Gang of Four Design Patterns Part 1

M19COM Software Development & Design

Week 9

Dr Faiyaz Doctor

2012

The Gang-of-Four (GoF) Design Patterns

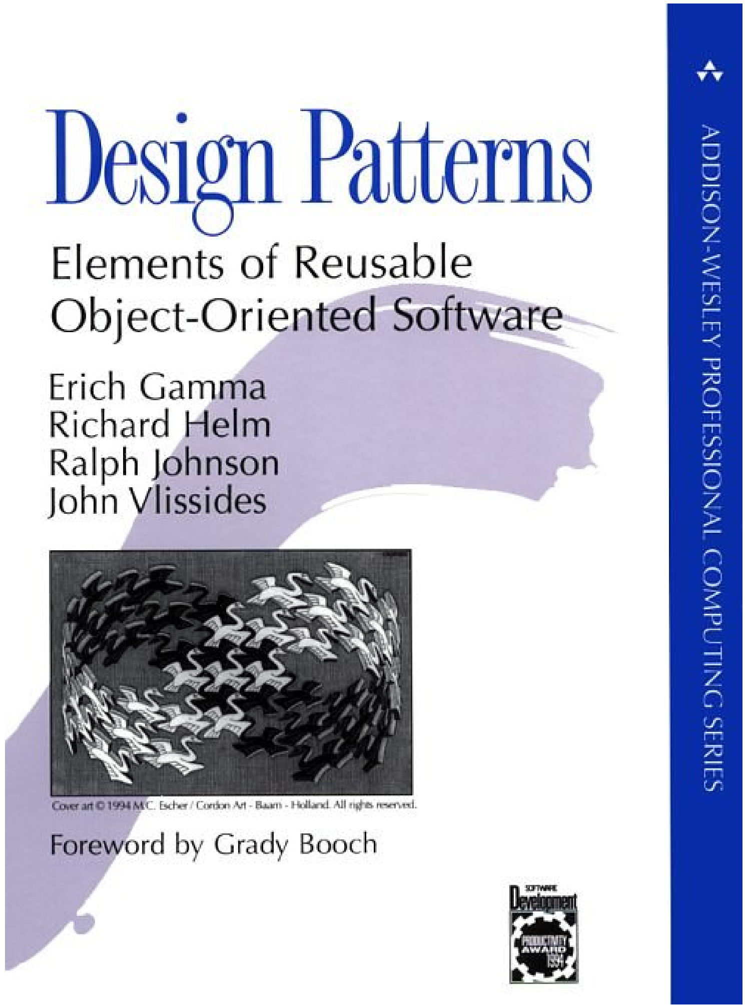
- 23 patterns
- 10-15 are widely used
- We will look at just a few

Session 1

- Adapter
- Singleton
- Factory / Factory Method
- Strategy

Session 2

- Observer
- MVC



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

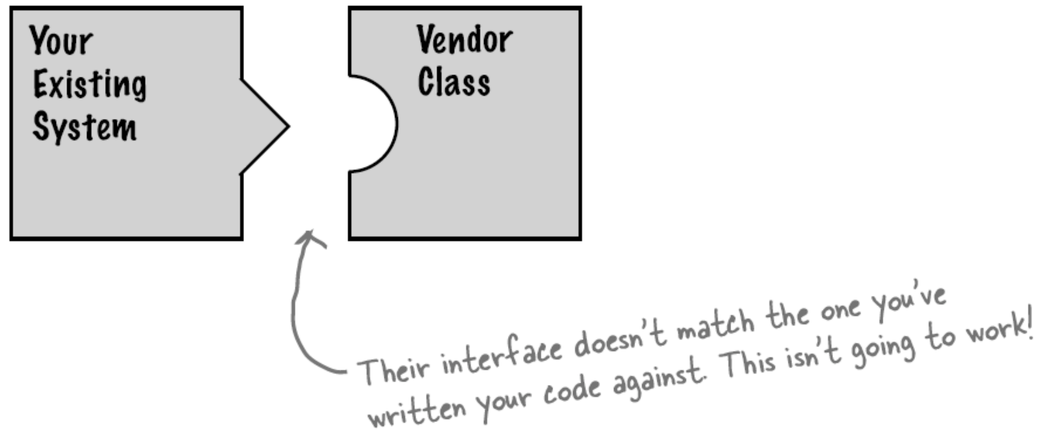
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

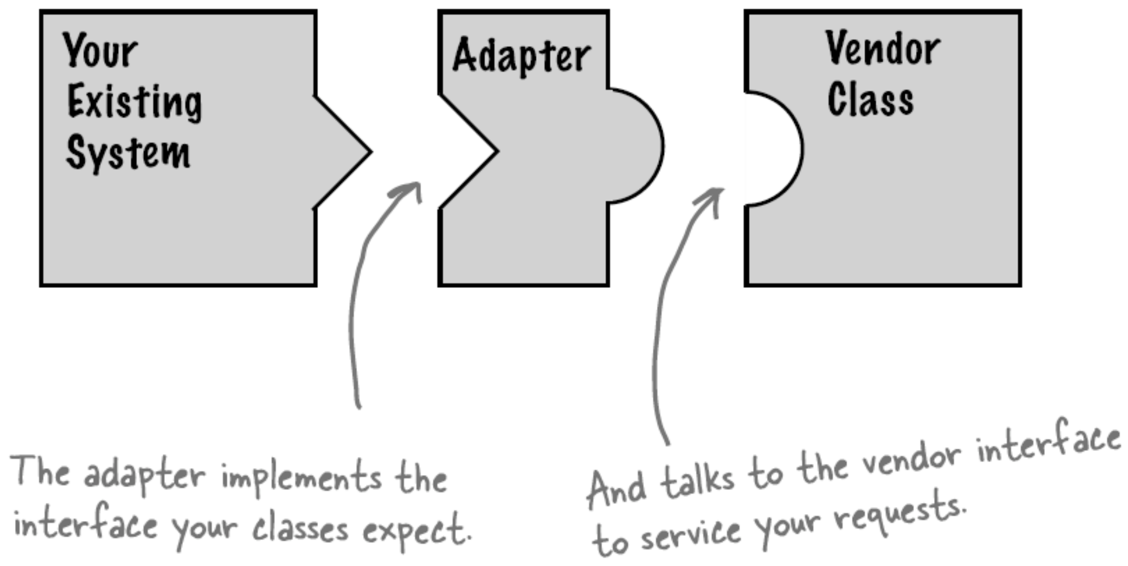
Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Acknowledgement: My examples and diagrams are from Craig Larman's book, head first design patterns Eric Freeman et al, and http://www.oodesign.com/

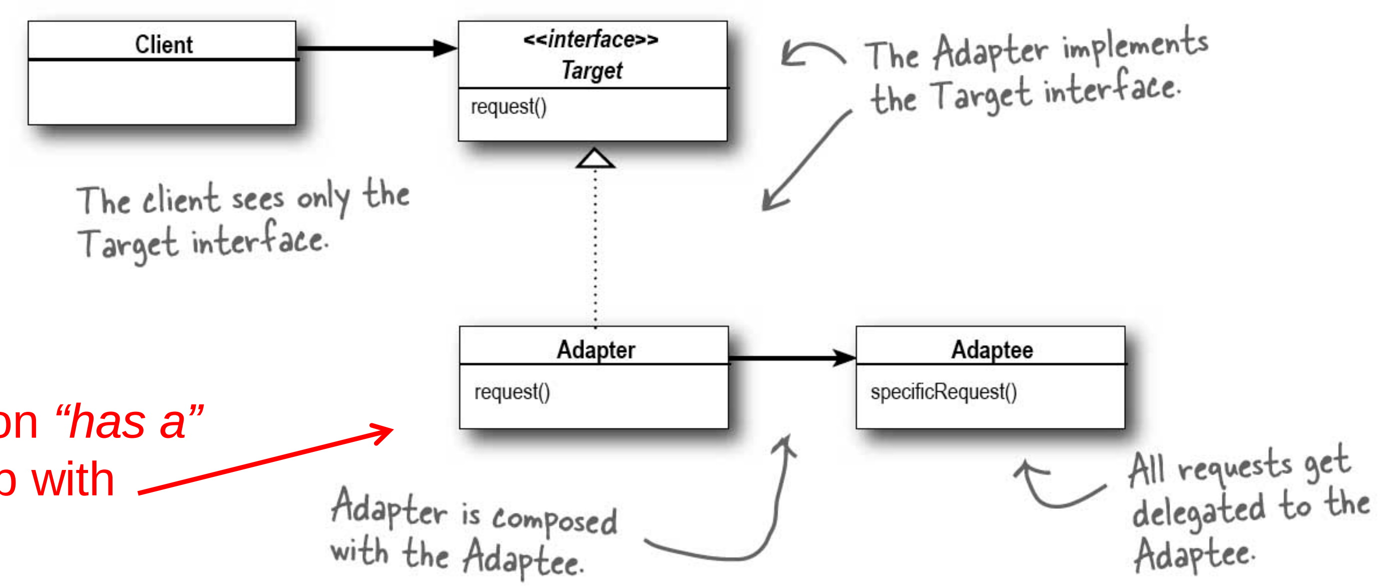
Adaptor Pattern


Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!


Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.

