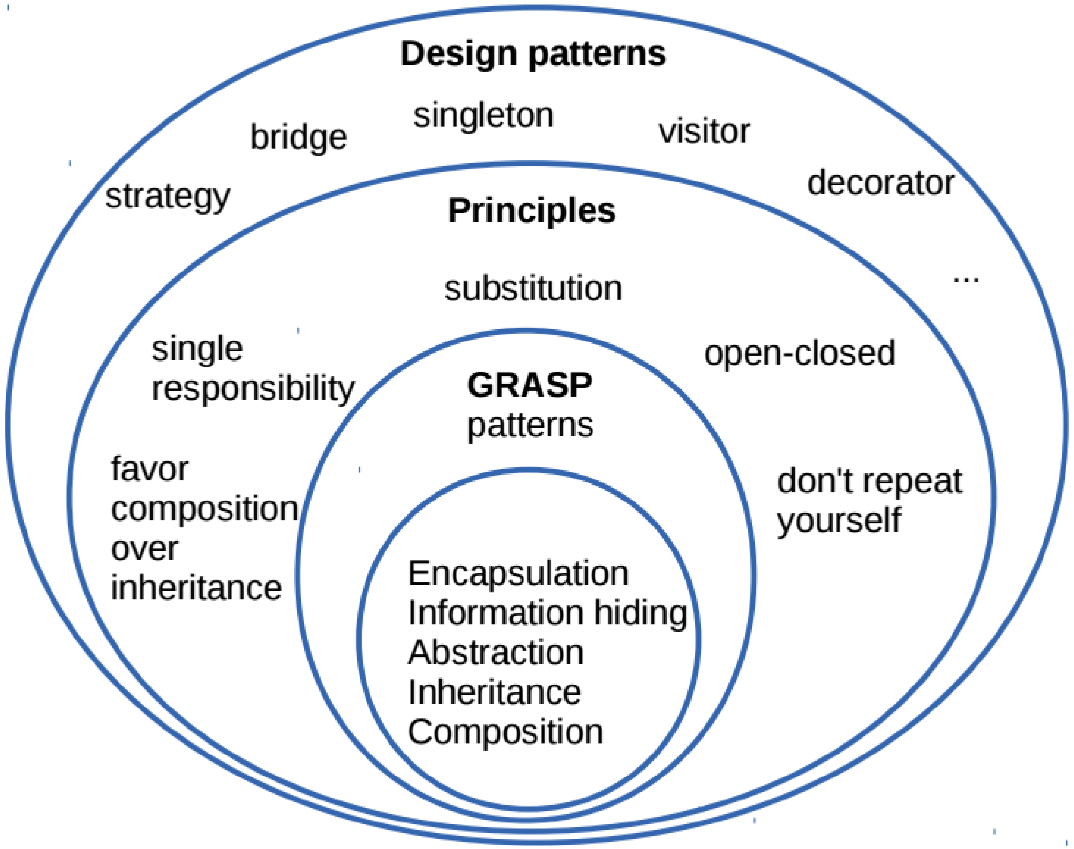
AKA Matthew P Jones

How to Design Object-Oriented?

There's **no methodology** to get the best object-oriented design,

but there are

Principles, Patterns, Best Practices, and **heuristics**.



S.O.L.I.D (mnemonic)

1. **S**ingle Responsibility Principle
2. **O**pen Closed Principle
3. **L**iskov Substitution Principle
4. **I**nterface Segregation Principle
5. **D**ependency Inversion Principle

SOLID

Concepts were introduced by Robert C. Martin in the early 2000s which stands for five basic patterns/principles of object-oriented programming and design

Principles when applied together make it much more likely that a programmer will

Create A System That Is Easy To Maintain And Extend Over Time

Single Responsibility Principle (SRP)

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should
WENGER

Single Responsibility Principle

"There should never be more than one reason for a class to change." *— Robert Martin, SRP paper linked from [The Principles of OOD]*

A class should concentrate on doing **one** thing and one thing only

The Single Responsibility Principle

Any class must have one, and only one, reason to change

If a class has more than one reason to change, it should be **refactored**

A change in a class having more responsibilities, the change might affect the

other functionality of the classes

SRP Benefits

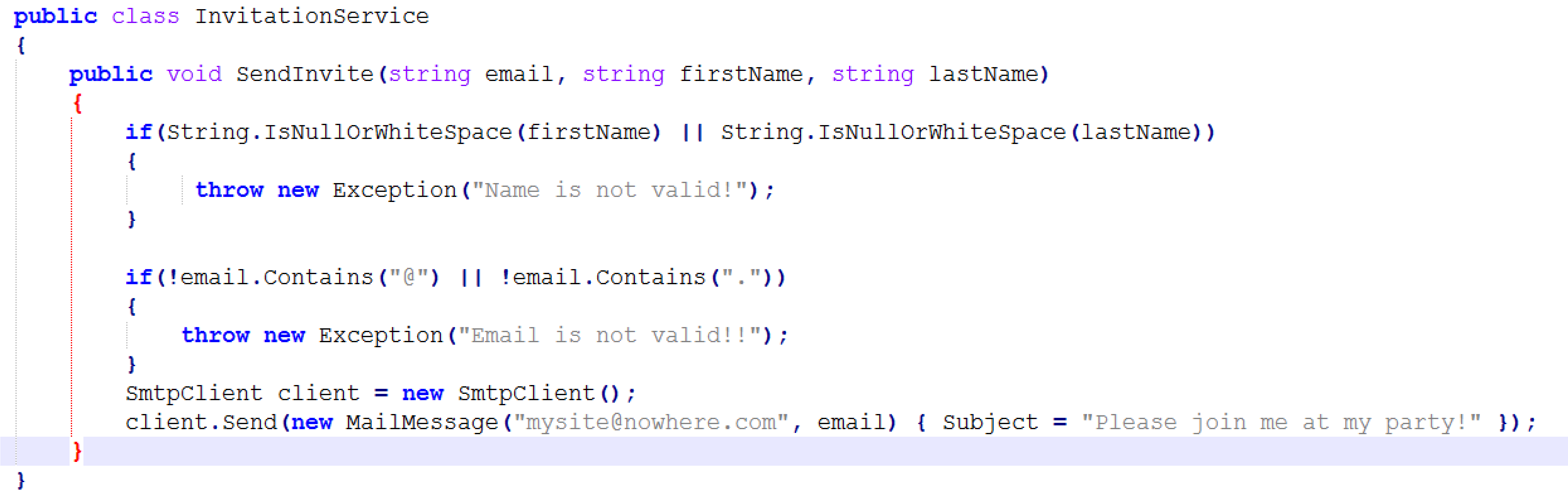
High-cohesion, Low-coupling code

When changes arise, **impact** is minimized

Minimizes possible times **more** than **one class**
will have to change for a given requirement or issue

Maximizes the possibility that changing **one** class
will **not** impact any other classes

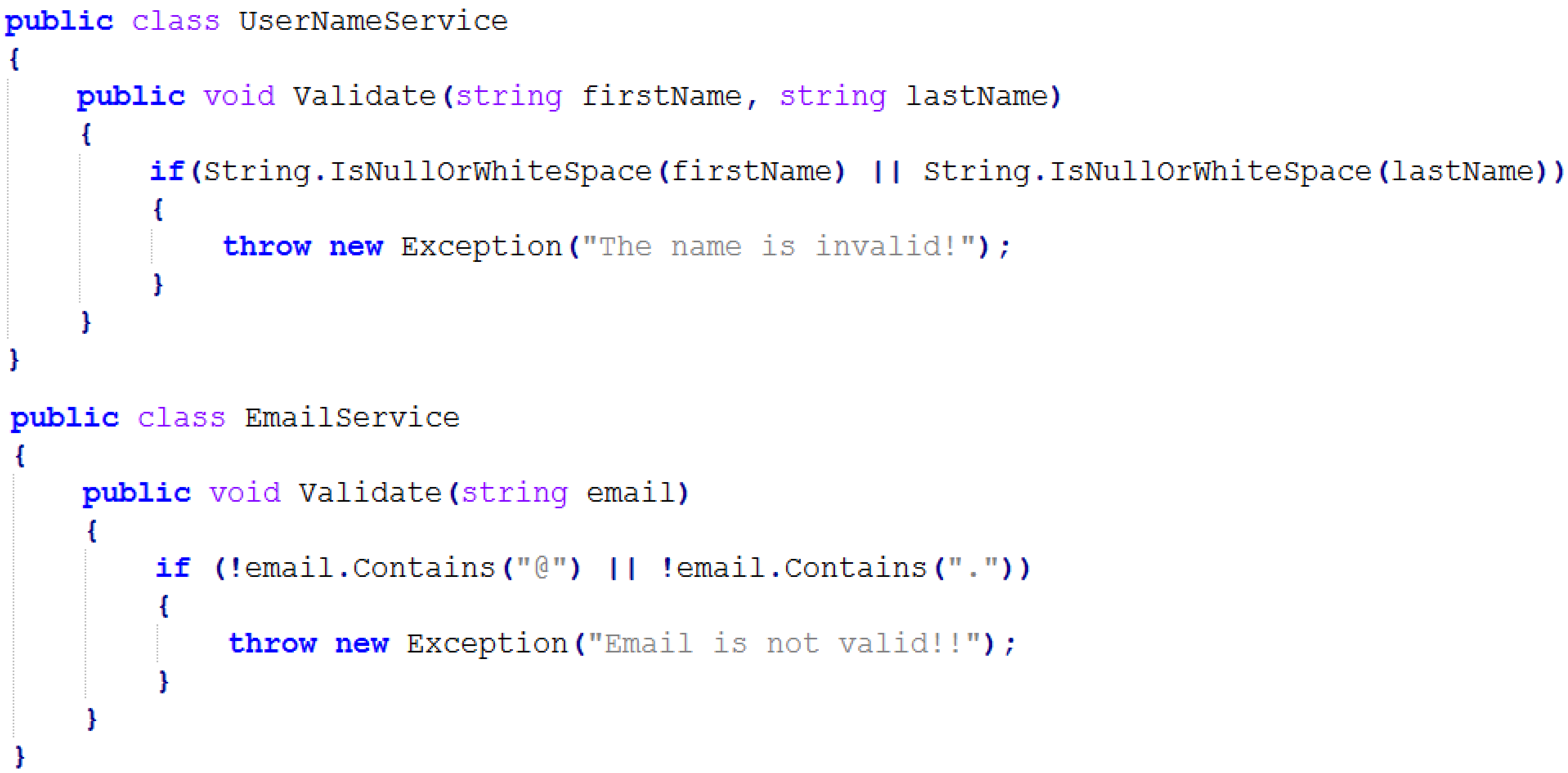
SRP Example



```
public class InvitationService
{
    public void SendInvite(string email, string firstName, string lastName)
    {
        if(String.IsNullOrWhiteSpace(firstName) || String.IsNullOrWhiteSpace(lastName))
        {
             throw new Exception("Name is not valid!");
        }

        if(!email.Contains("@") || !email.Contains("."))
        {
            throw new Exception("Email is not valid!!");
        }
        SmtpClient client = new SmtpClient();
        client.Send(new MailMessage("mysite@nowhere.com", email) { Subject = "Please join me at my party!" });
    }
}
```

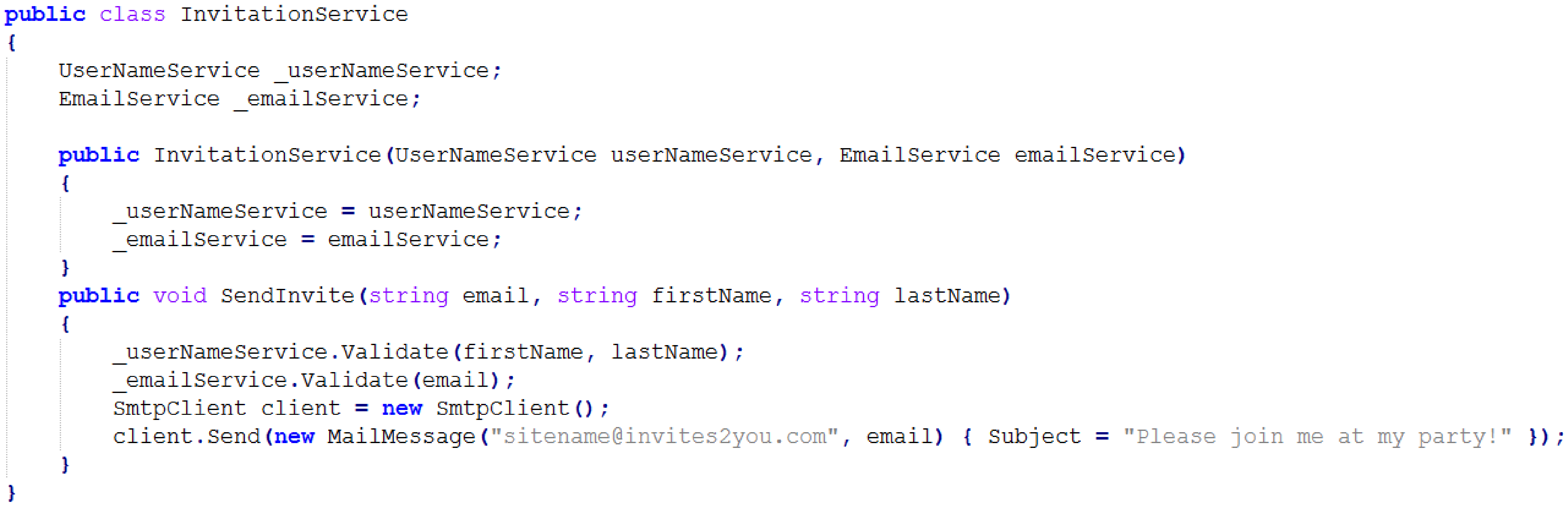
SRP Applied—Refactored



```
public class UserNameService
{
    public void Validate(string firstName, string lastName)
    {
        if(String.IsNullOrWhiteSpace(firstName) || String.IsNullOrWhiteSpace(lastName))
        {
            throw new Exception("The name is invalid!");
        }
    }
}

public class EmailService
{
    public void Validate(string email)
    {
        if (!email.Contains("@") || !email.Contains("."))
        {
            throw new Exception("Email is not valid!!");
        }
    }
}
```