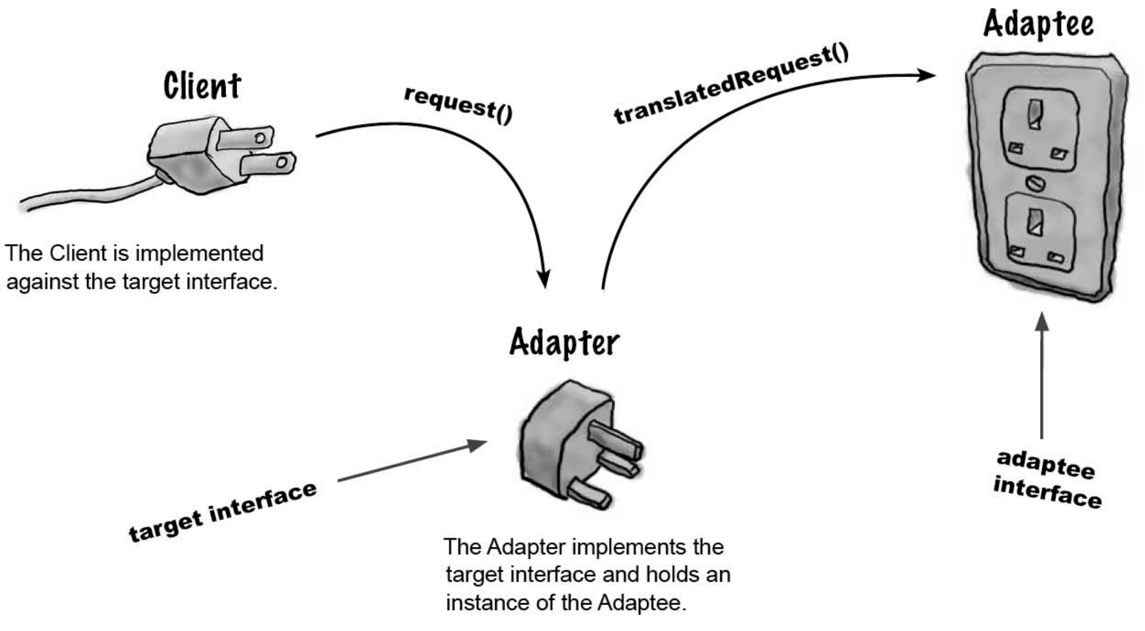
Head first design patterns, Eric Freeman et al

Adapter Pattern


Client
<<interface>>
Target
request()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter
request()
Adaptee
specificRequest()
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

Composition *"has a"* relationship with Adaptee

- *Solution: Converts the interface of a class into another interface the client expects*
- Lets classes work together that couldn't otherwise because of incompatible interfaces.
- The Adapter is known as a **structural** pattern


Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
target interface
The Adapter implements the target interface and holds an instance of the Adaptee.
adaptee interface

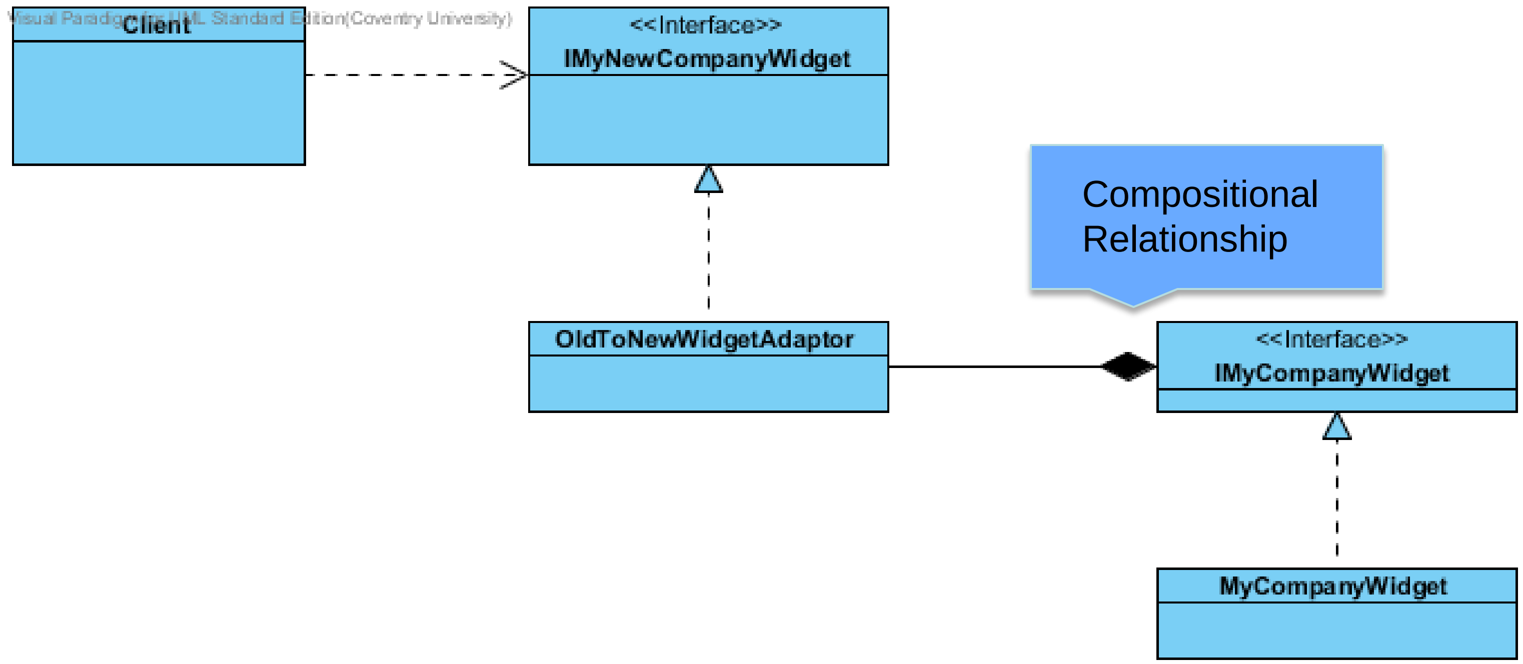
Head first design patterns, Eric Freeman et al

- **The client makes a request to the adapter by calling a method on it using the target interface.**

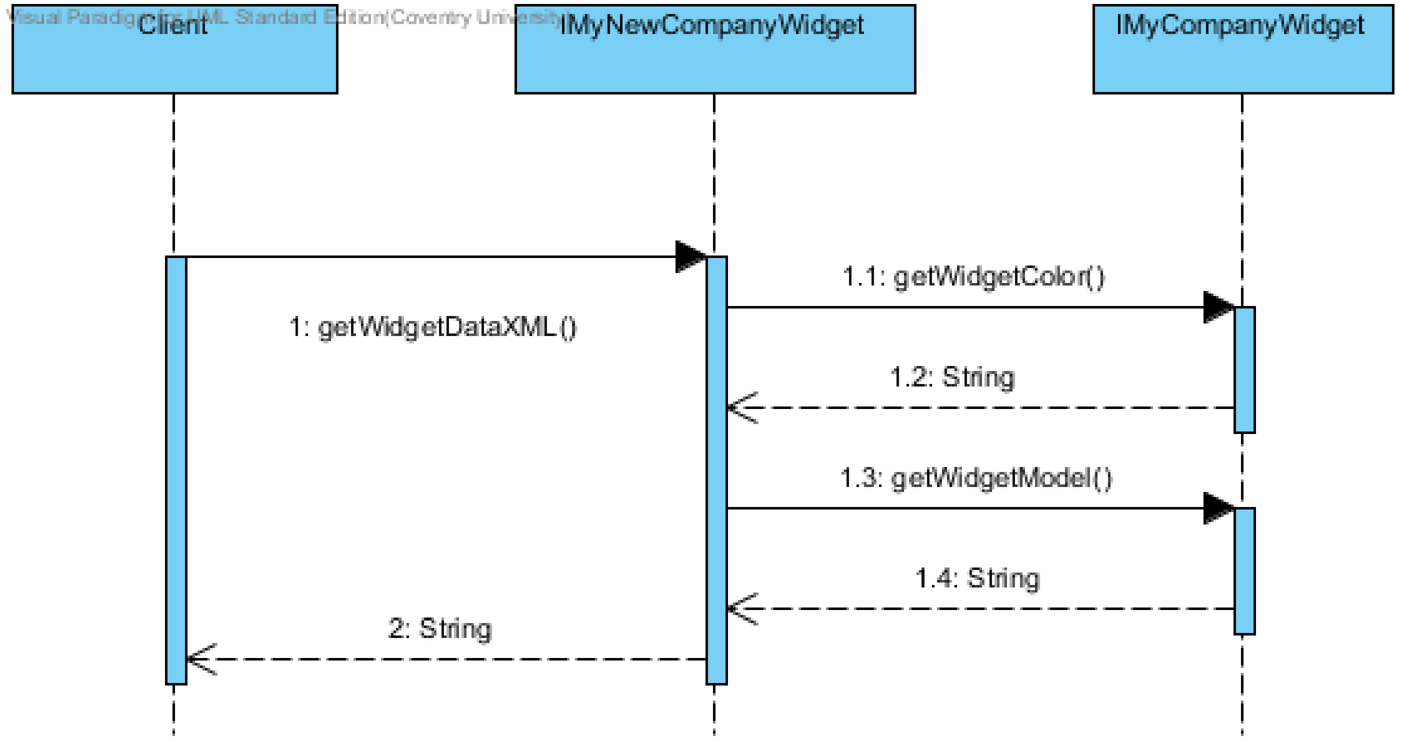
- **The adapter translates the request into one or more calls on the adaptee using the adaptee interface (*implemented in the vendors code*).**

- **The client receives the results of the call and never knows there is an adapter doing the translation (*coupling between client code and vendor code is low*).**

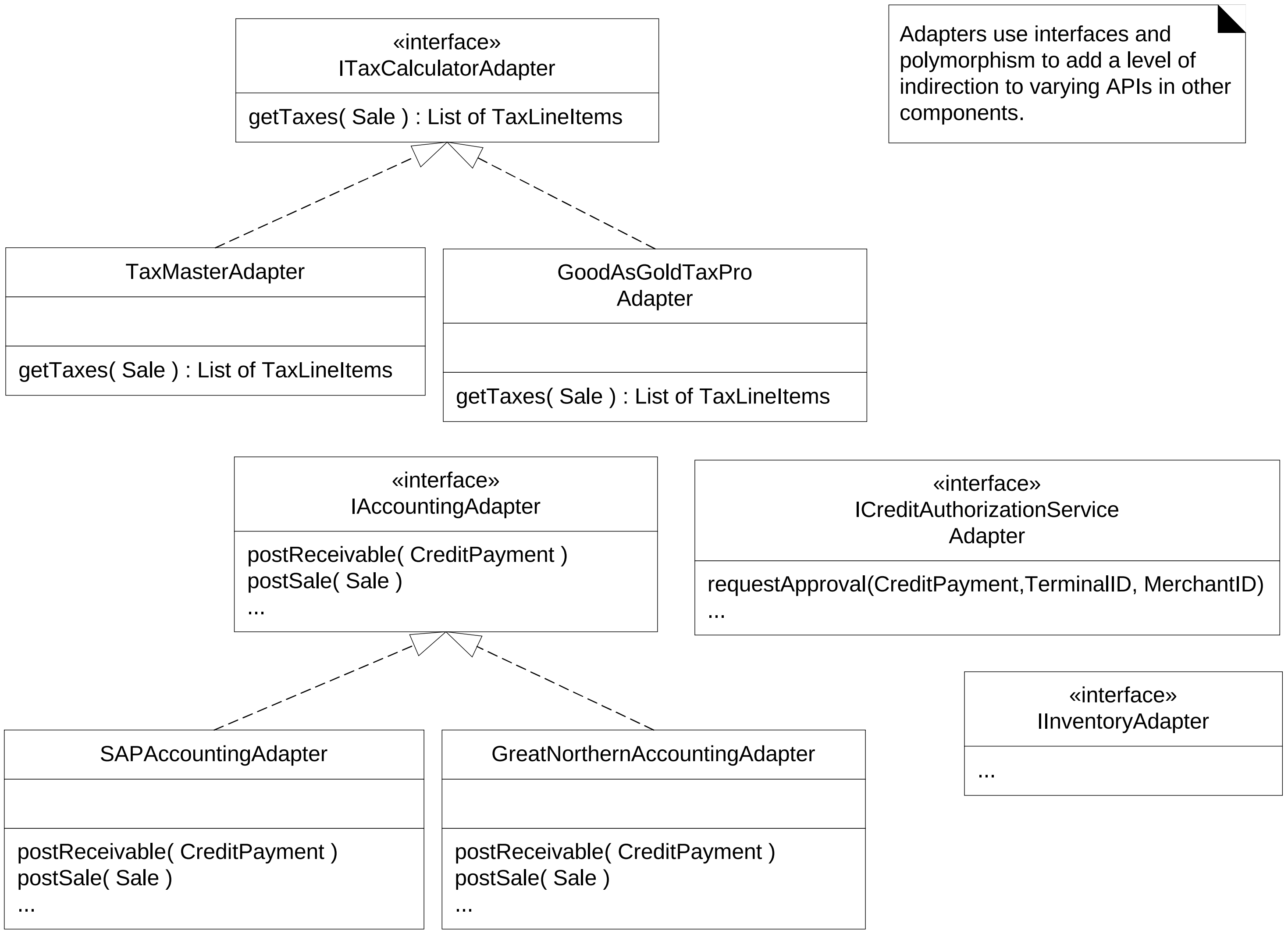
Worked Example of Adaptor Pattern



Worked Example of Adaptor Pattern



Another Adaptor Pattern Example


«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
...

“Higher” Patterns are built on basic GRASP ideas

Most higher level patterns can be seen as specialisations of one or more lower level GRASP patterns.

The Adaptor pattern incorporates

- – Low coupling
- – Polymorphism
- – Pure Fabrication
- – Indirection

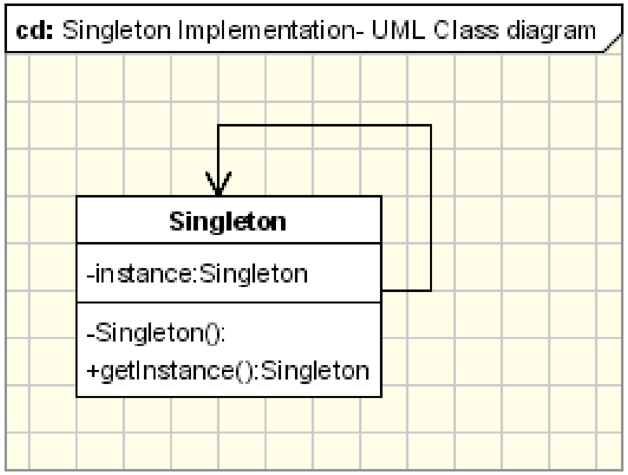
PATTERN—GoF: Singleton

- Singleton is known as a **creational** pattern - it's used to construct objects such that they can be decoupled from the implementing system.

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

- *Many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards.*

PATTERN-GoF: Singleton



Singleton class provides one point of access to create a single unique *instance* of itself using a getInstance operation.

The getInstance() operation has the responsibility of providing access to an instance of the Singleton class to the outside clients.

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.

PATTERN—GoF: Singleton

```
public class Logger {

         private static Logger uniqueInstance = null;

         private Logger() {

            // Prevents instance being created
          }

          public static Logger getInstance() {
            if(uniqueInstance == null) {
               uniqueInstance = new Logger();
            }
            return uniqueInstance;
          }

} // end of class
```

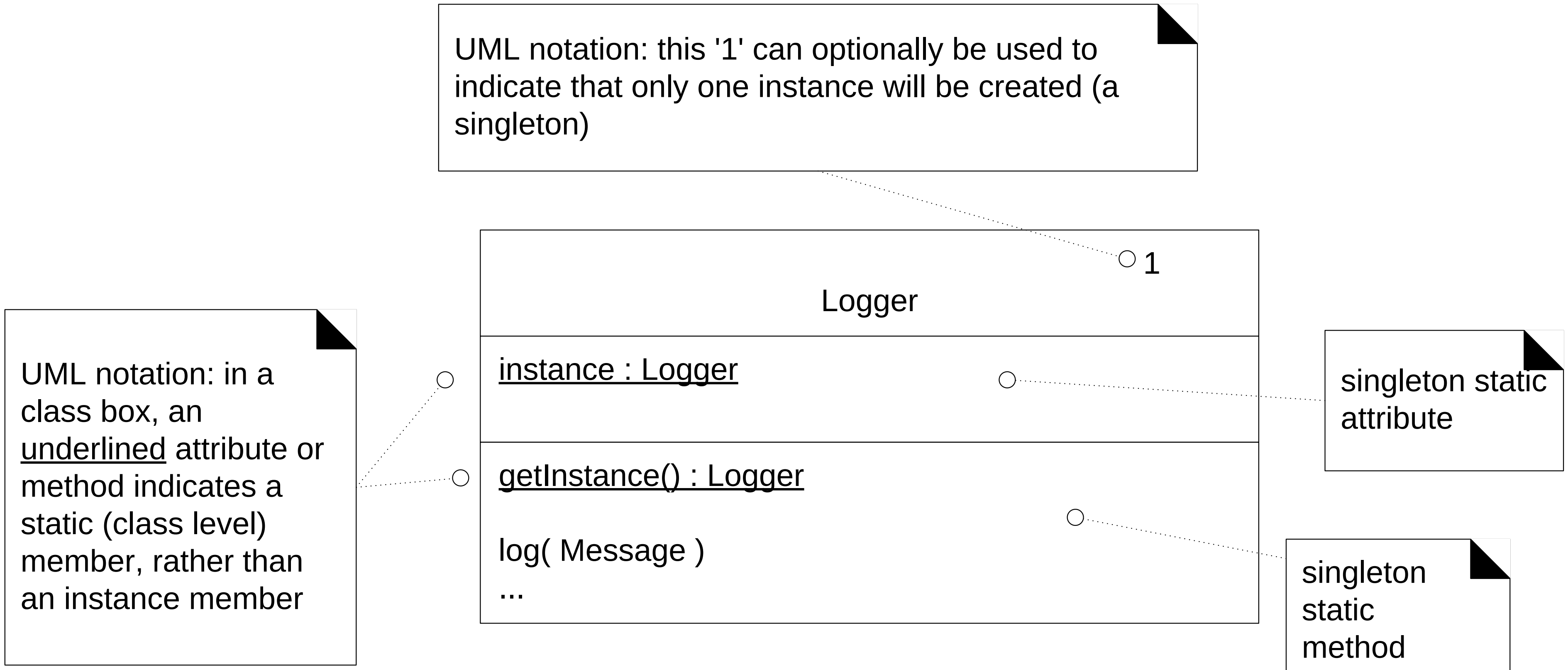
Singleton

~~Logger l = Logger.getInstance();~~

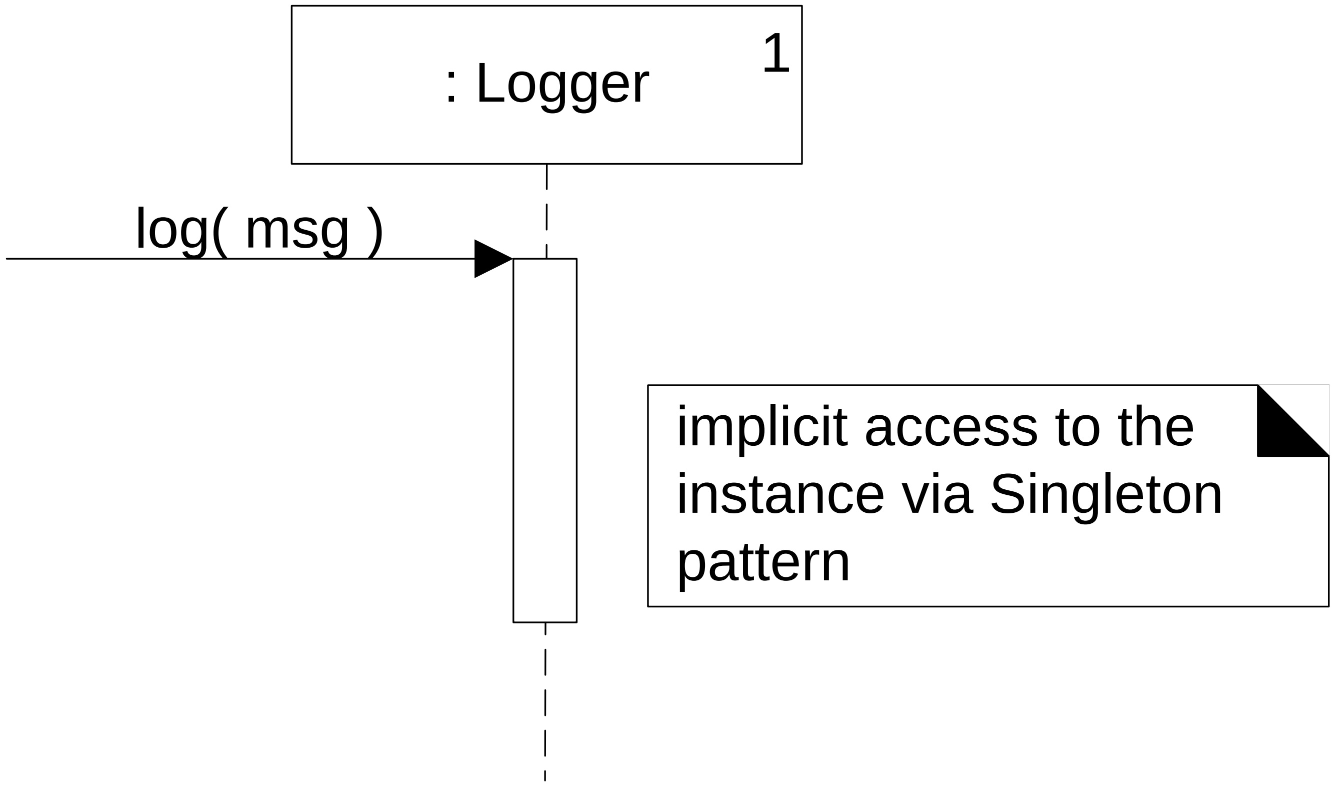
Pattern Discipline
The client class's code should access the instance using *getInstance* each time.