SRP Applied—Refactored



```
public class InvitationService
{
    UserNameService _userNameService;
    EmailService _emailService;

    public InvitationService(UserNameService userNameService, EmailService emailService)
    {
        _userNameService = userNameService;
        _emailService = emailService;
    }
    public void SendInvite(string email, string firstName, string lastName)
    {
        _userNameService.Validate(firstName, lastName);
        _emailService.Validate(email);
        SmtpClient client = new SmtpClient();
        client.Send(new MailMessage("sitename@invites2you.com", email) { Subject = "Please join me at my party!" });
    }
}
```

Now each class will have one, and only one, reason to change

Example 2

RectangleShape class implements two methods:

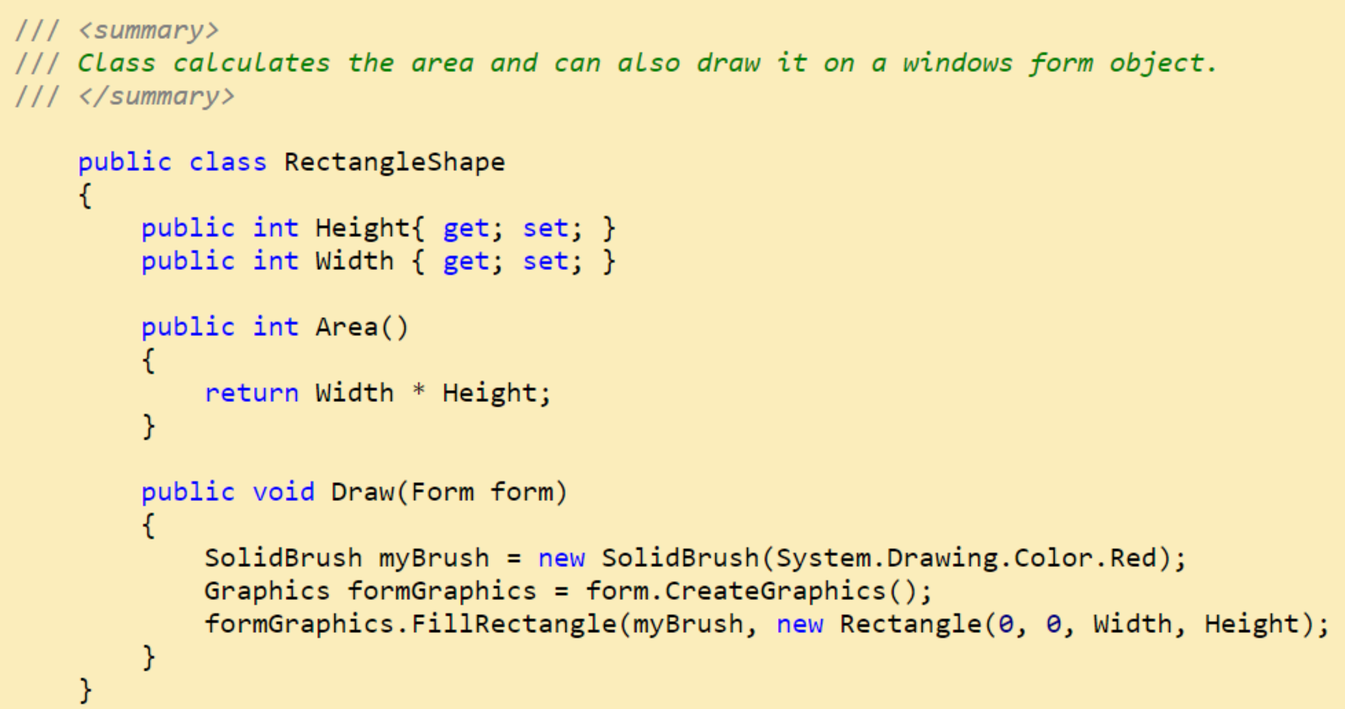
- One that calculates its rectangle area and
- One that draws the rectangle.

Area calculation changes, drawing method has to change, or properties are altered, it influences both methods.

After a code change, the class must be **tested** as a whole again.

There is clearly **more than one reason to change this class**.

Example 2

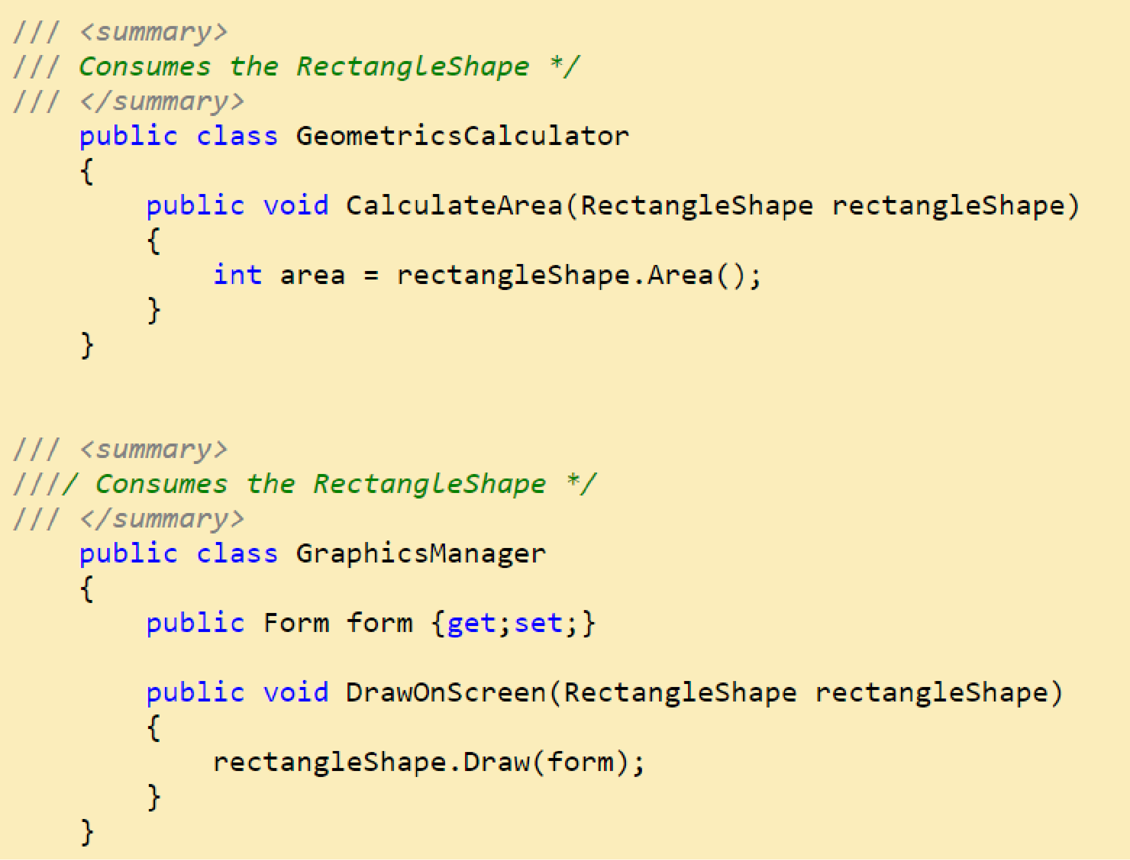


```
/// <summary>
/// Class calculates the area and can also draw it on a windows form object.
/// </summary>

    public class RectangleShape
    {
        public int Height{ get; set; }
        public int Width { get; set; }

        public int Area()
        {
            return Width * Height;
        }

        public void Draw(Form form)
        {
            SolidBrush myBrush = new SolidBrush(System.Drawing.Color.Red);
            Graphics formGraphics = form.CreateGraphics();
            formGraphics.FillRectangle(myBrush, new Rectangle(0, 0, Width, Height);
        }
    }
```

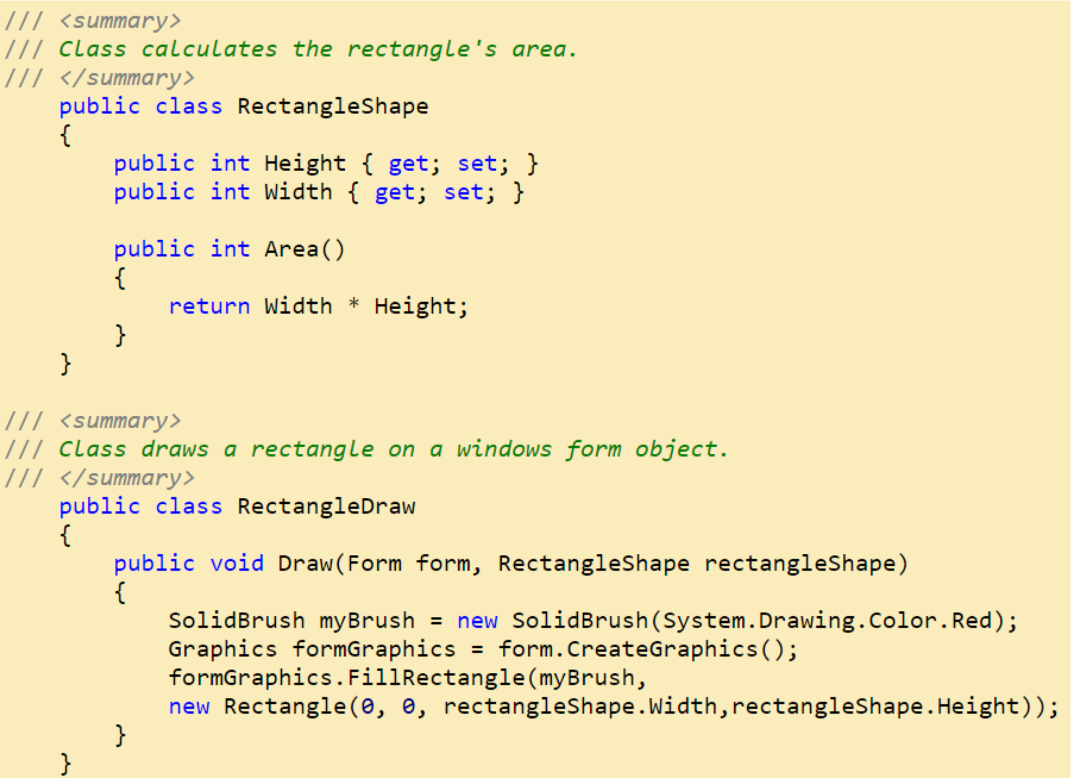


```
/// <summary>
/// Consumes the RectangleShape */
/// </summary>
    public class GeometricsCalculator
    {
        public void CalculateArea(RectangleShape rectangleShape)
        {
            int area = rectangleShape.Area();
        }
    }


/// <summary>
//// Consumes the RectangleShape */
/// </summary>
    public class GraphicsManager
    {
        public Form form {get;set;}

        public void DrawOnScreen(RectangleShape rectangleShape)
        {
            rectangleShape.Draw(form);
        }
    }
```

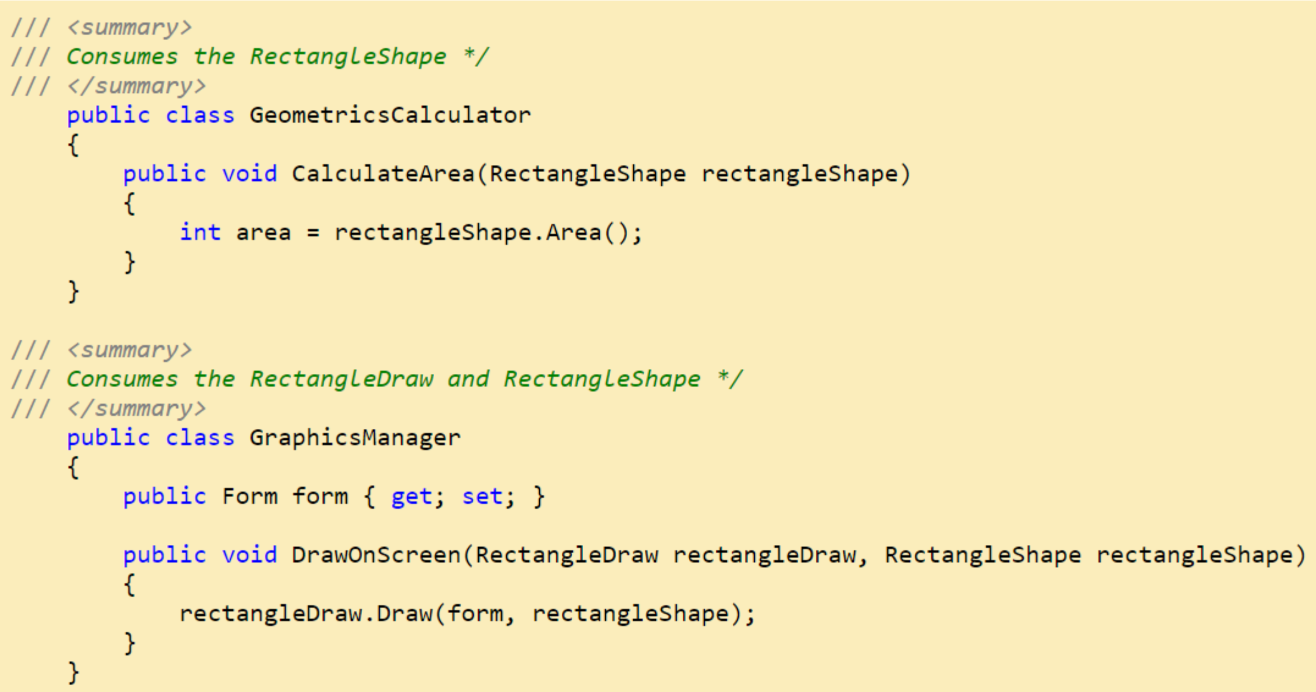
SRP Applied—Refactored



```
/// <summary>
/// Class calculates the rectangle's area.
/// </summary>
    public class RectangleShape
    {
        public int Height { get; set; }
        public int Width { get; set; }

        public int Area()
        {
            return Width * Height;
        }
    }

/// <summary>
/// Class draws a rectangle on a windows form object.
/// </summary>
    public class RectangleDraw
    {
        public void Draw(Form form, RectangleShape rectangleShape)
        {
            SolidBrush myBrush = new SolidBrush(System.Drawing.Color.Red);
            Graphics formGraphics = form.CreateGraphics();
            formGraphics.FillRectangle(myBrush,
            new Rectangle(0, 0, rectangleShape.Width,rectangleShape.Height));
        }
    }
```



```
/// <summary>
/// Consumes the RectangleShape */
/// </summary>
    public class GeometricsCalculator
    {
        public void CalculateArea(RectangleShape rectangleShape)
        {
            int area = rectangleShape.Area();
        }
    }

/// <summary>
/// Consumes the RectangleDraw and RectangleShape */
/// </summary>
    public class GraphicsManager
    {
        public Form form { get; set; }

        public void DrawOnScreen(RectangleDraw rectangleDraw, RectangleShape rectangleShape)
        {
            rectangleDraw.Draw(form, rectangleShape);
        }
    }
```

SRP Notes

What exactly represents a "reason to change“? .