It should not save its own reference to that instance.

UML: Singleton

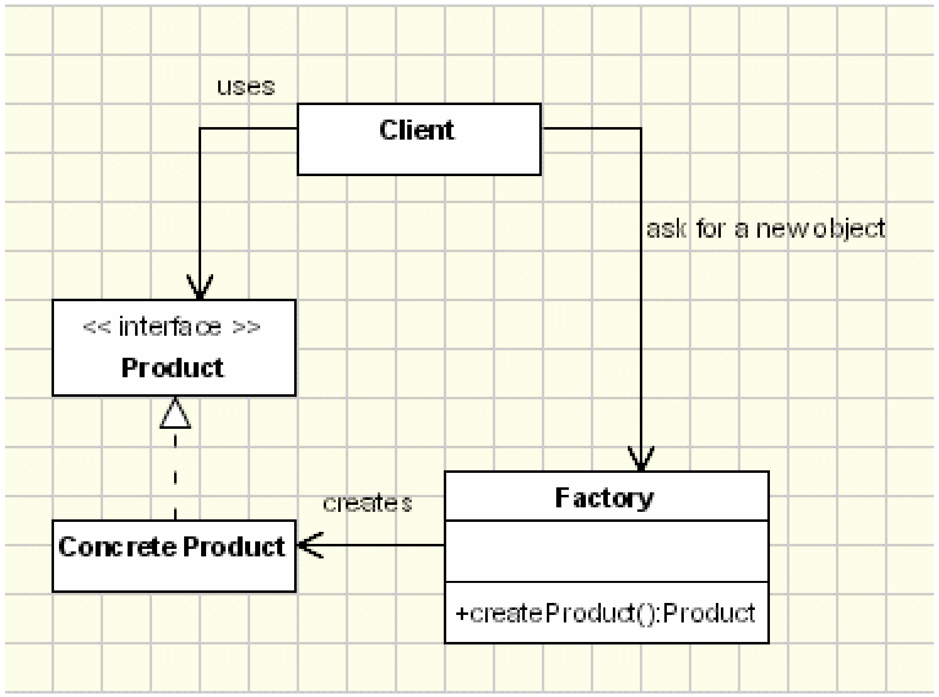

UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)
1
Logger
instance : Logger
getInstance() : Logger
log(Message)
...
UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member
singleton static attribute
singleton static method

UML: Singleton



Factory Pattern

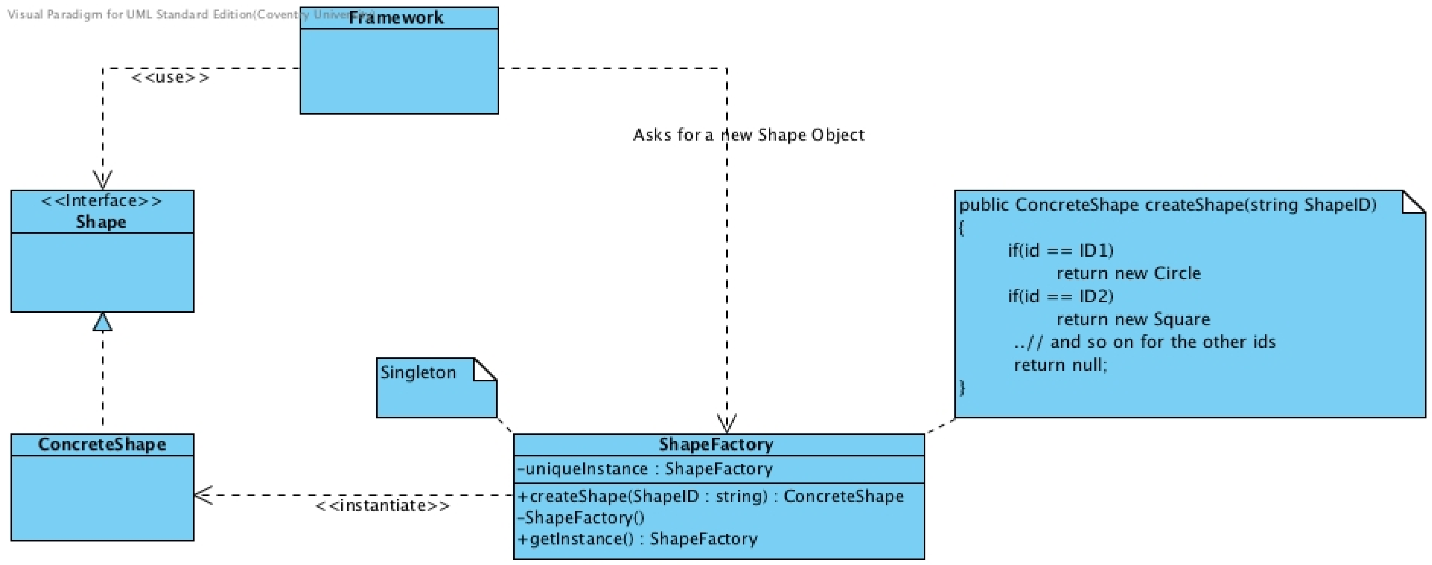
- Construct objects without exposing the instantiation logic to the client.
- Refers to the newly created object through a common interface



- The client needs a product, but instead of creating it directly using the **new** operator, it asks the factory object for a new product, providing the information about the type of object it needs.
- The client uses the products as abstract products without being aware about their concrete implementation.

Factory Pattern

For example a graphical application works with shapes. In our implementation the drawing framework is the client and the shapes are the products. All the shapes are derived from a shape interface. The Shape interface defines the draw and move operations which must be implemented by the concrete shapes. Let's assume a command is selected from the menu to create a new Circle. The framework receives the shape type as a string parameter, it asks the factory to **create** a new shape sending the parameter received from menu. The factory creates a new circle (*which implements Shape*) and returns it to the framework. Then the framework uses the interface (*calling necessary methods on the interface*) without being aware of the concrete object type.



Visual Paradigm for UML Standard Edition(Coventry University)
Framework
<<use>>
Asks for a new Shape Object
<<Interface>>
Shape
ConcreteShape
Singleton
<<instantiate>>
ShapeFactory
-uniqueInstance : ShapeFactory
+createShape(ShapeID : string) : ConcreteShape
-ShapeFactory()
+getInstance() : ShapeFactory

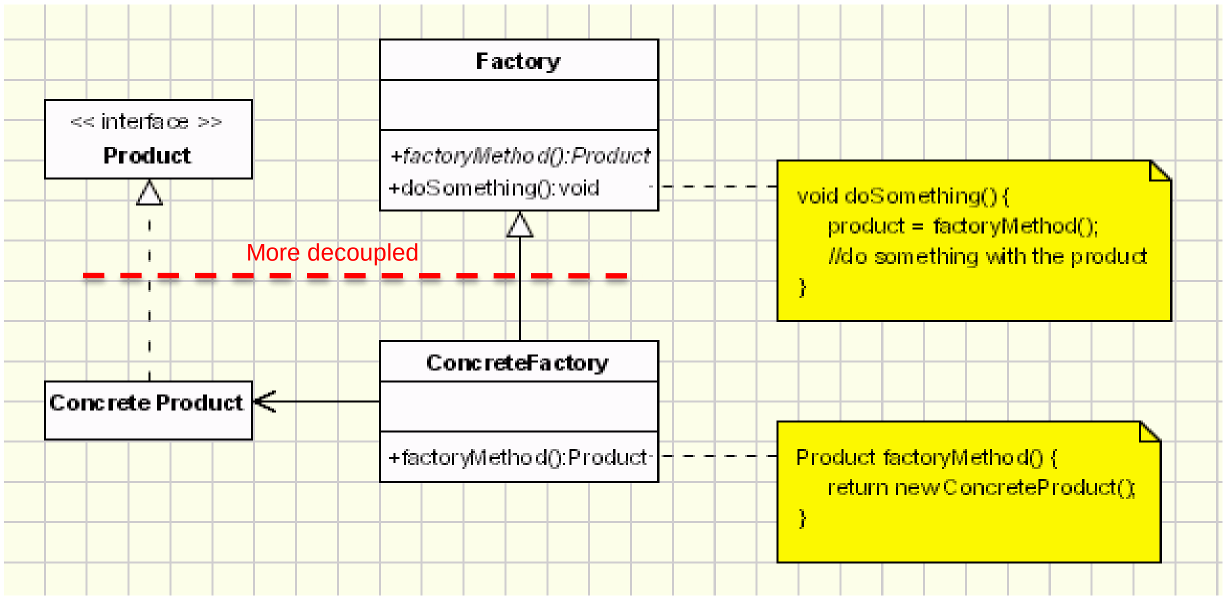
```
public ConcreteShape createShape(string ShapeID)
{
        if(id == ID1)
                return new Circle
        if(id == ID2)
                return new Square
         ..// and so on for the other ids
         return null;
}
```

The problem, adding a new concrete product call requires modifying the factory class.