Strictly SRP rule application could lead you to a lot of

- one-line methods,
- causes unnecessary code **bloat**

Trying to implement SRP on a **code base that didn't try** to implement it at all, is often

hard to refactor

Open Closed Principle (OCP)

OPEN CLOSED PRINCIPLE
Open Chest Surgery Is Not Needed When Putting On A Coat

Open Closed Principle

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." *— Robert Martin paraphrasing Bertrand Meyer, OCP paper linked from [The Principles of OOD]()*

Change a class' behavior using **inheritance** and **composition**

Open Closed Principle

A given software entity should be open for extension, but closed for modification

Any given class (or module, or function, etc) should allow for its

functionality to be extended,

but not allow for modification to its own source code

OCP Benefits

Reduce introduction of bugs into code by requiring classes to not change their own implementation

unless absolutely necessary

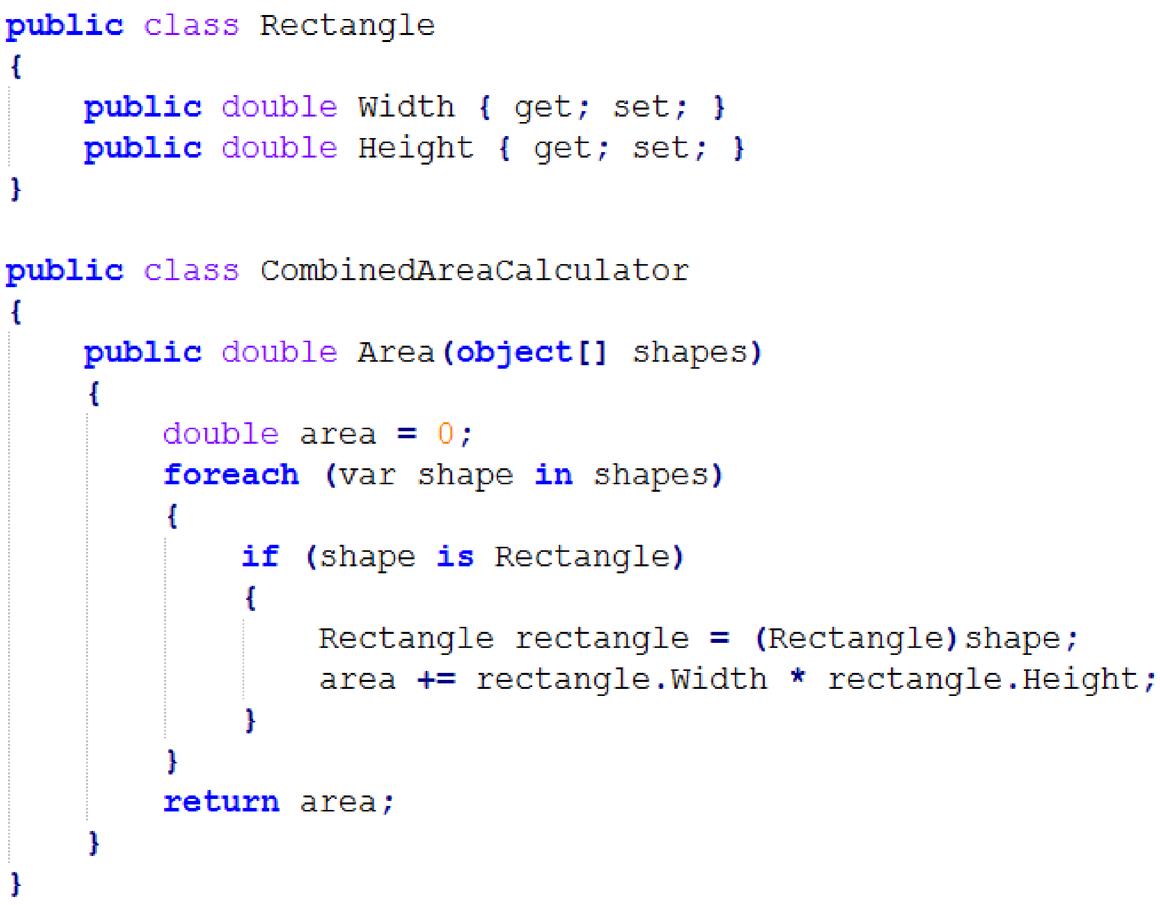
Other derived or implemented classes may be relying on that implementation to function properly

OCP Benefits

Implement classes that can easily have their functionality extended

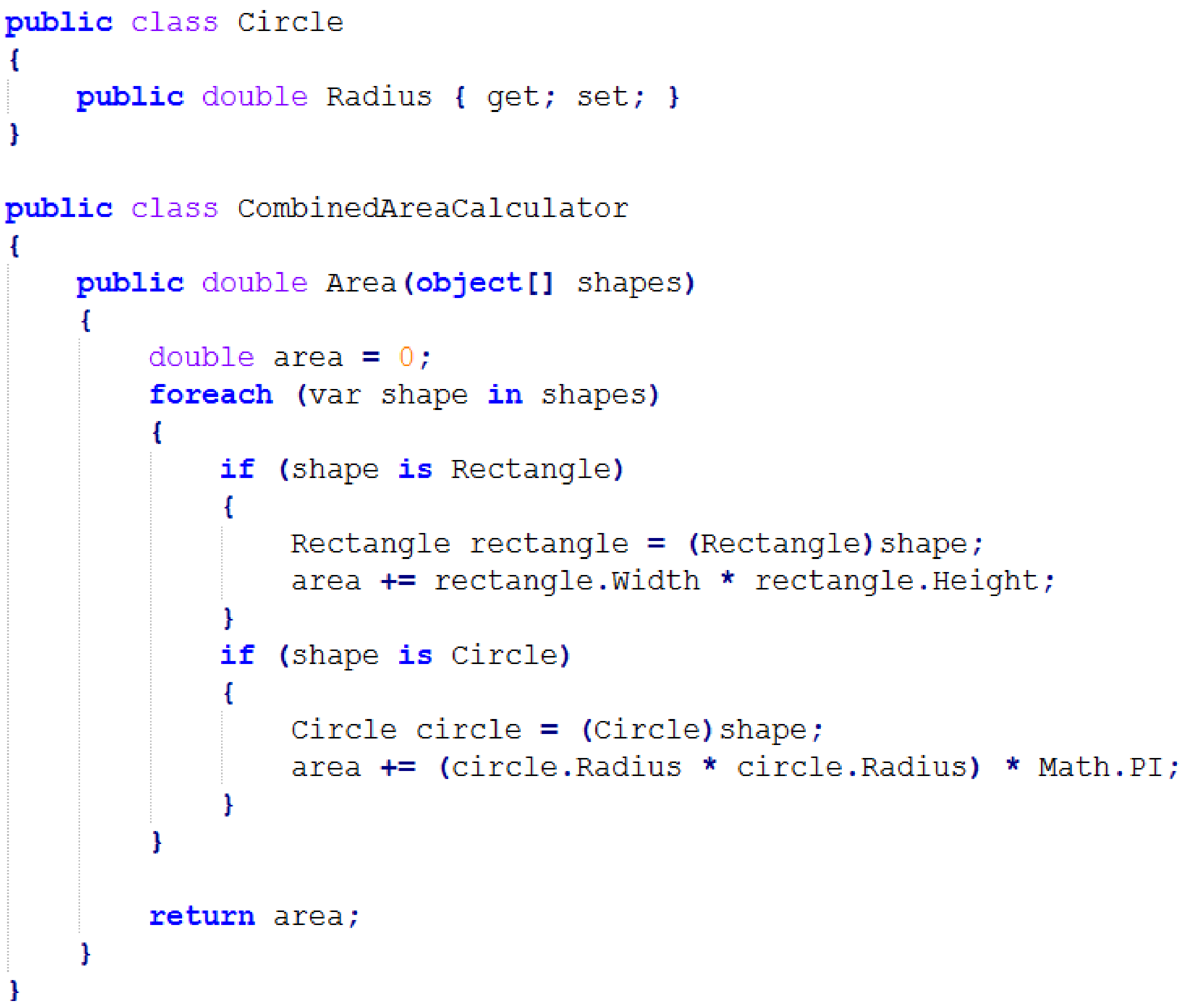
Code changes to occur **without** completely disrupting our **design**

OCP Example, add circle?



```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class CombinedAreaCalculator
{
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle)shape;
                area += rectangle.Width * rectangle.Height;
            }
        }
        return area;
    }
}
```



```
public class Circle
{
    public double Radius { get; set; }
}

public class CombinedAreaCalculator
{
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle)shape;
                area += rectangle.Width * rectangle.Height;
            }
            if (shape is Circle)
            {
                Circle circle = (Circle)shape;
                area += (circle.Radius * circle.Radius) * Math.PI;
            }
        }

        return area;
    }
}
```

Violation of Open/Closed Principle

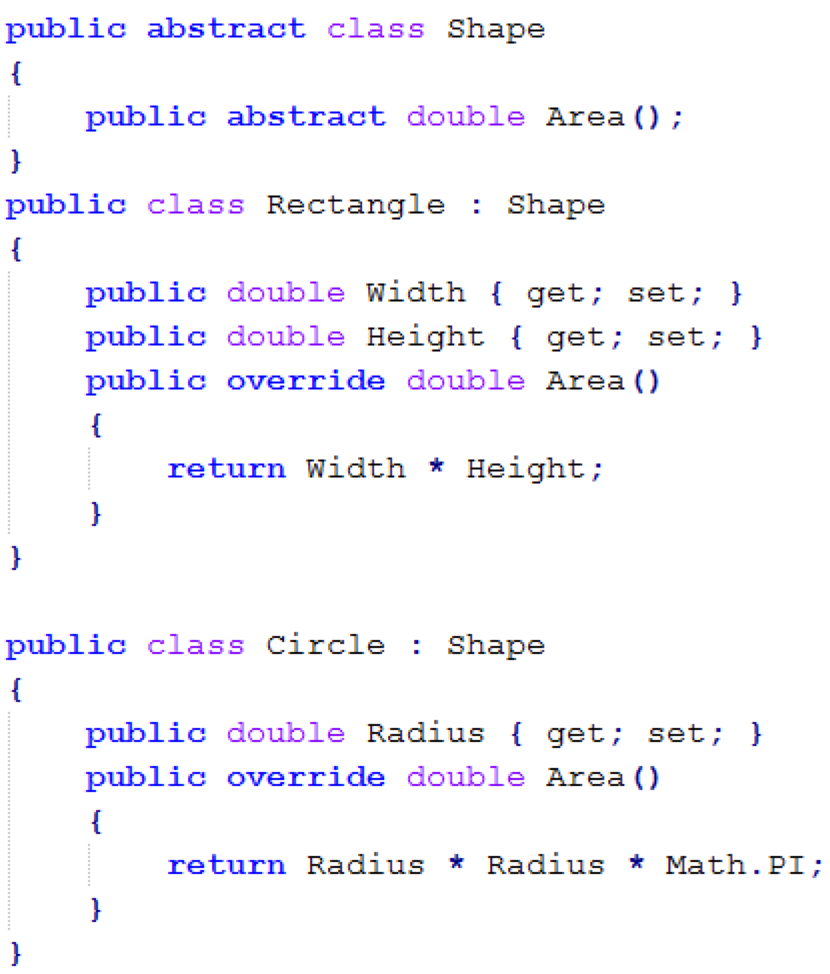
To extend **CombinedAreaCalculator** class, modify class's source!

More **shapes** … **triangles**, or **octogons**, or **trapezoids**? In each case, we have to add a new if clause to the **CombinedAreaCalculator**

In essence, **CombinedAreaCalculator** is **not closed** for **modification**, and **isn't really open for extension**

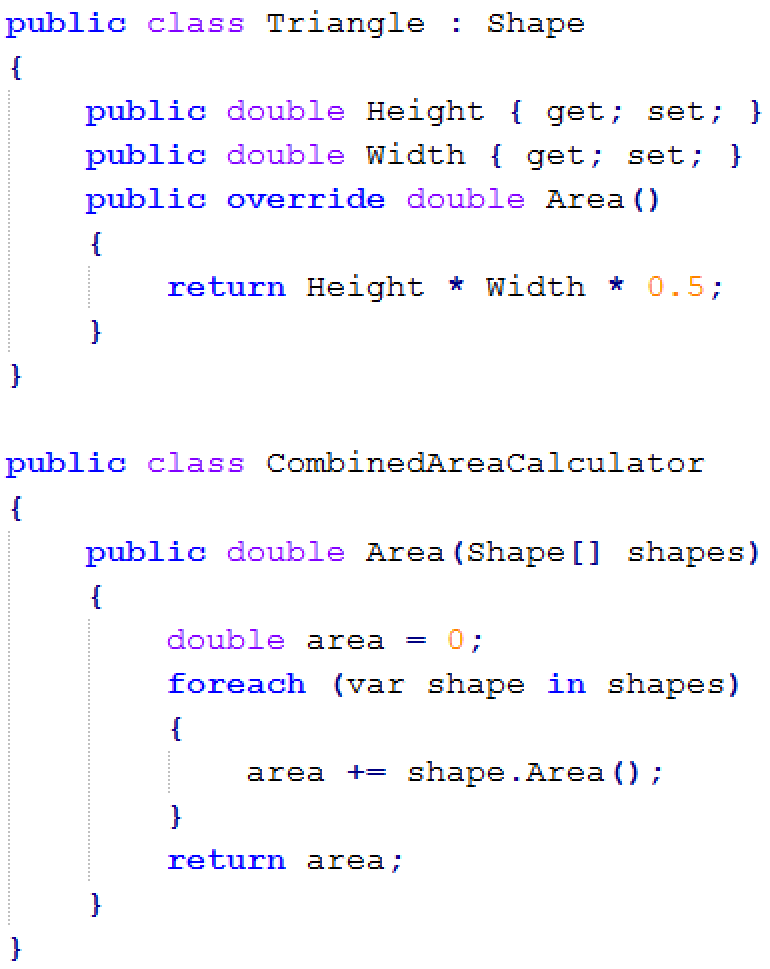
Refactor!

OCP Applied—Refactored



```
public abstract class Shape
{
    public abstract double Area();
}
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width * Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius * Radius * Math.PI;
    }
}
```



```
public class Triangle : Shape
{
    public double Height { get; set; }
    public double Width { get; set; }
    public override double Area()
    {
        return Height * Width * 0.5;
    }
}

public class CombinedAreaCalculator
{
    public double Area(Shape[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Area();
        }
        return area;
    }
}
```

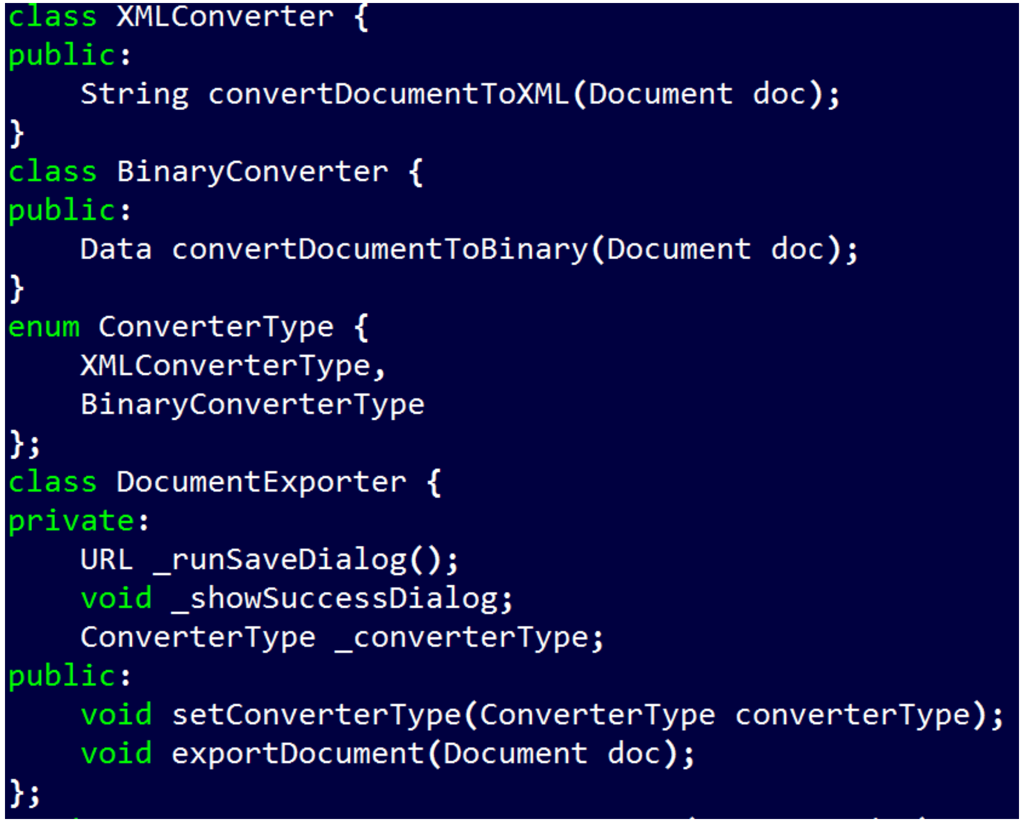
OCP Application

Shape is an **abstract** class

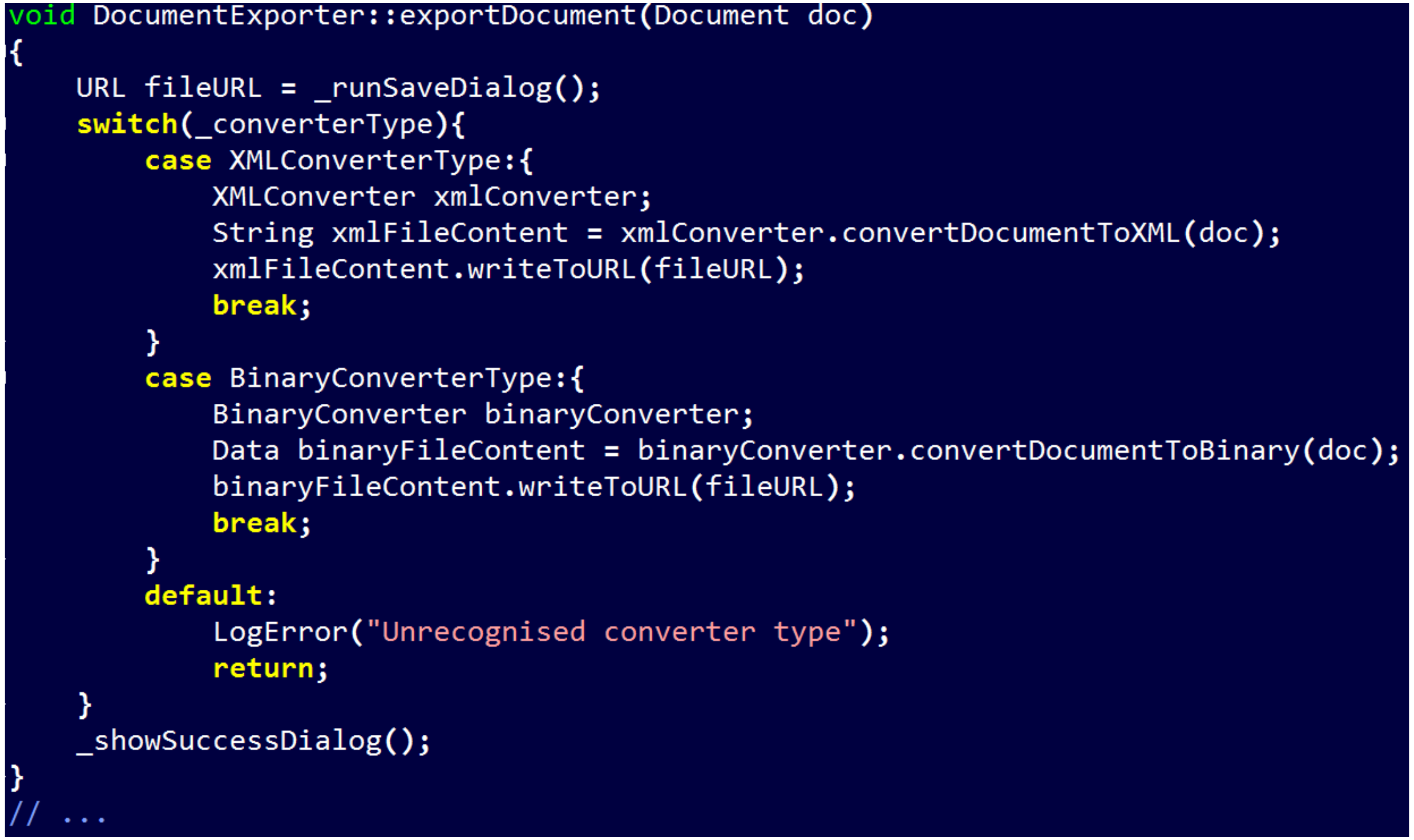
CombinedAreaCalculator class **open for extension** and **closed for modification**

Other shapes? Create a class for them that inherits from **Shape**

OCP Example 2

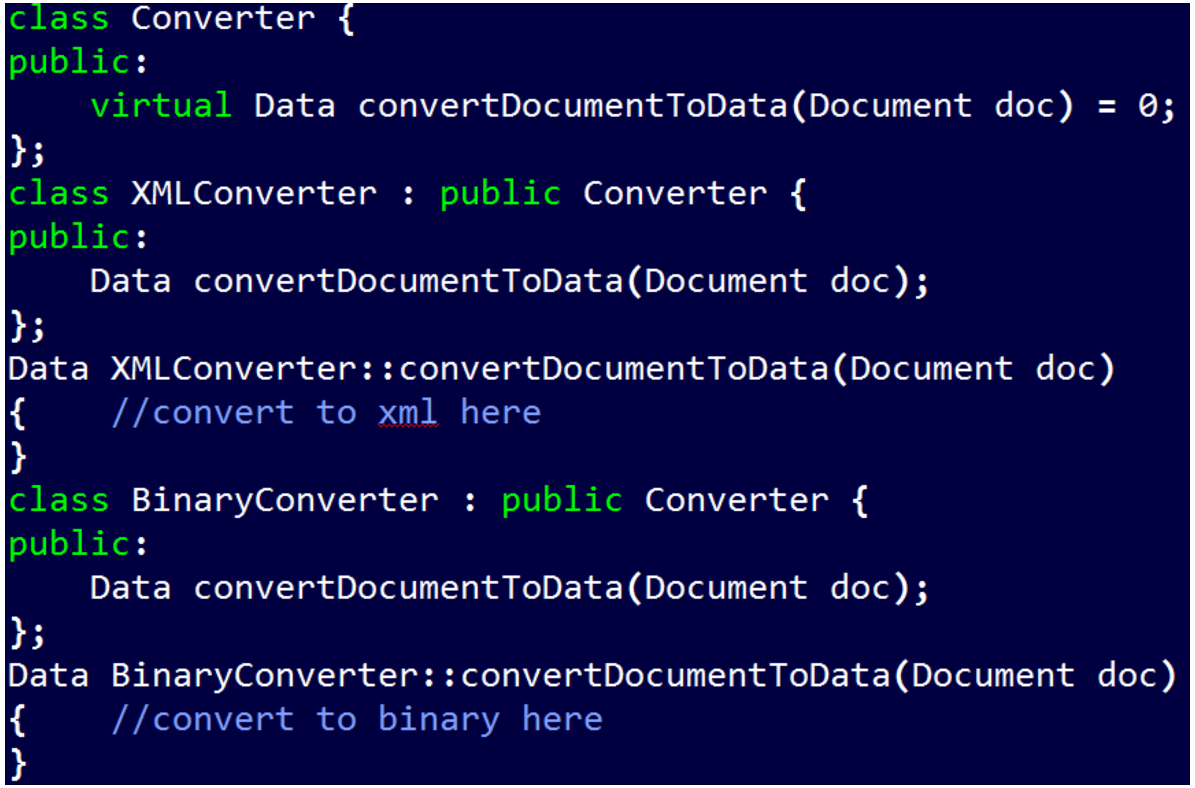


```
class XMLConverter {
public:
    String convertDocumentToXML(Document doc);
}
class BinaryConverter {
public:
    Data convertDocumentToBinary(Document doc);
}
enum ConverterType {
    XMLConverterType,
    BinaryConverterType
};
class DocumentExporter {
private:
    URL _runSaveDialog();
    void _showSuccessDialog;
    ConverterType _converterType;
public:
    void setConverterType(ConverterType converterType);
    void exportDocument(Document doc);
};
```

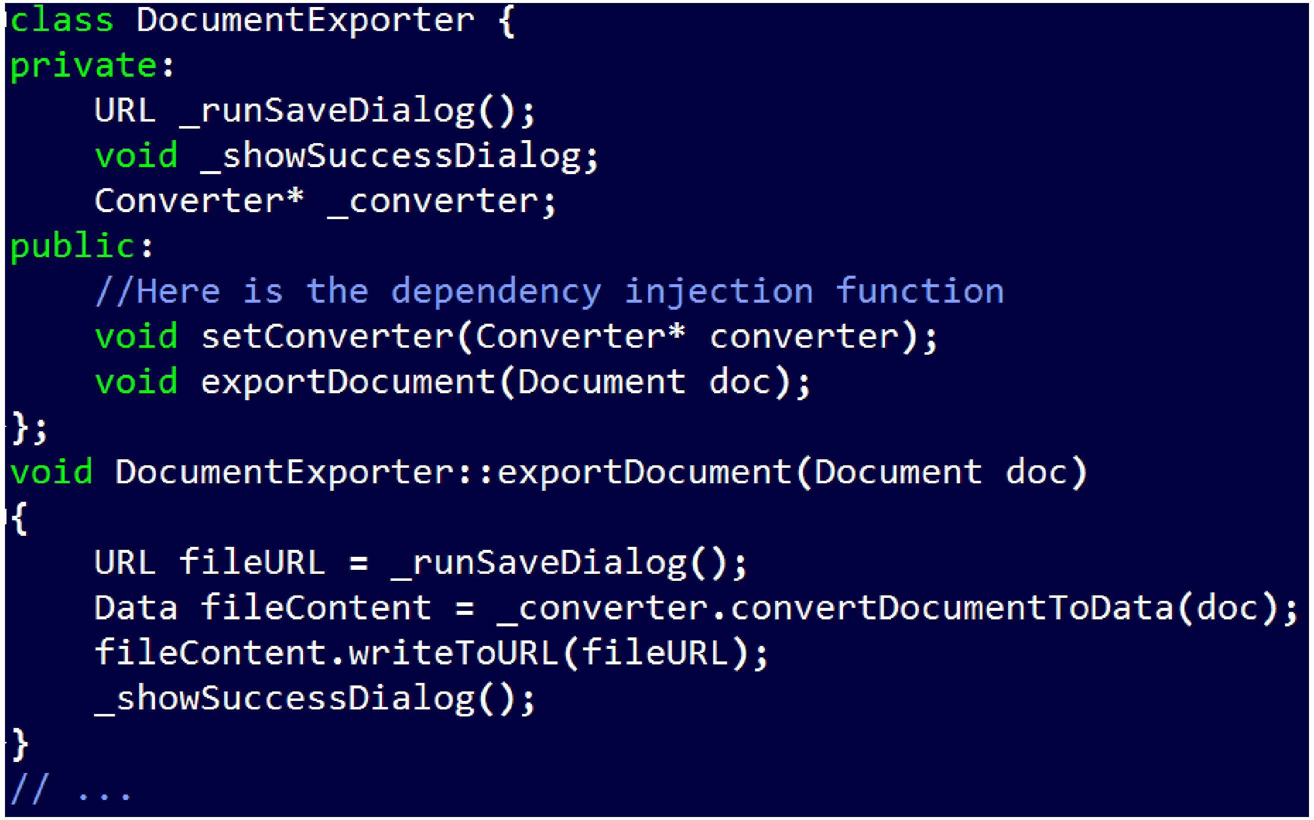


```
void DocumentExporter::exportDocument(Document doc)
{
    URL fileURL = _runSaveDialog();
    switch(_converterType){
        case XMLConverterType:{
            XMLConverter xmlConverter;
            String xmlFileContent = xmlConverter.convertDocumentToXML(doc);
            xmlFileContent.writeToURL(fileURL);
            break;
        }
        case BinaryConverterType:{
            BinaryConverter binaryConverter;
            Data binaryFileContent = binaryConverter.convertDocumentToBinary(doc);
            binaryFileContent.writeToURL(fileURL);
            break;
        }
        default:
            LogError("Unrecognised converter type");
            return;
    }
    _showSuccessDialog();
}
// ...
```

OCP Applied—Refactored .



```cpp
class Converter {
public:
    virtual Data convertDocumentToData(Document doc) = 0;
};
class XMLConverter : public Converter {
public:
    Data convertDocumentToData(Document doc);
};
Data XMLConverter::convertDocumentToData(Document doc)
{    //convert to xml here
}
class BinaryConverter : public Converter {
public:
    Data convertDocumentToData(Document doc);
};
Data BinaryConverter::convertDocumentToData(Document doc)
{    //convert to binary here
}
```



```cpp
class DocumentExporter {
private:
    URL _runSaveDialog();
    void _showSuccessDialog;
    Converter* _converter;
public:
    //Here is the dependency injection function
    void setConverter(Converter* converter);
    void exportDocument(Document doc);
};
void DocumentExporter::exportDocument(Document doc)
{
    URL fileURL = _runSaveDialog();
    Data fileContent = _converter.convertDocumentToData(doc);
    fileContent.writeToURL(fileURL);
    _showSuccessDialog();
}
// ...
```

Dependency Injection Pattern .