Full GoF Pattern: Factory Method Pattern

- ***Product*** defines the interface for objects the factory method creates.

- **ConcreteProduct** implements the *Product* interface.
- **Factory** creates the *Product* objects and declares the operation ***factoryMethod***, which returns *Product* object (*May also call the generating method for creating Product objects*)
- **ConcreteFactory** *extends* **Factory** and overrides the *factoryMethod*, for creating specific ConcreteProduct objects



```
void doSomething() {
    product = factoryMethod();
    //do something with the product
}
```

```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

Extends Factory Pattern to define an abstract class for creating an object, but leaves the choice of its type to the subclasses (*creation being deferred at run-time)*

All concrete products are subclasses of the Product class, so all of them have the same basic implementation, at some extent. The Factory class specifies all standard and generic behavior of the products and when a new product is needed, it sends the creation details that are supplied by the client to the ConcreteFactory (*ConcreteFactory extends Factory*)

A Framework for Desktop Applications

A framework for desktop applications contains definitions for operations such as opening, creating and saving a document.

How can we extend the framework to be able to handle different types of documents that could be specific for different kinds of applications.

For example generating a drawing application, they need to define the *DrawingApplication* and *DrawingDocument* classes.

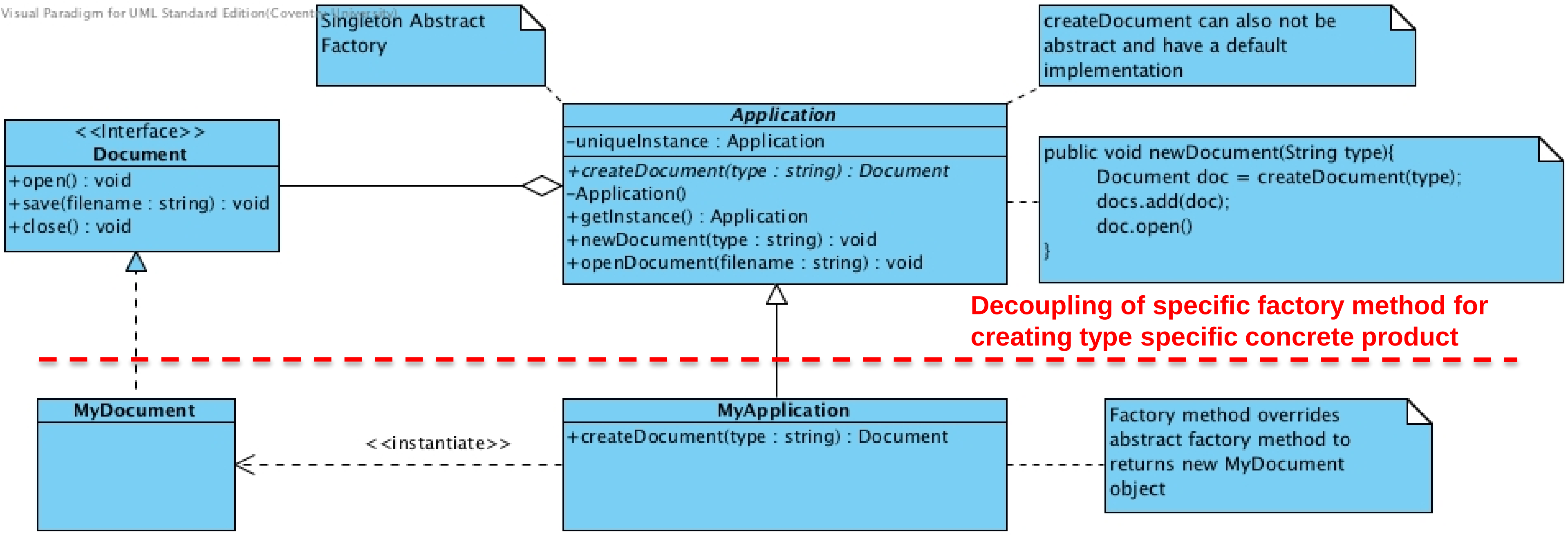
The Application class has the task of managing the documents, taking action at the request of the client (*for example, when the user selects the open or save command form the menu*).

Because the *document* class could be specific to the application, the application class does not know its type advance.

The Factory Method design pattern solves the problem by putting all the information related to the classes that need to be instantiated into specific subclasses class, so (**DrawingApplication** extends an abstract ***Application*** class is responsible for creating **DrawingDocuments** that implement the ***Document*** interface.

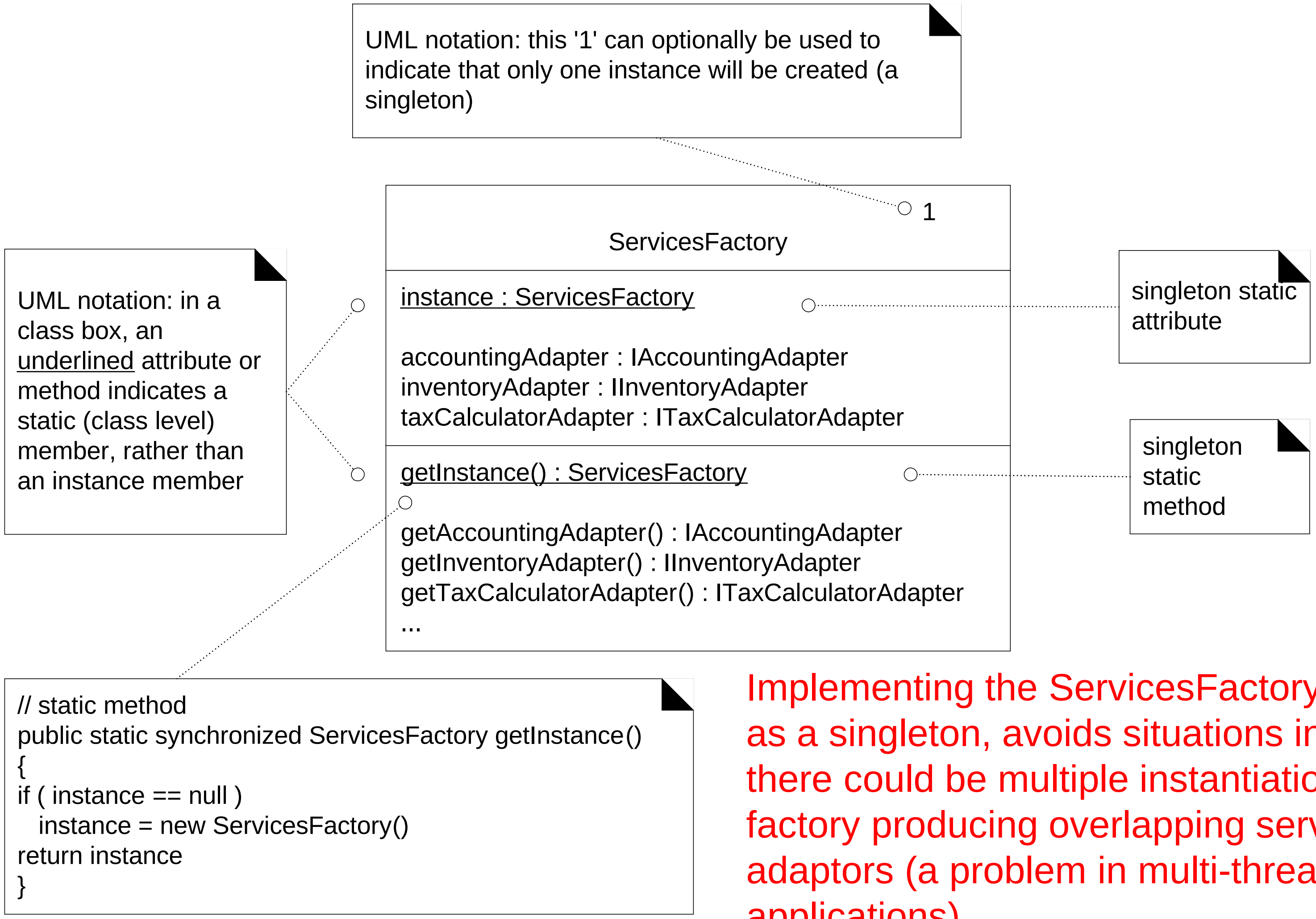
The factory is then decoupled from handling the specific types of documents produced.

A Framework for Desktop Applications


Visual Paradigm for UML Standard Edition(Coventry University)
Singleton Abstract Factory
createDocument can also not be abstract and have a default implementation
<<Interface>>
Document
+open() : void
+save(filename : string) : void
+close() : void
Application
-uniqueInstance : Application
+createDocument(type : string) : Document
-Application()
+getInstance() : Application
+newDocument(type : string) : void
+openDocument(filename : string) : void
MyDocument
<<instantiate>>
MyApplication
+createDocument(type : string) : Document
Factory method overrides abstract factory method to returns new MyDocument object
Decoupling of specific factory method for creating type specific concrete product