OCP Notes

You shouldn't interpret this rule as

"don't change already implemented classes, ever!“

Of course scenarios will arise that will force or require you to change classes that are already implemented

OCP Notes

Use discretion when attempting to make these modifications,

and keeping **OCP** in mind allows us to do that in a more **efficient** manner

Liskov Substitution Principle (LSP)

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle

*"**Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.**" — Robert Martin, LSP paper linked from [The Principles of OOD]()*

Subclasses should behave nicely when used in place of their base class

Liskov Substitution Principle

Treat a child class as though it were the parent class

All derived classes should retain the functionality of their parent class and

cannot replace any functionality the parent provides

Extend it and not replace it

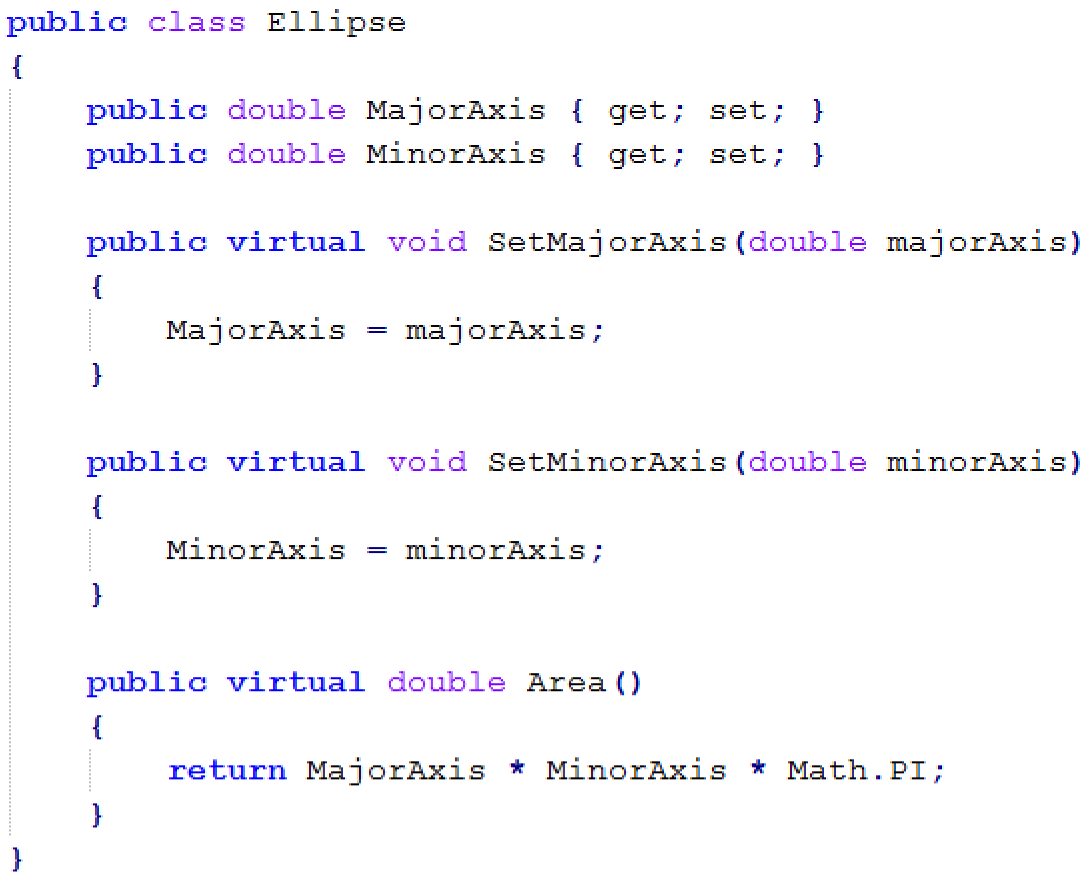
LSP Benefits

Keep functionality **intact**

To guarantee that objects lower in a relational hierarchy can be **treated as though** they are objects higher in the hierarchy

Any child class should be able to do anything the parent can do

LSP Example



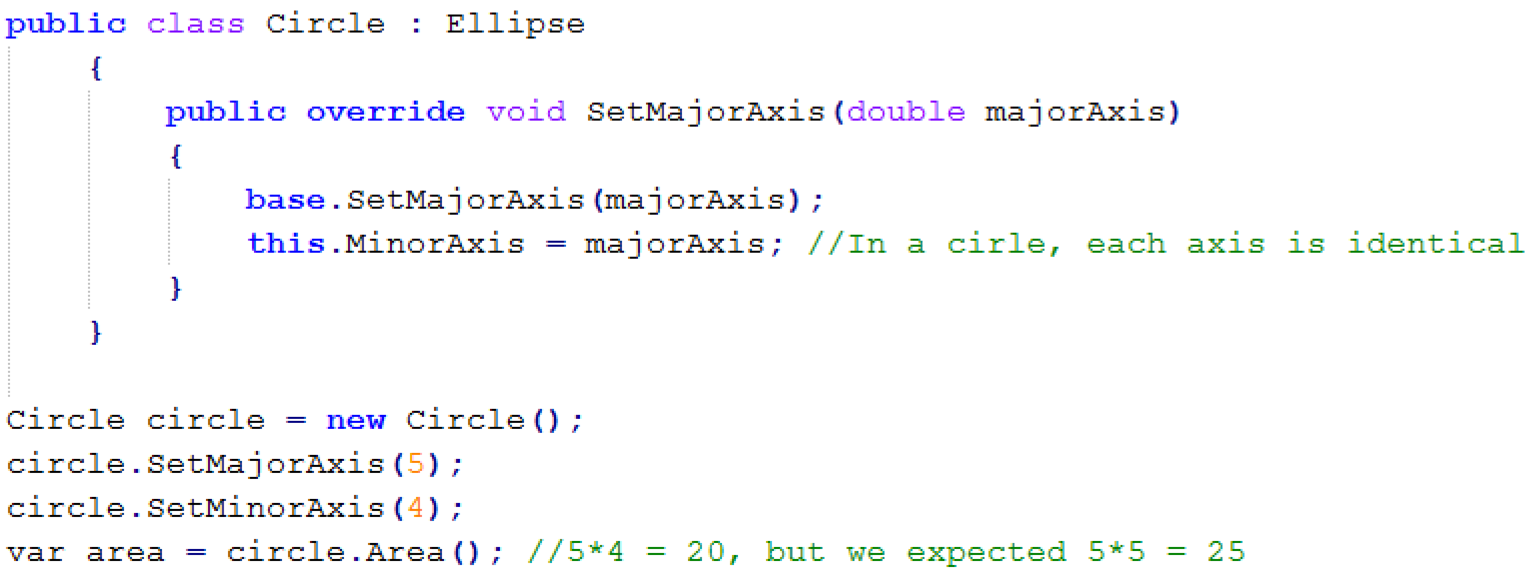
```
public class Ellipse
{
    public double MajorAxis { get; set; }
    public double MinorAxis { get; set; }

    public virtual void SetMajorAxis(double majorAxis)
    {
        MajorAxis = majorAxis;
    }

    public virtual void SetMinorAxis(double minorAxis)
    {
        MinorAxis = minorAxis;
    }

    public virtual double Area()
    {
        return MajorAxis * MinorAxis * Math.PI;
    }
}
```

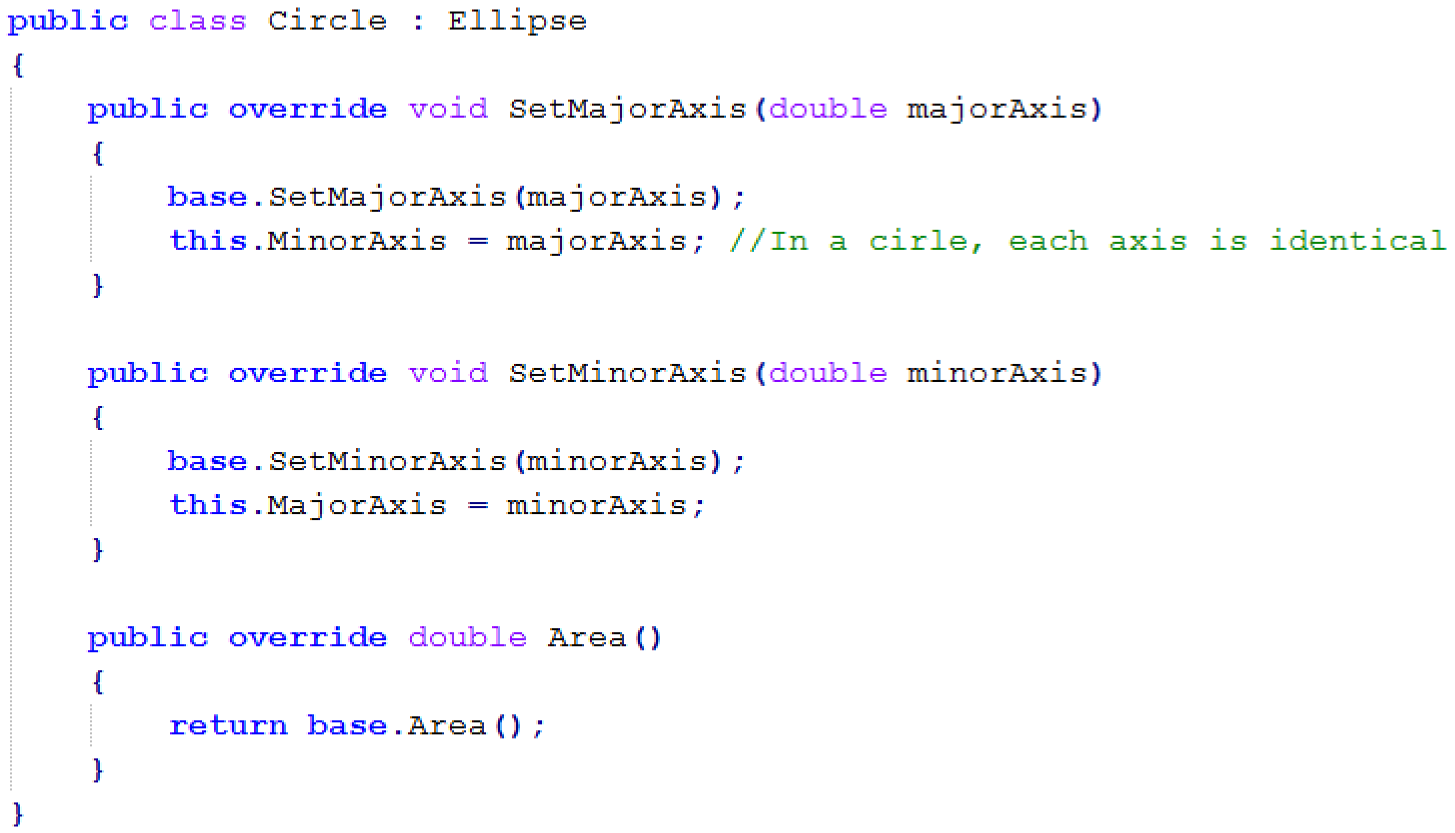
LSP Example



```
public class Circle : Ellipse
    {
        public override void SetMajorAxis(double majorAxis)
        {
            base.SetMajorAxis(majorAxis);
            this.MinorAxis = majorAxis; //In a cirle, each axis is identical
        }
    }

Circle circle = new Circle();
circle.SetMajorAxis(5);
circle.SetMinorAxis(4);
var area = circle.Area(); //5*4 = 20, but we expected 5*5 = 25
```

LSP Applied—Refactored

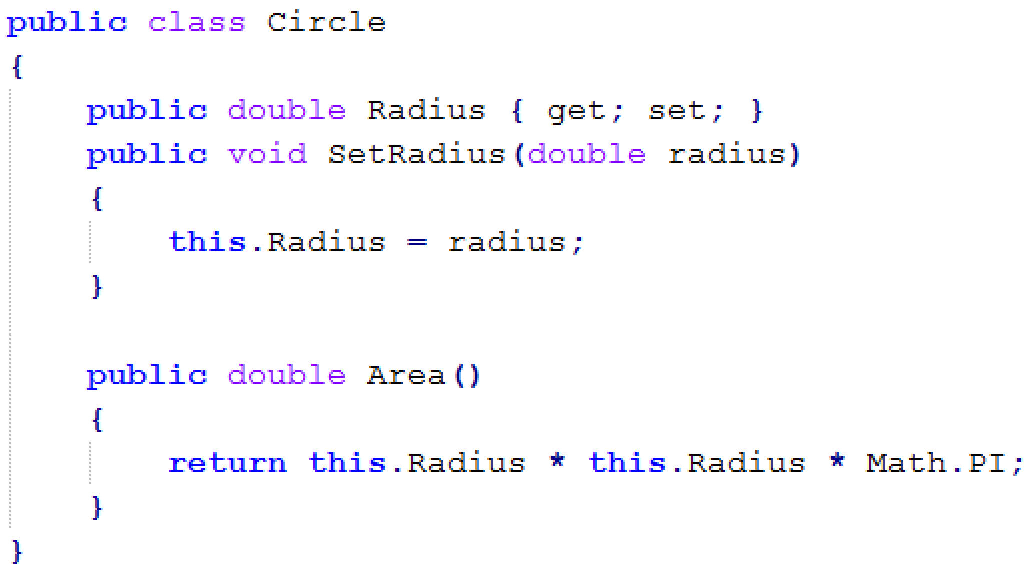


```
public class Circle : Ellipse
{
    public override void SetMajorAxis(double majorAxis)
    {
        base.SetMajorAxis(majorAxis);
        this.MinorAxis = majorAxis; //In a cirle, each axis is identical
    }

    public override void SetMinorAxis(double minorAxis)
    {
        base.SetMinorAxis(minorAxis);
        this.MajorAxis = minorAxis;
    }

    public override double Area()
    {
        return base.Area();
    }
}
```

LSP Applied—Refactored



```
public class Circle
{
    public double Radius { get; set; }
    public void SetRadius(double radius)
    {
        this.Radius = radius;
    }

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

LSP Notes

Both solutions have their own drawbacks

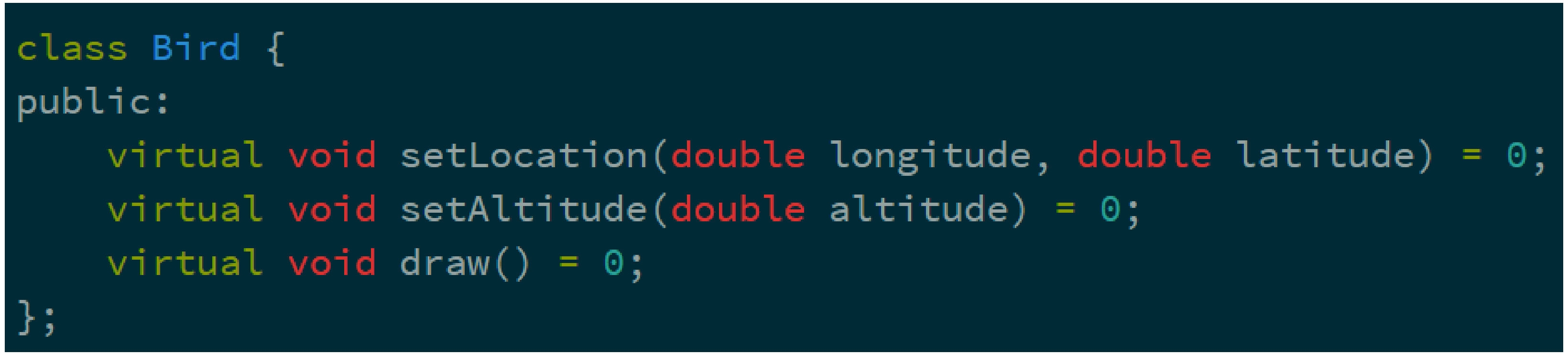
First can be considered a **hack**, since we have two different methods on the Circle class that essentially **do the same thing**

Second could be considered improper modeling, as we are treating **Circle** like a separate class even though it really is a special case of Ellipse

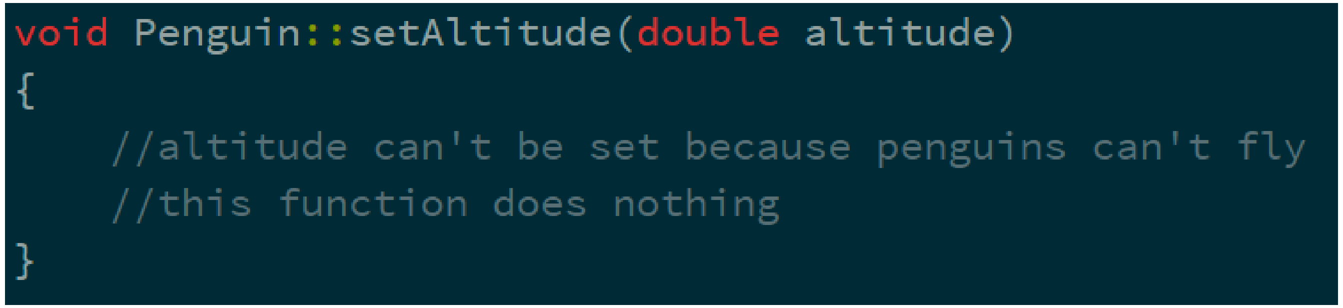
LSP is very useful in **maintaining functionality** over hierarchies

Example 2

For an application that shows birds flying around in patterns in the sky.



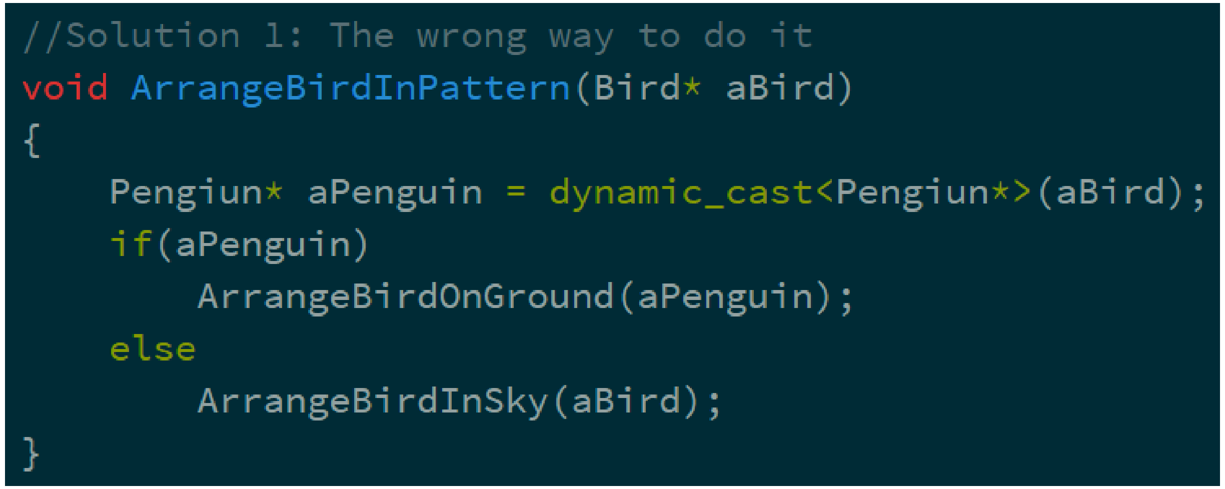
```
class Bird {
public:
    virtual void setLocation(double longitude, double latitude) = 0;
    virtual void setAltitude(double altitude) = 0;
    virtual void draw() = 0;
};
```



```
void Penguin::setAltitude(double altitude)
{
    //altitude can't be set because penguins can't fly
    //this function does nothing
}
```

The penguins are just flopping around on the ground!!
Violating The LSP

Apply LSP, Bad Solution



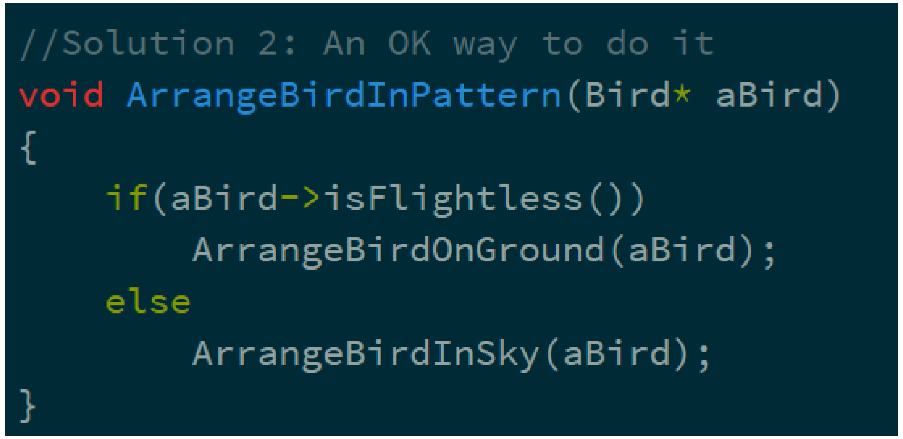
```
//Solution 1: The wrong way to do it
void ArrangeBirdInPattern(Bird* aBird)
{
    Pengiun* aPenguin = dynamic_cast<Pengiun*>(aBird);
    if(aPenguin)
        ArrangeBirdOnGround(aPenguin);
    else
        ArrangeBirdInSky(aBird);
}
```

A blatant violation of the **OCP!!**

LSP says code should work without knowing the actual class of **Bird** object.

What if you want to add another type of flightless bird, like an **Emu**? You have to go through all your existing code and check if the **Bird** pointers are actually **Emu** pointers.

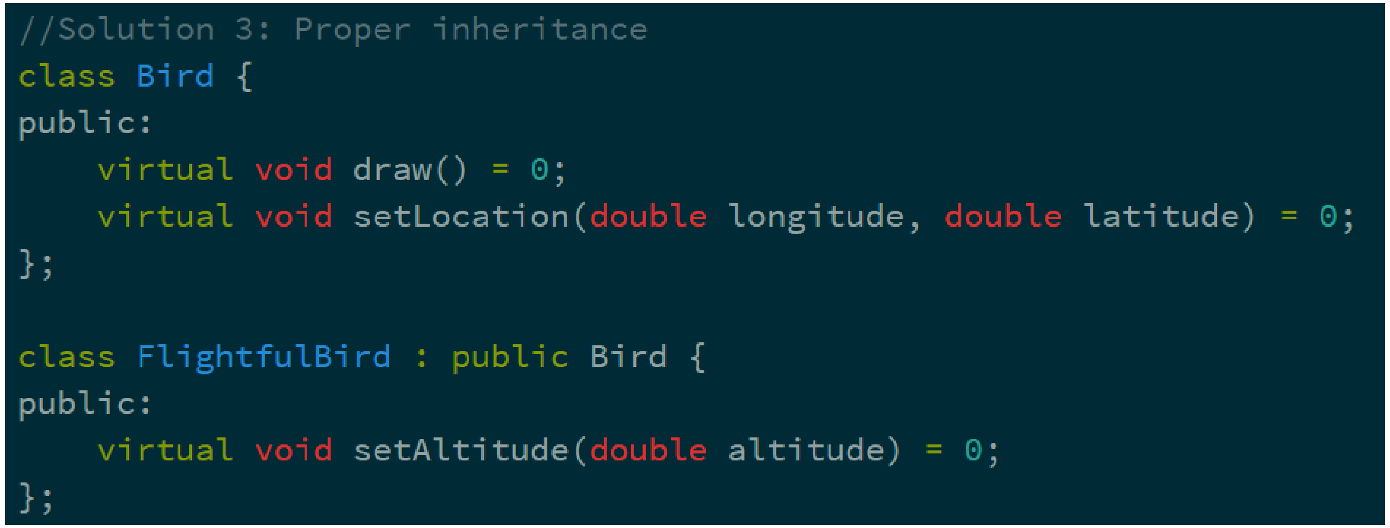
Apply LSP, A Hacky Solution



```
//Solution 2: An OK way to do it
void ArrangeBirdInPattern(Bird* aBird)
{
    if(aBird->isFlightless())
        ArrangeBirdOnGround(aBird);
    else
        ArrangeBirdInSky(aBird);
}
```

This is really a band-aid solution.
It hasn't fixed the underlying problem. It just provides a way to check whether the problem exists for a particular object.

Apply LSP, The Solution!



```
//Solution 3: Proper inheritance
class Bird {
public:
    virtual void draw() = 0;
    virtual void setLocation(double longitude, double latitude) = 0;
};