```
public void newDocument(String type){
        Document doc = createDocument(type);
        docs.add(doc);
        doc.open()
}
```

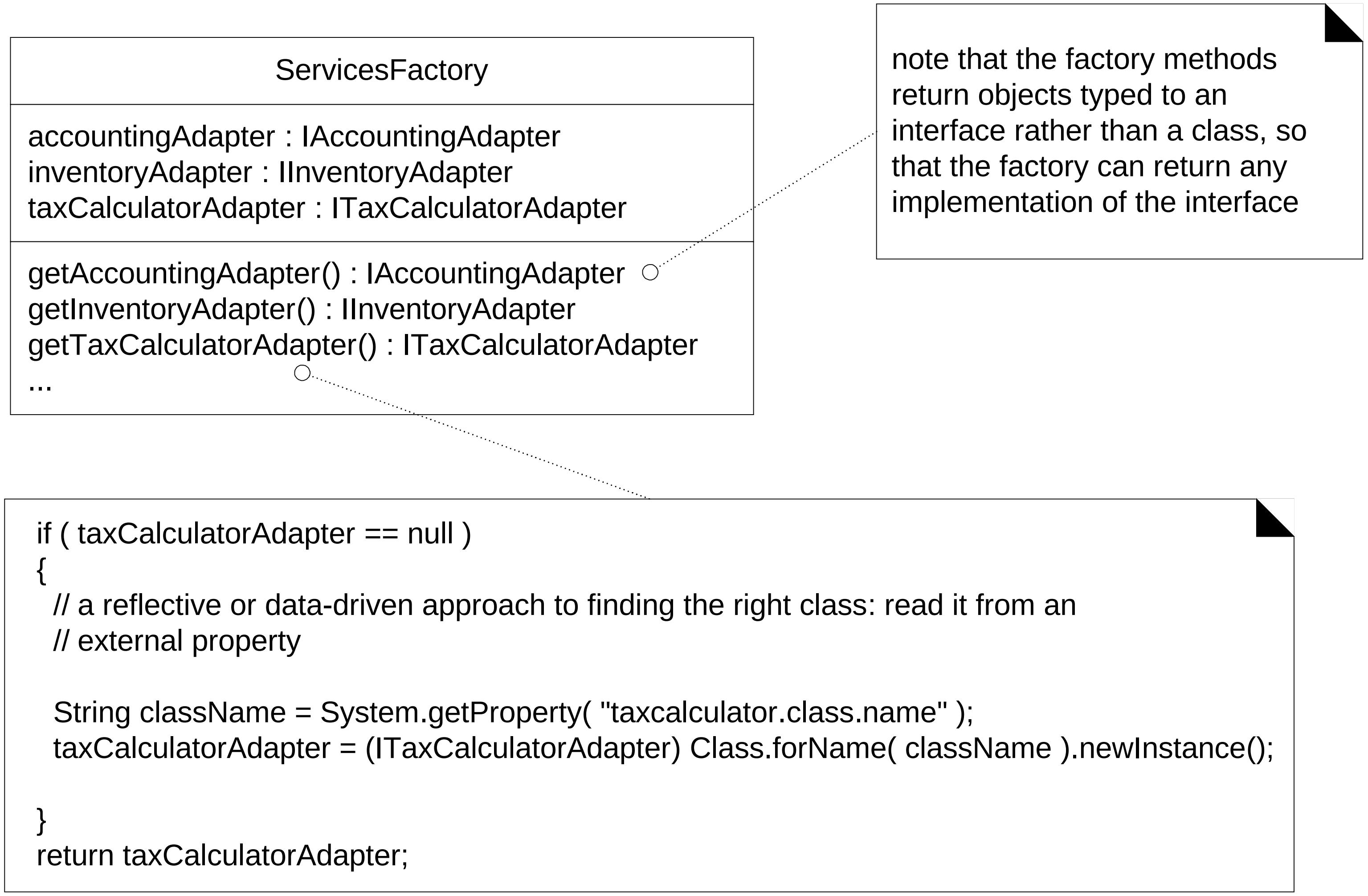
Fig. 26.6: **The Singleton Pattern** in a ServicesFactory class


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
}
```

Implementing the ServicesFactory class as a singleton, avoids situations in which there could be multiple instantiations of the factory producing overlapping service adaptors (a problem in multi-threaded applications)

The Factory Pattern Revisited

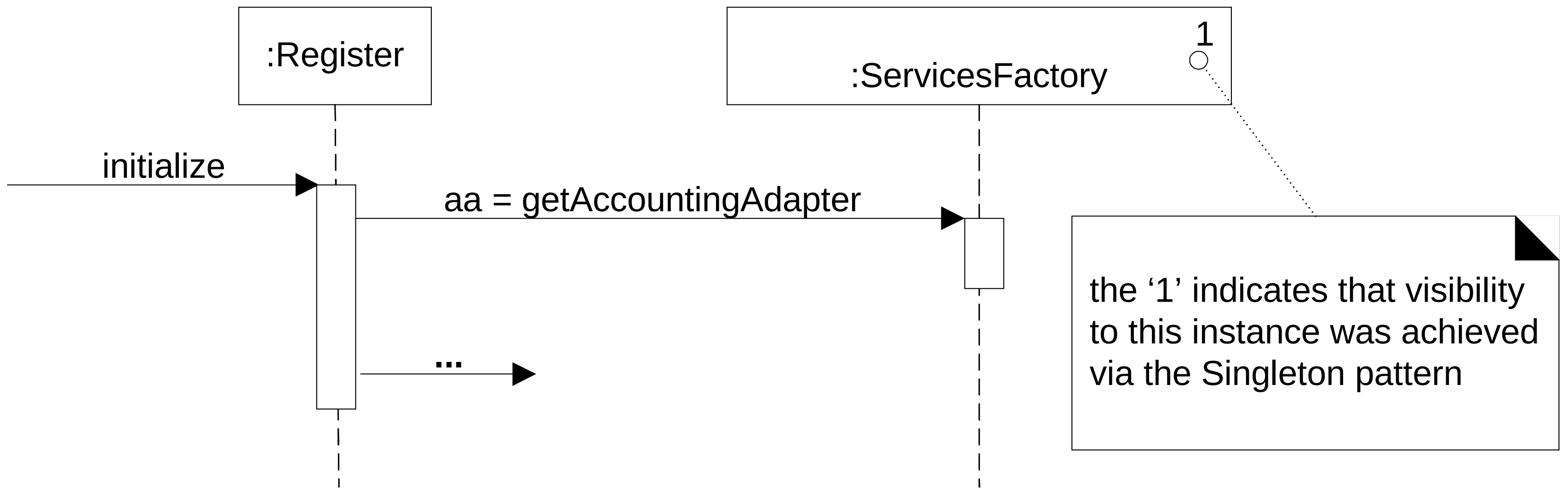

ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

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

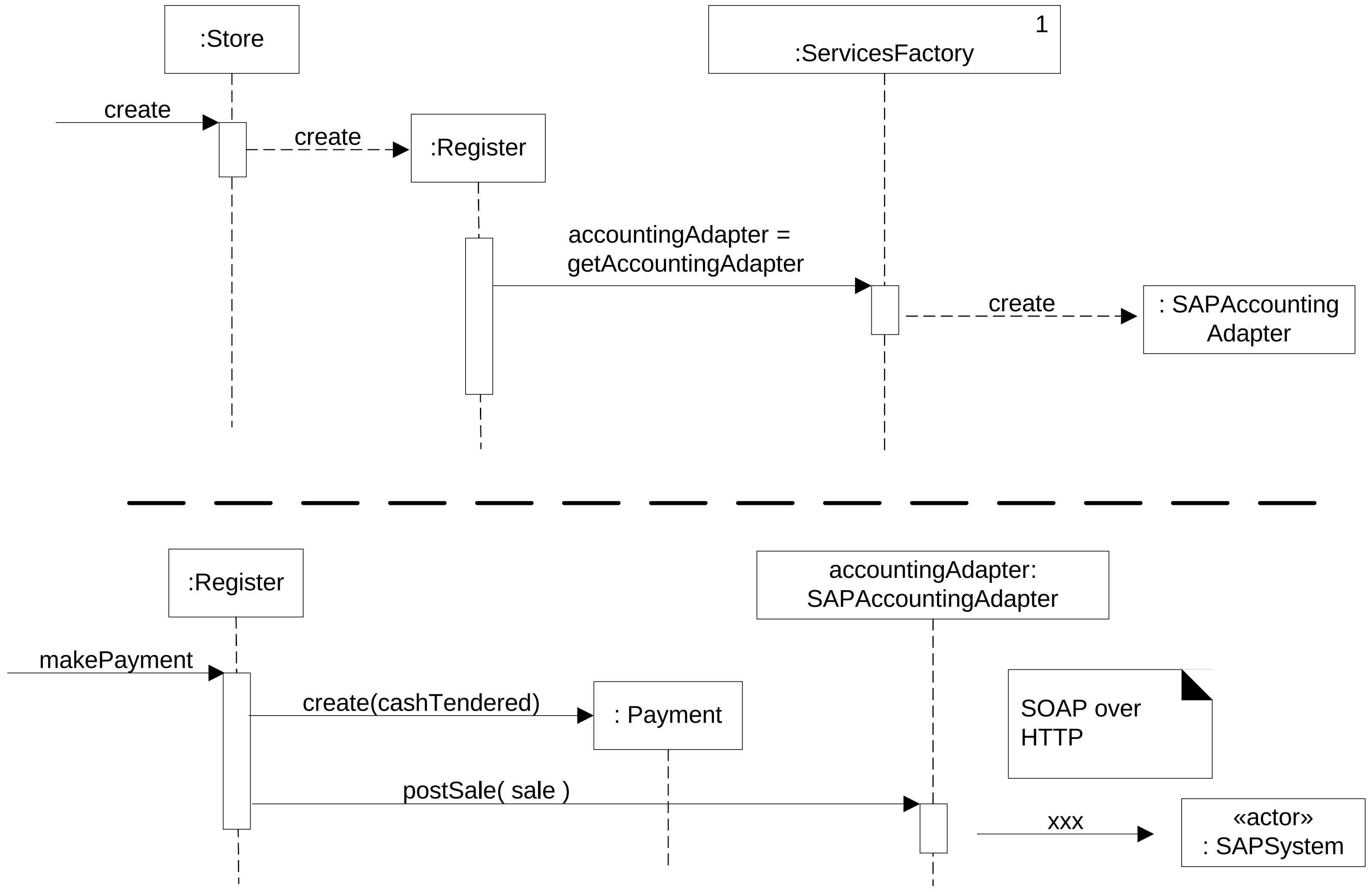
```
if ( taxCalculatorAdapter == null )
{
  // a reflective or data-driven approach to finding the right class: read it from an
  // external property

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

}
return taxCalculatorAdapter;
```

Specifies a new instance of a concrete tax collector object to be returned that implements ITaxCalculatorAdaptor

Implicit getInstance Singleton pattern message indicated in the UML because of the “1” mark