class FlightfulBird : public Bird {
public:
    virtual void setAltitude(double altitude) = 0;
};
```

Add a level to the hierarchy,
Make sure flightless bird classes don't inherit flying functionality from their superclasses
(ISP, ha, what is that? The next SOLID)

LSP Notes

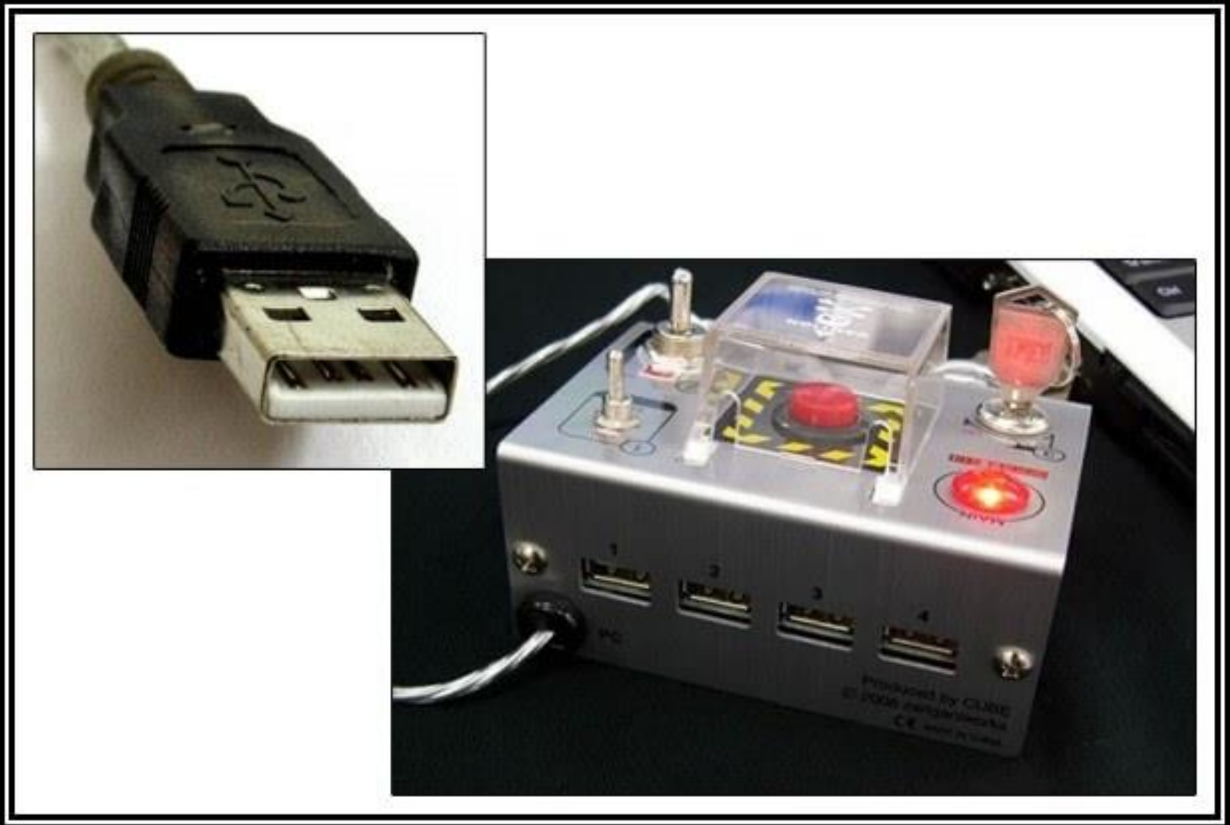
What does the violation of this principle mean?

An object doesn’t **fulfill** the contract imposed by an abstraction expressed with an interface.

Or,

It means that you identified your **abstractions wrong**.

Interface Segregation Principle (ISP)



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Interface Segregation Principle

*"**Clients should not be forced to depend upon interfaces that they do not use**." — Robert Martin, ISP paper linked from The Principles of OOD*

Keep interfaces small

Interface Segregation Principle

The Interface Segregation Principle states that no client code object should be **forced** to depend on methods it does not use

Each code object should **only implement what it needs**, and not be required to implement anything else

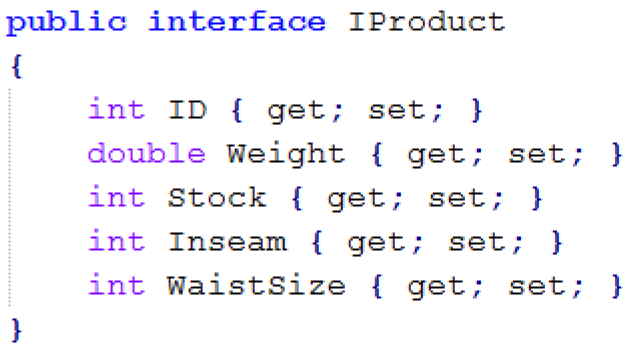
ISP Benefits

Reducing code objects down to their smallest possible implementation

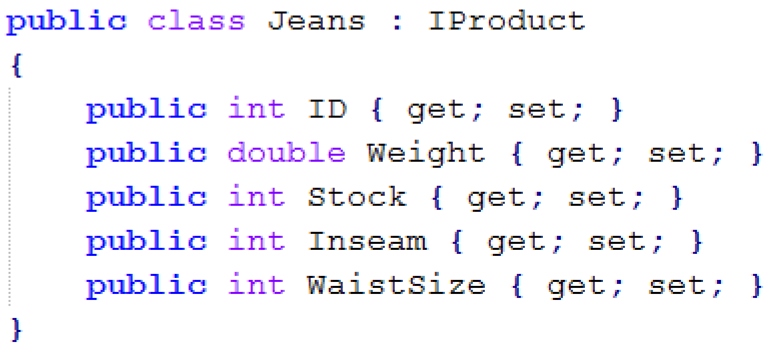
Removing dependencies the object doesn't need to function properly

Implementing this principle, is to have a lot of small, focused interfaces that define only what is needed by their implementations

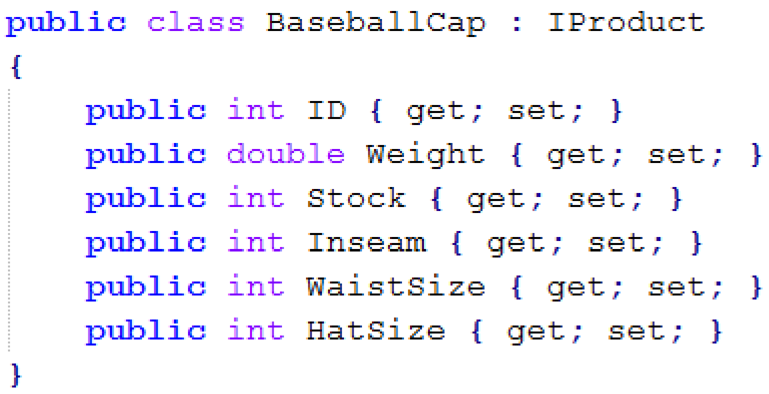
ISP Example



```
public interface IProduct
{
    int ID { get; set; }
    double Weight { get; set; }
    int Stock { get; set; }
    int Inseam { get; set; }
    int WaistSize { get; set; }
}
```

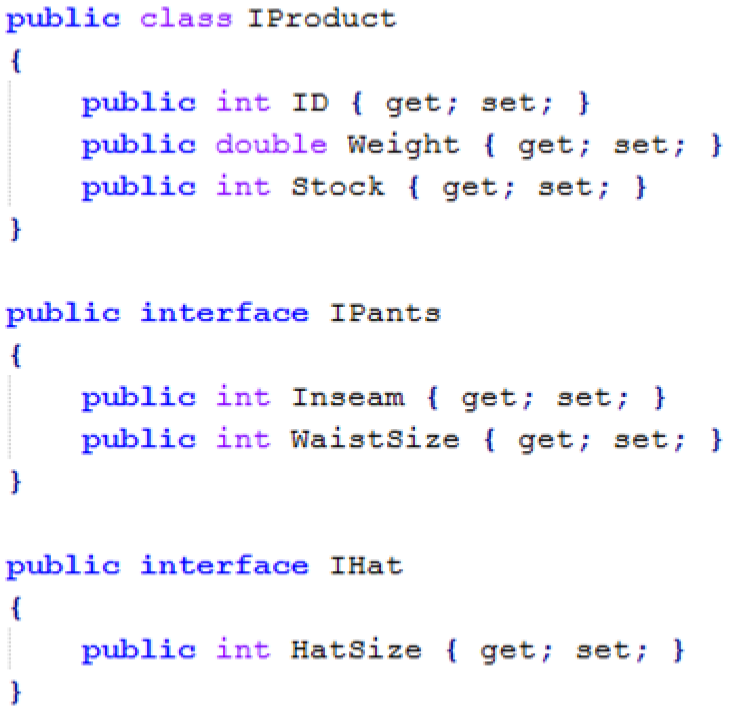


```
public class Jeans : IProduct
{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
    public int Inseam { get; set; }
    public int WaistSize { get; set; }
}
```



```
public class BaseballCap : IProduct
{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
    public int Inseam { get; set; }
    public int WaistSize { get; set; }
    public int HatSize { get; set; }
}
```

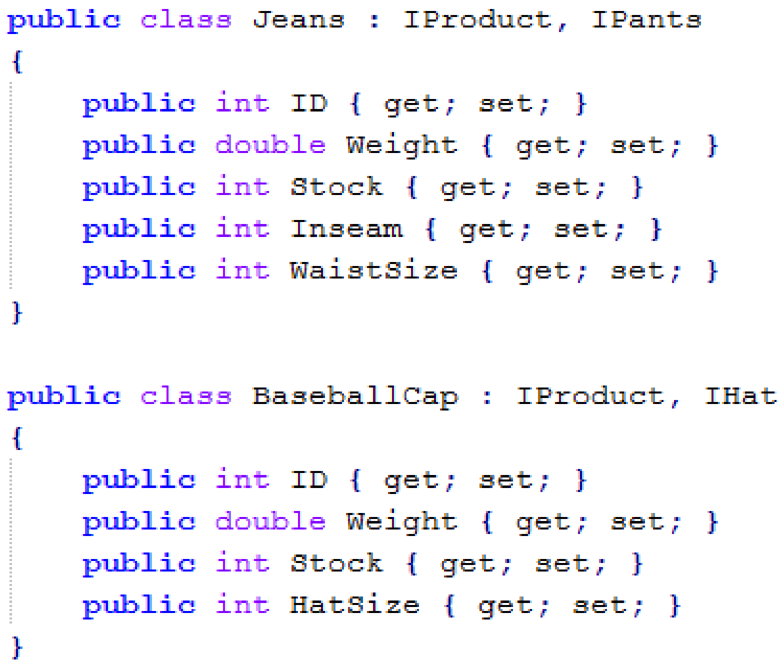
ISP Applied—Refactored



```
public class IProduct
{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
}

public interface IPants
{
    public int Inseam { get; set; }
    public int WaistSize { get; set; }
}

public interface IHat
{
    public int HatSize { get; set; }
}
```



```
public class Jeans : IProduct, IPants
{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
    public int Inseam { get; set; }
    public int WaistSize { get; set; }
}

public class BaseballCap : IProduct, IHat
{
    public int ID { get; set; }
    public double Weight { get; set; }
    public int Stock { get; set; }
    public int HatSize { get; set; }
}
```

ISP Notes

As you might have guessed from the example, the ISP can potentially result in a lot of additional interfaces

If the store was to start selling t-shirts, for example, we would probably create another interface **IShirt**

There is a possibility that we will have a **LOT** of interfaces if we strictly adhere to this rule

Dependency Inversion Principle (DIP)

DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

“A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions." *— Robert Martin, DIP paper linked from The Principles of OOD*

Dependency Inversion Principle

Use lots of interfaces and abstractions

Depend on abstractions, not on concretions

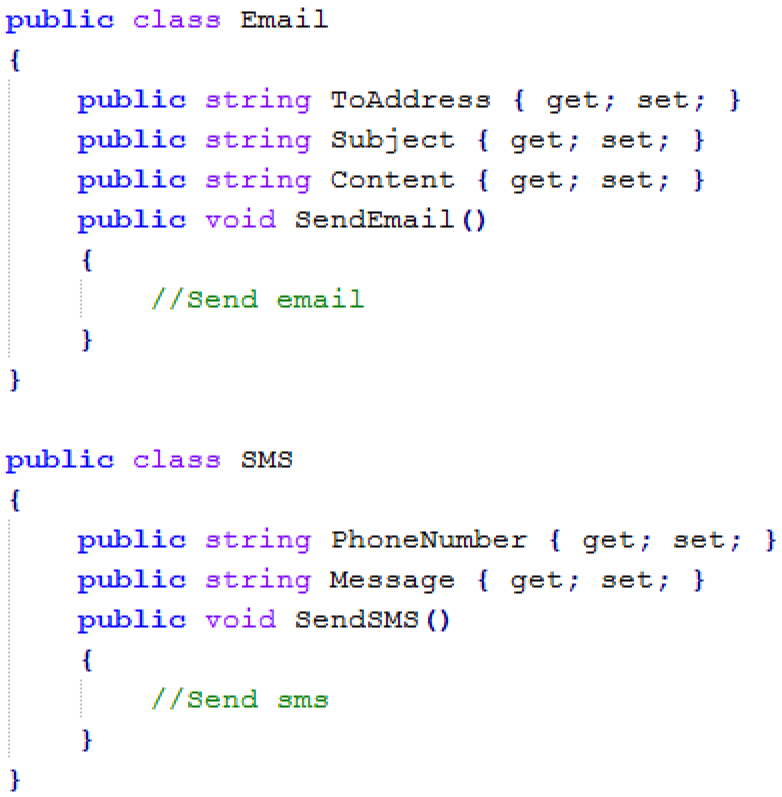
DIP Benefits

Reducing dependencies amongst the code modules

The low-level objects to define contracts (interfaces) that the high-level objects can use,

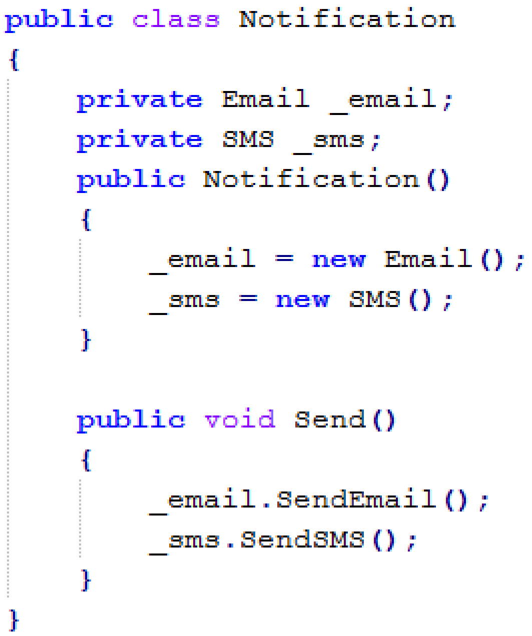
without the high-level objects needing to care about the specific implementation the low-level objects provide

DIP Example



```
public class Email
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendEmail()
    {
        //Send email
    }
}

public class SMS
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendSMS()
    {
        //Send sms
    }
}
```



```
public class Notification
{
    private Email _email;
    private SMS _sms;
    public Notification()
    {
        _email = new Email();
        _sms = new SMS();
    }