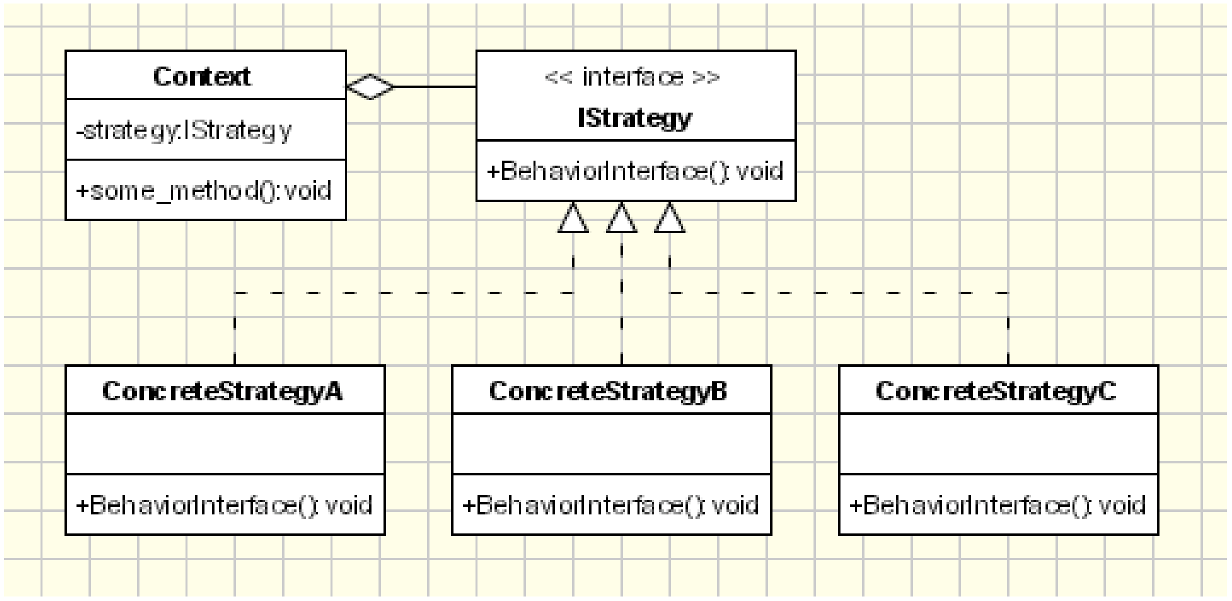

:Register
1
:ServicesFactory
initialize
aa = getAccountingAdapter
...
the ‘1’ indicates that visibility to this instance was achieved via the Singleton pattern

Adaptor, Factory, and Singleton patterns applied to Larman's case study design



PATTERN—GoF: Strategy

- *Problem*: How to design for varying, related algorithms, that may change at runtime?
- *Solution*: Define each in a separate class, with a common interface.



Context contains a reference to a strategy object. Defines an *IStrategy* interface that lets strategy access its methods. This would references the **ConcreteStrategy** that should be used.

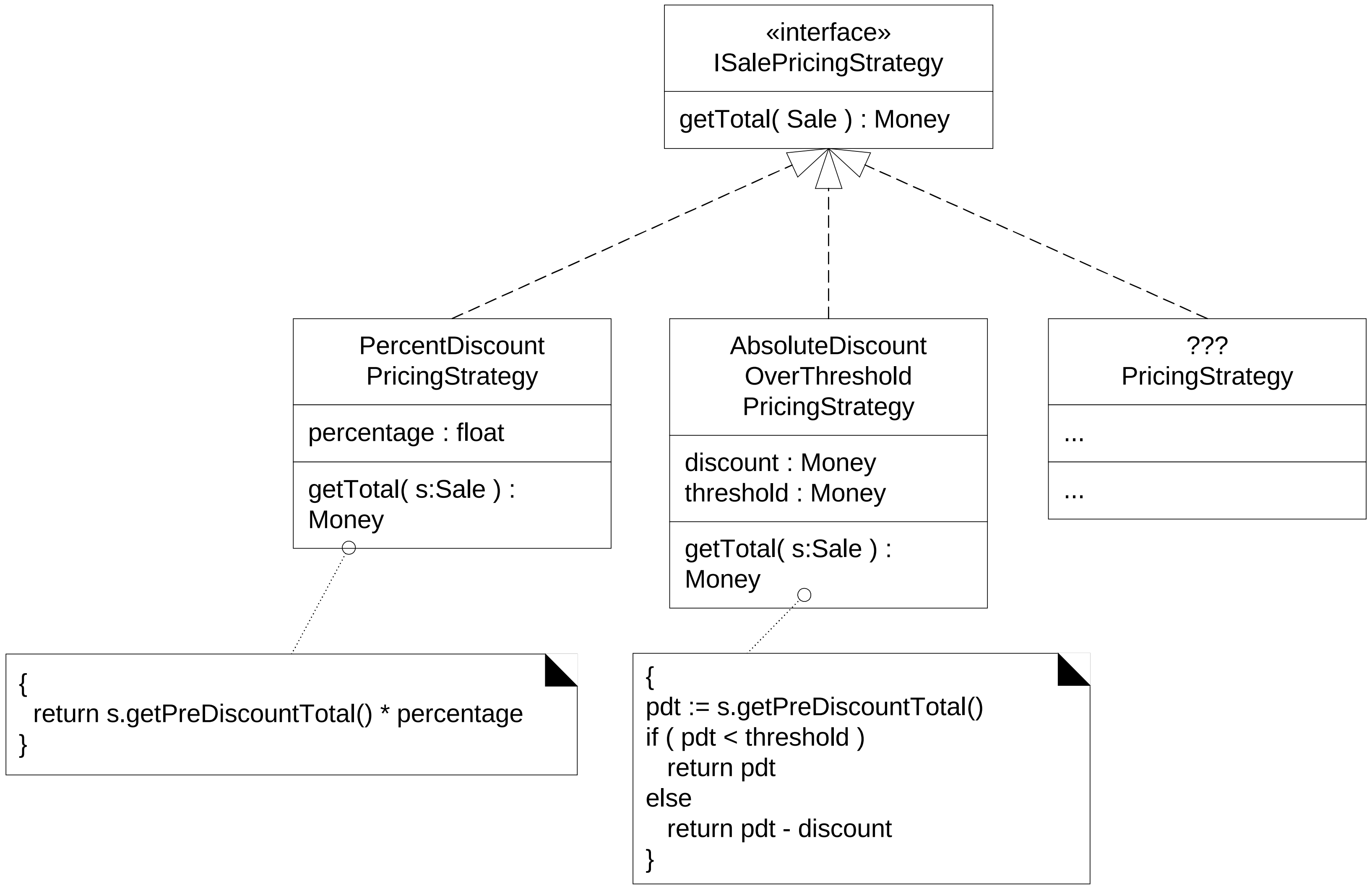
When an operation is required then the algorithm is run from the strategy object. The **Context** is not aware of the strategy implementation. If necessary, addition objects can be defined to pass data from context object to strategy.

The context object receives requests from the client and delegates them to the strategy object. Usually the **ConcreteStartegy** is created by the client and passed to the context. From this point the clients interacts only with the context.

Strategy - defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy - each concrete strategy implements an algorithm.