    public void Send()
    {
        _email.SendEmail();
        _sms.SendSMS();
    }
}
```

DIP Violation .

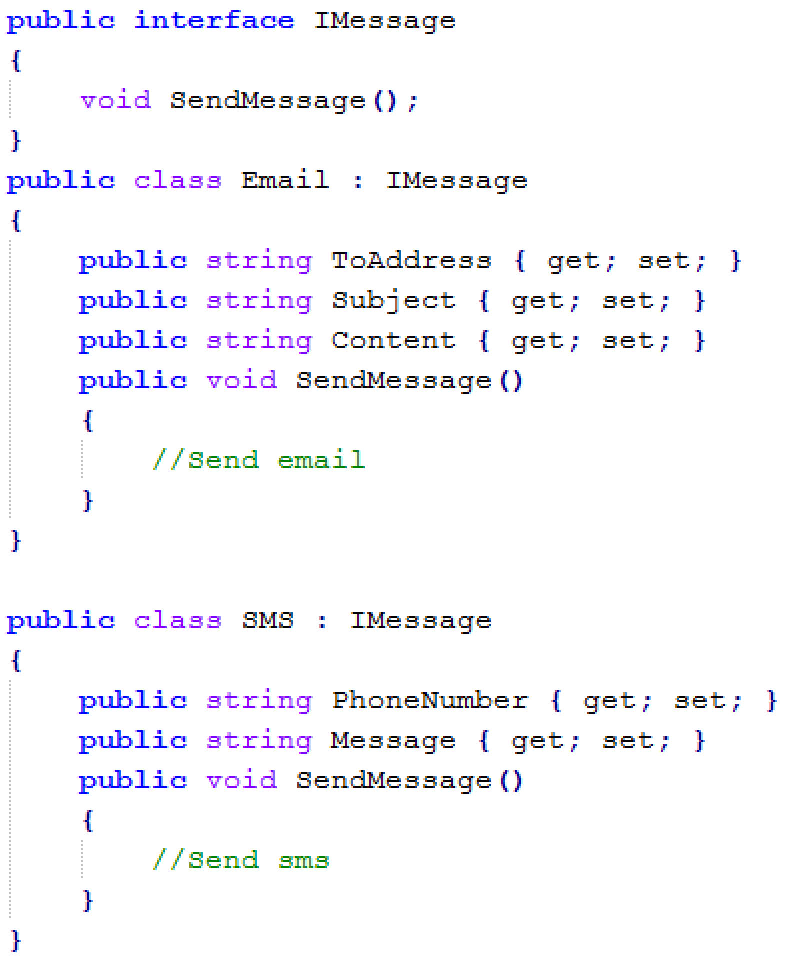
Notification class, a higher-level class, has a dependency on both lower-level classes **Email** and **SMS**

Notification is depending on the concrete implementation of both Email and SMS

Since **DIP** wants us to have both high and low-level classes depend on abstractions,

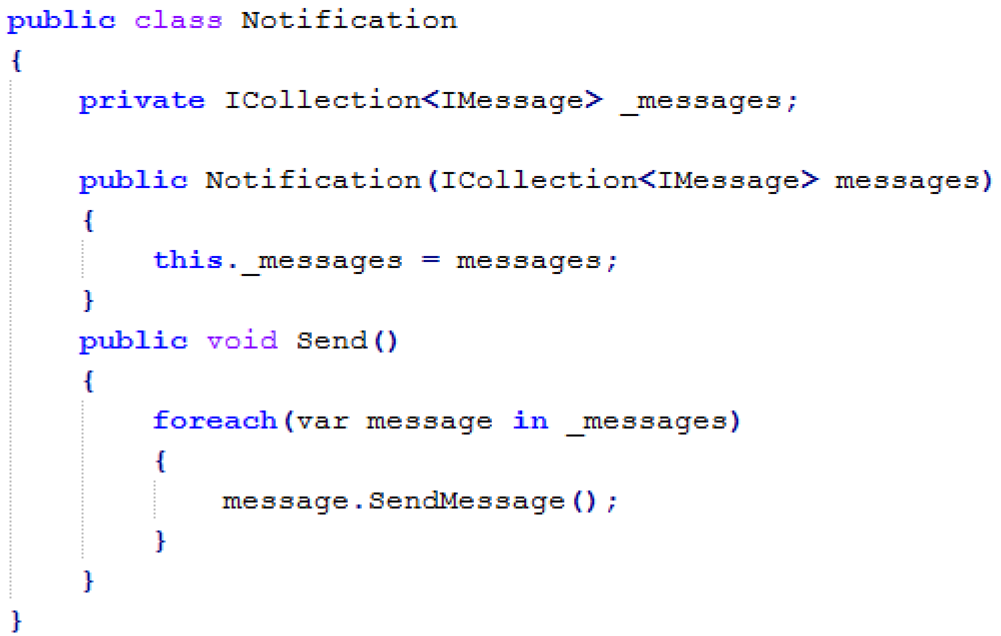
we are currently violating this principle!

DIP Applied—Refactored



```
public interface IMessage
{
    void SendMessage();
}
public class Email : IMessage
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendMessage()
    {
        //Send email
    }
}

public class SMS : IMessage
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendMessage()
    {
        //Send sms
    }
}
```



```
public class Notification
{
    private ICollection<IMessage> _messages;

    public Notification(ICollection<IMessage> messages)
    {
        this._messages = messages;
    }
    public void Send()
    {
        foreach(var message in _messages)
        {
            message.SendMessage();
        }
    }
}
```

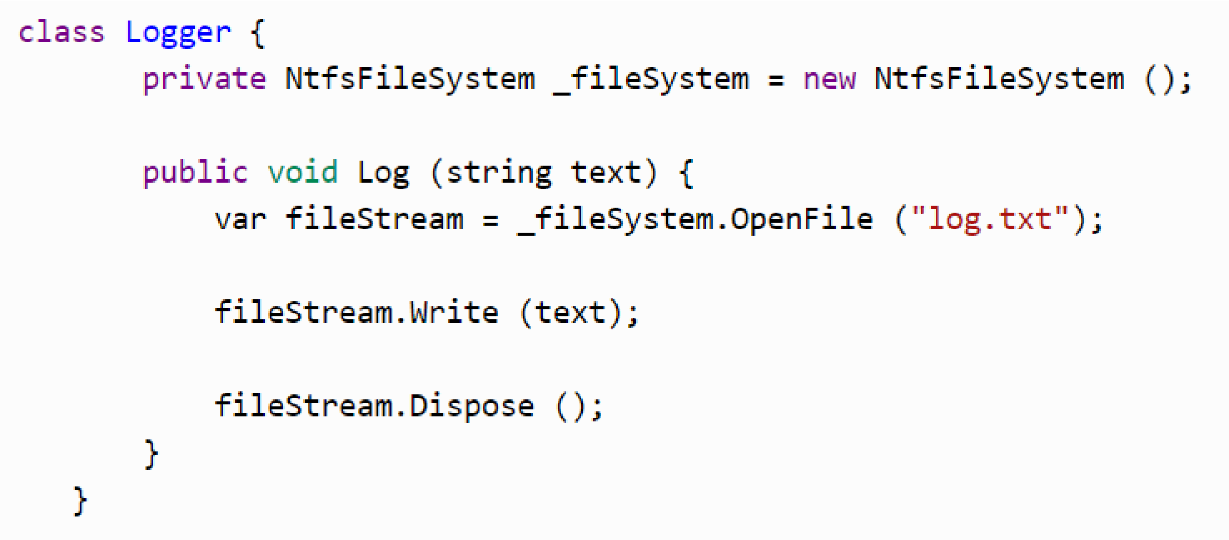
Apply DIP

Remove the dependency between **Notification** and **Email**, **Notification** and **SMS**

Notification cares about an abstraction (the interface **IMessage**)

Allowed both high-level and low-level classes to rely on **abstractions**, thereby **upholding** DIP.

Example 2



```
class Logger {
       private NtfsFileSystem _fileSystem = new NtfsFileSystem ();

       public void Log (string text) {
           var fileStream = _fileSystem.OpenFile ("log.txt");

           fileStream.Write (text);

           fileStream.Dispose ();
       }
   }
```

DIP Viloation

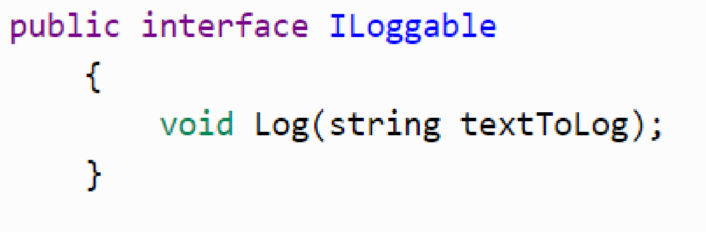
Logger class logs a text message into a specific file of the file system

Logger depends on specific implementation of **NtfsFileSystem** class

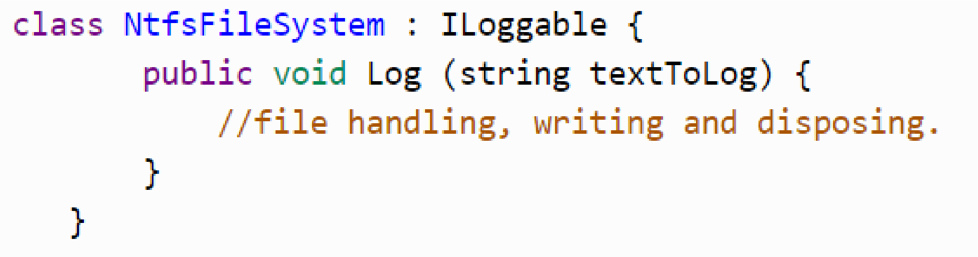
New **Ntfs** version, different **API**?
What if we need to log into a **database**?
It's likely that our class will need to change, too!

Tightly coupled and maintenance is difficult.

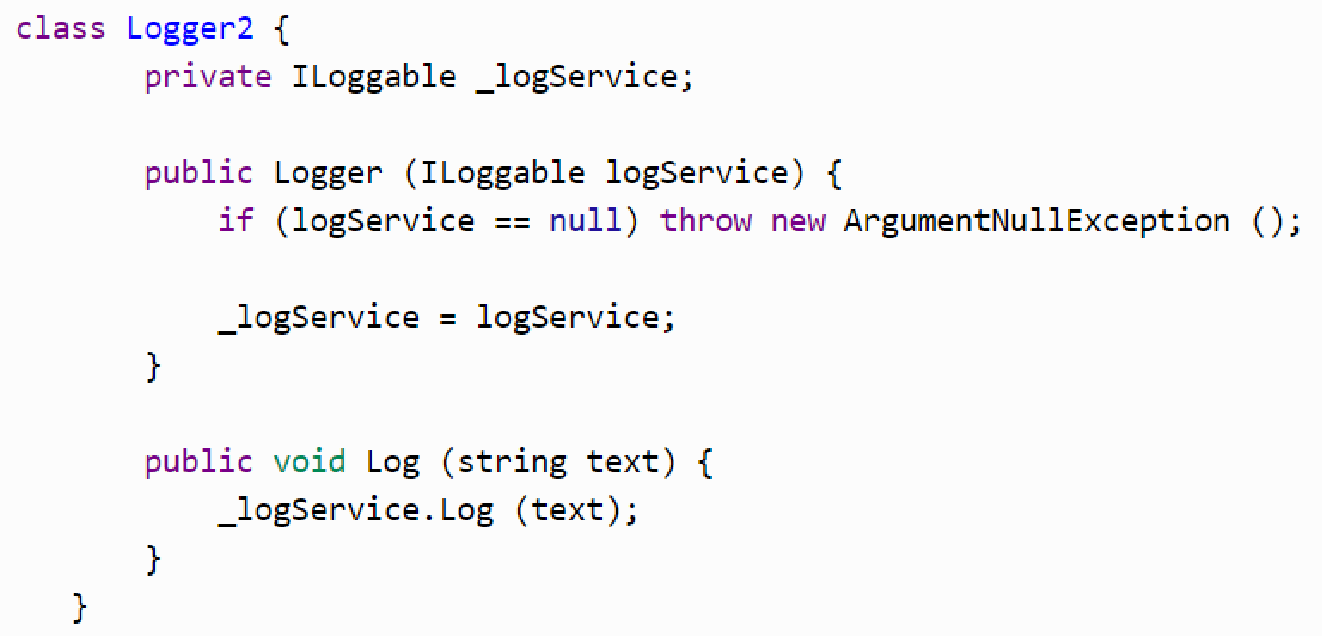
Apply DIP



```
public interface ILoggable
    {
        void Log(string textToLog);
    }
```



```
class NtfsFileSystem : ILoggable {
       public void Log (string textToLog) {
           //file handling, writing and disposing.
       }
   }
```



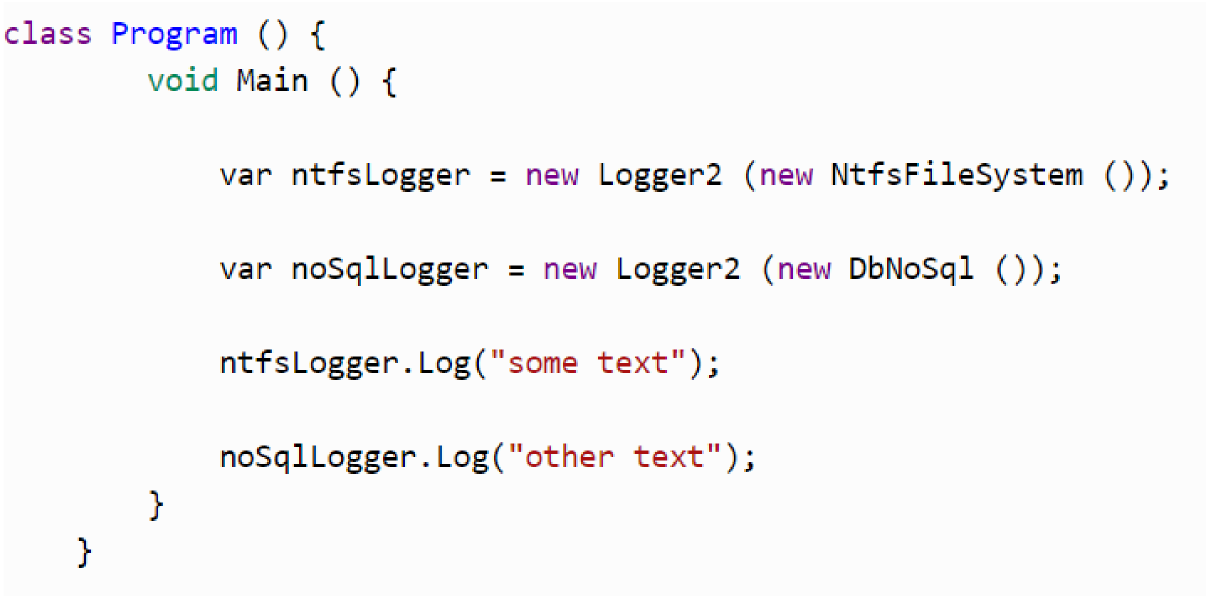
```
class Logger2 {
       private ILoggable _logService;

       public Logger (ILoggable logService) {
           if (logService == null) throw new ArgumentNullException ();

           _logService = logService;
       }

       public void Log (string text) {
           _logService.Log (text);
       }
   }
```

Apply DIP



```
class Program () {
        void Main () {

            var ntfsLogger = new Logger2 (new NtfsFileSystem ());

            var noSqlLogger = new Logger2 (new DbNoSql ());

            ntfsLogger.Log("some text");

            noSqlLogger.Log("other text");
        }
    }
```

Apply DIP

Inject (**Dependency Injection** .) into the constructor log sub-system

Logger2 class can work with **every class that implements ILoggable**

Very useful for unit testing when we would like to remove external dependencies and test our code in isolation

Code is now **loosely-coupled** because the **Logger2** class doesn't depend on a specific implementation, but only on an interface.

DIP Notes

Both high-level and low-level classes to rely on abstractions, thereby upholding the Dependency Inversion Principle

Too Much DIP

We cannot just implement a bunch of interfaces and call that DIP.

Creating code just for the sake of having it leads to unnecessary complexity

DIP Notes

DIP Enables Change Tolerance

SOLID Note

SOLID principles are to support **GRASP's Protected Variation, How?**

References

- Articles UncleBob Principles Of Ood
- Vikingcodeschool SOLID
- CodeProject Is-code-complete
- Scotch SOLID
- Wikipedia SOLID
- Sssup SOLID
- Exceptionnotfound solid-principles

Thanks!! O-O