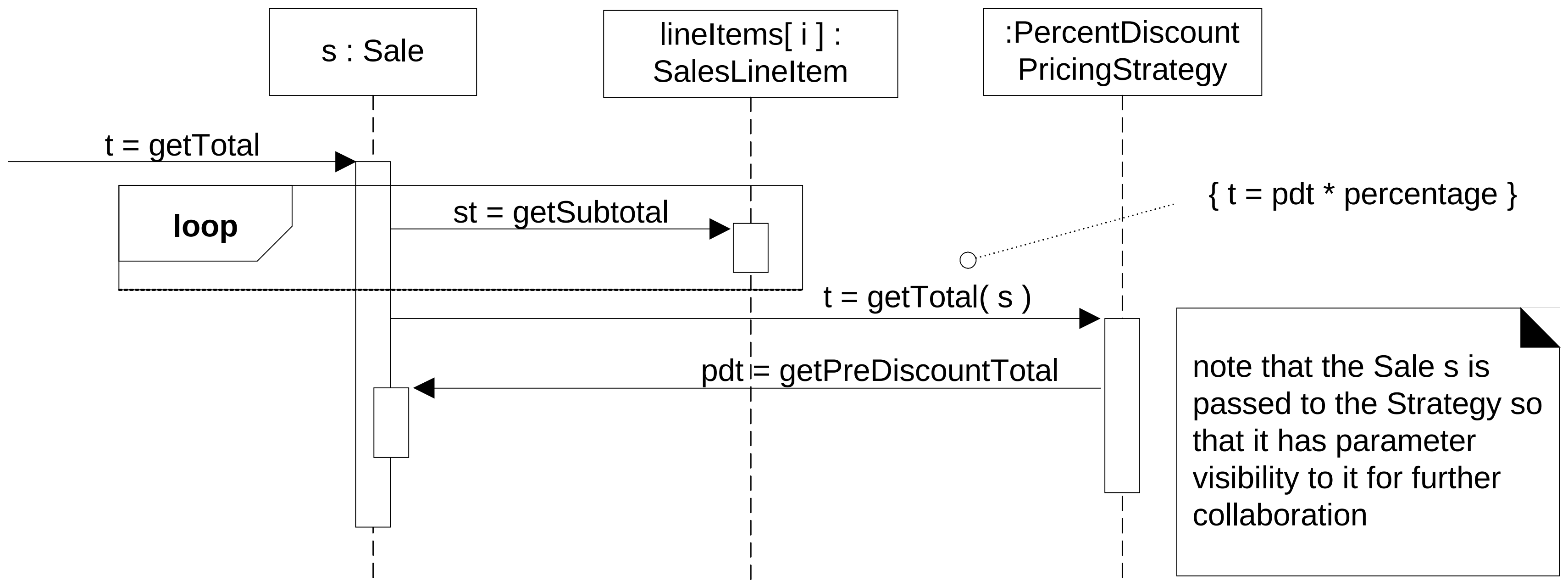
Pricing Strategy classes



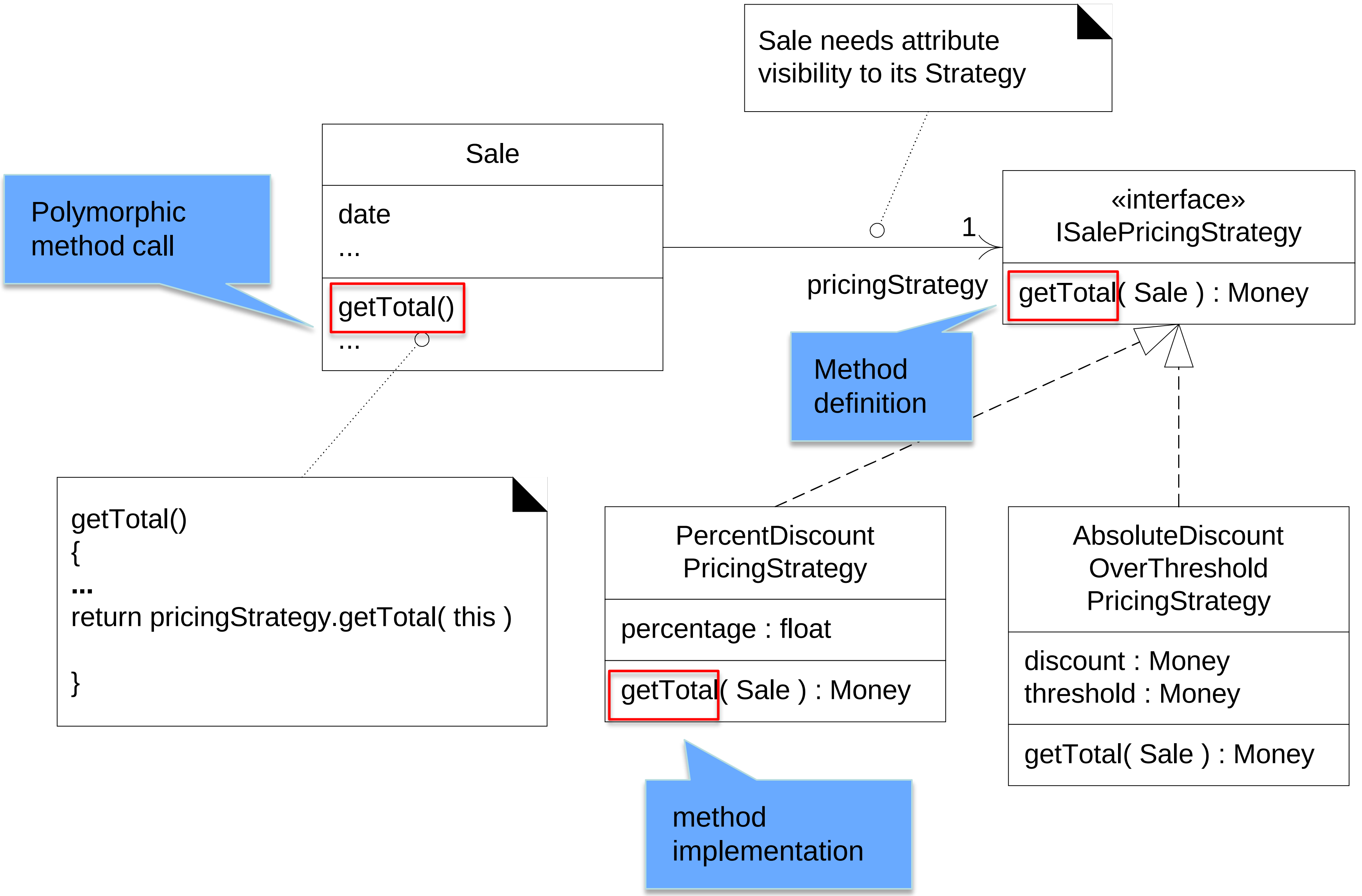
```
{
  return s.getPreDiscountTotal() * percentage
}
```

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

Strategy in collaboration

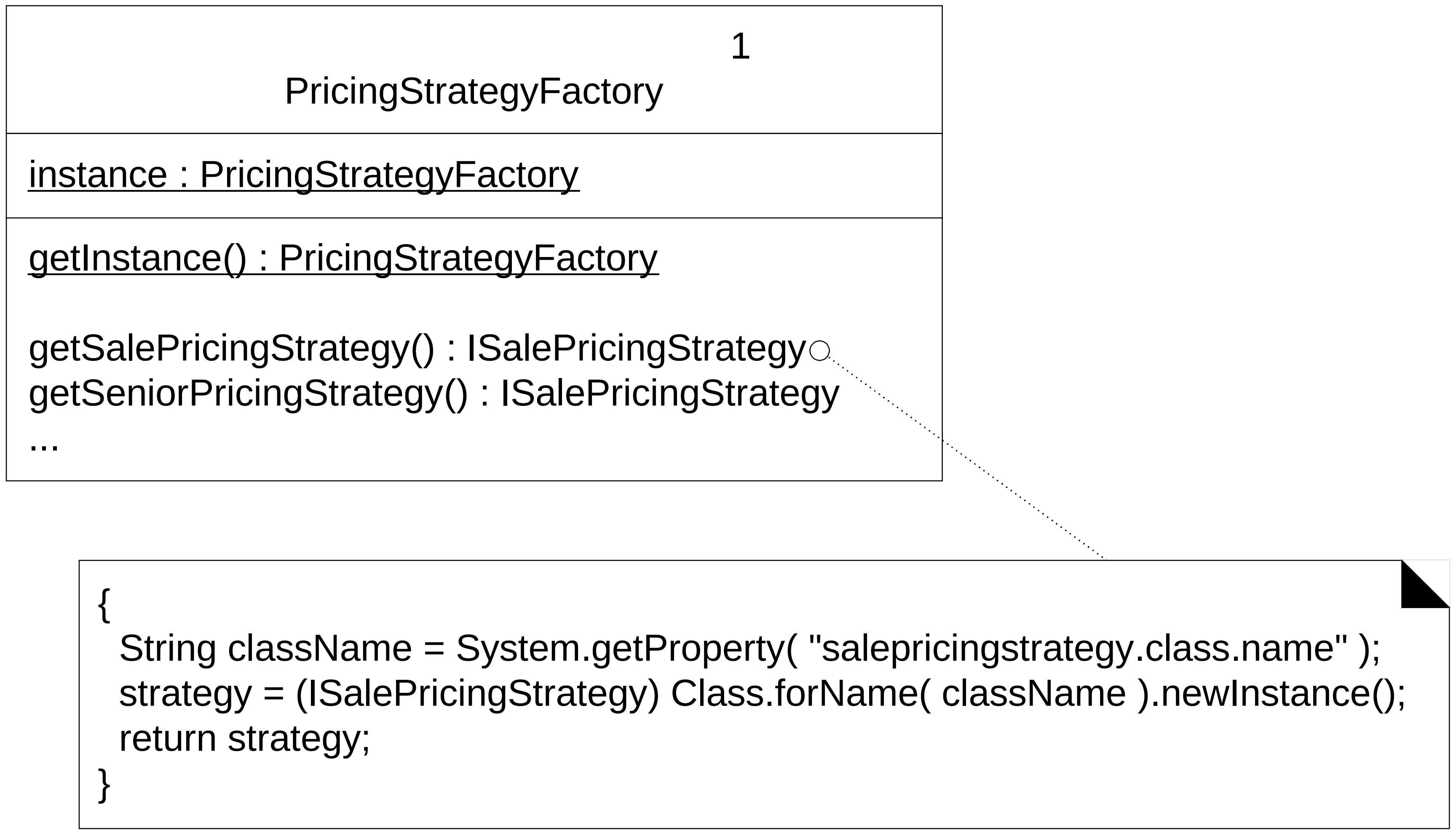

s : Sale
lineItems[i] : SalesLineItem
:PercentDiscount PricingStrategy
t = getTotal
loop
st = getSubtotal
t = getTotal(s)
pdt = getPreDiscountTotal
{ t = pdt * percentage }
note that the Sale s is passed to the Strategy so that it has parameter visibility to it for further collaboration

Context object needs attribute visibility to its strategy



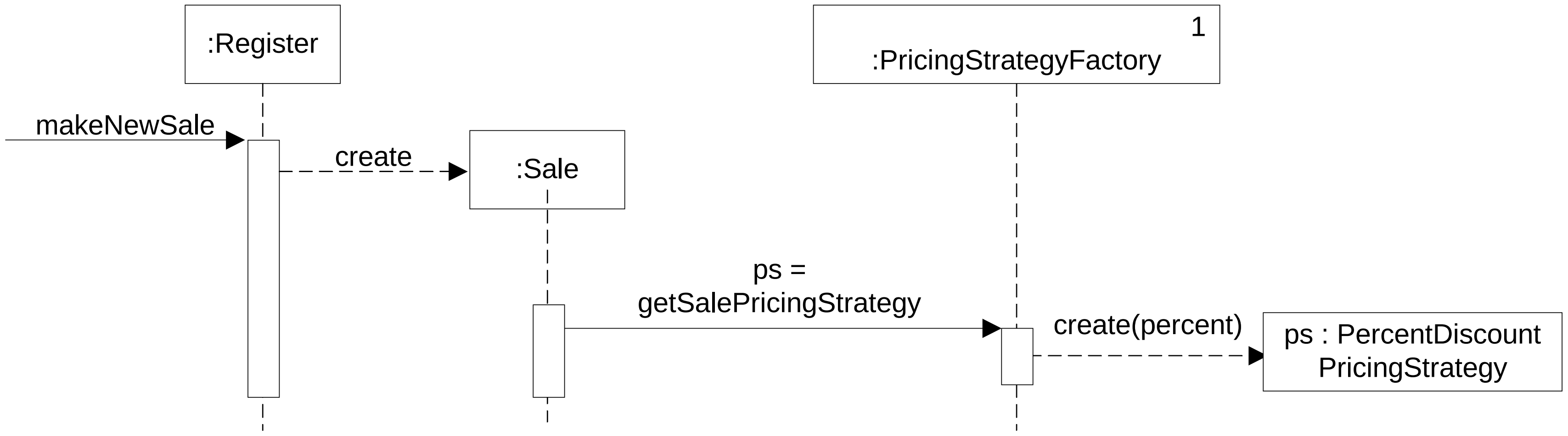
```
getTotal()
{
...
return pricingStrategy.getTotal( this )

}
```

Factory for Strategy


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;
}
```

Creating a Strategy


:Register
1
:PricingStrategyFactory
makeNewSale
create
:Sale
ps =
getSalePricingStrategy
create(percent)
ps : PercentDiscount
PricingStrategy