# Some GoF Design Patterns: Adapter, Factory, Singleton, and Strategy

Curt Clifton

Rose-Hulman Institute of Technology

Q1

http://www.research.ibm.com/designpatterns/pubs/ddj-eip-award.htm

## Gang of Four

Ralph Johnson, Richard Helm, Erich Gamma, and John Vlissides (left to right)

# GoF Pattern Taxonomy

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

# GoF Pattern Taxonomy

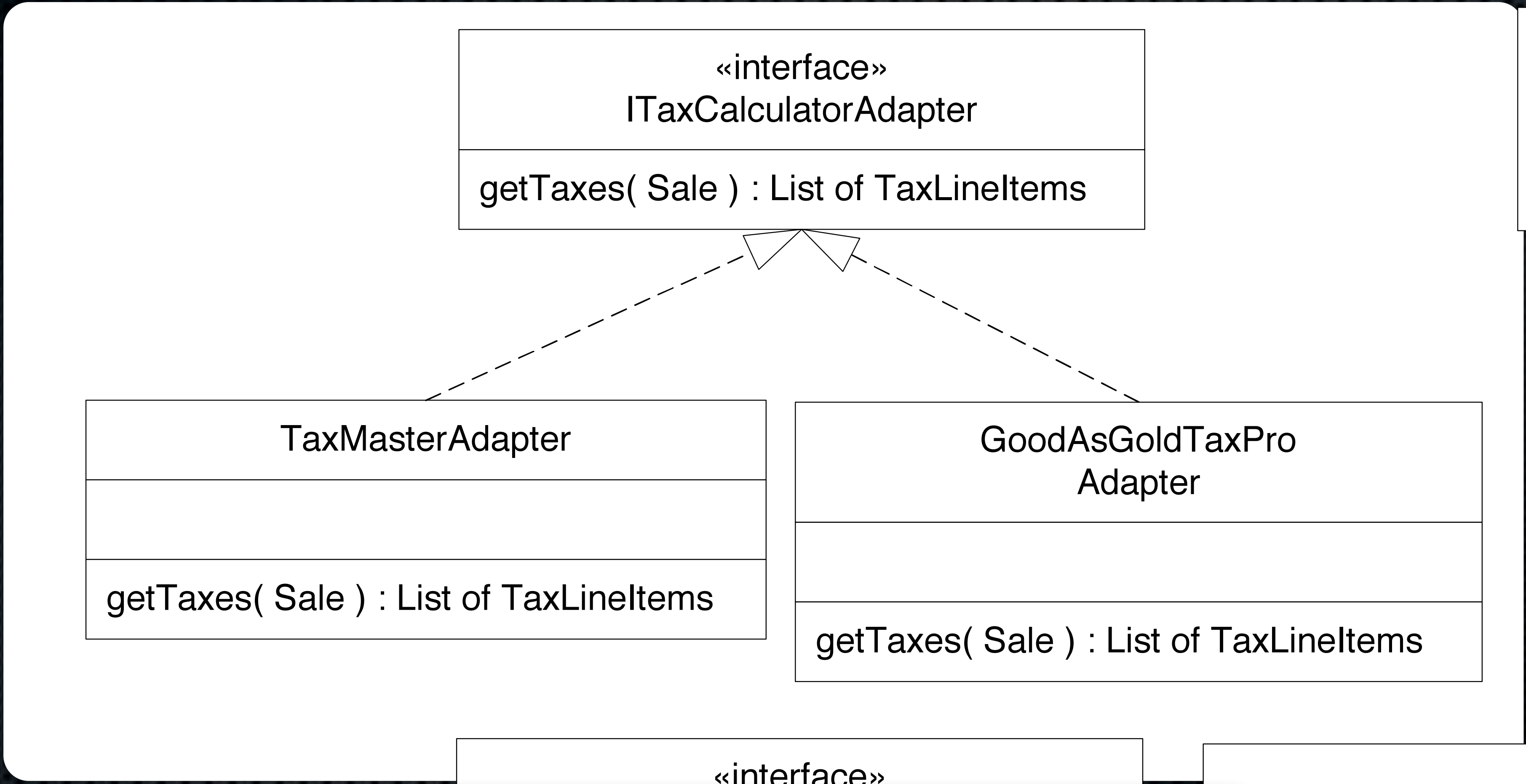
- **Behavioral**
  - Interpreter
  - Template Method
  - Chain of Responsibility
  - Command
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - **Strategy** $\leftarrow$
  - Visitor
- **Creational**
  - **Factory Method** $\leftarrow$
  - Abstract Factory
  - Builder
  - Prototype
  - **Singleton**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

## Adapter Pattern

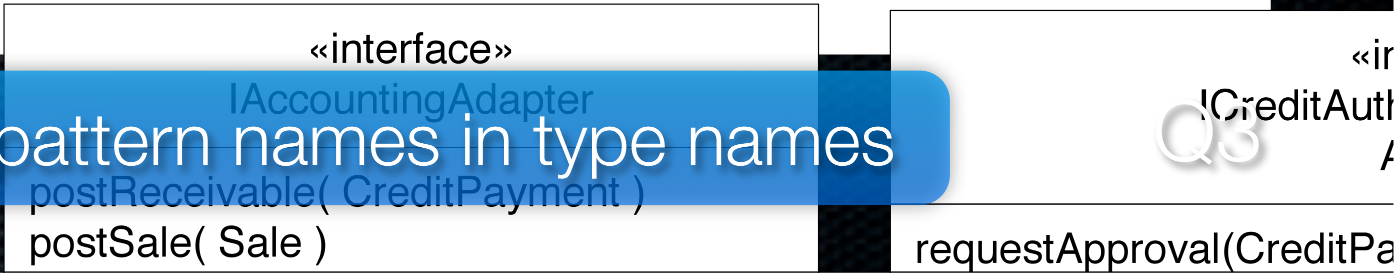
- **Problem**: How do we provide a single, stable interface to similar components with different interfaces
- **Solution**: Use an intermediate *adapter* object to convert calls to the appropriate interface for each component

Q2

# Adapter Examples


«interface»
ITaxCalculatorAdapter
getTaxes( Sale ) : List of TaxLineItems
TaxMasterAdapter
getTaxes( Sale ) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes( Sale ) : List of TaxLineItems

Adapt
polym
indire
compo



Guideline: Use pattern names in type names

## GRASP Principles in Adapter?

- Low coupling?
- High cohesion?
- Information Expert?
- Creator?
- Controller?
- Polymorphism?
- Pure Fabrication?
- Indirection?
- Protected Variations?

So why bother learning patterns?

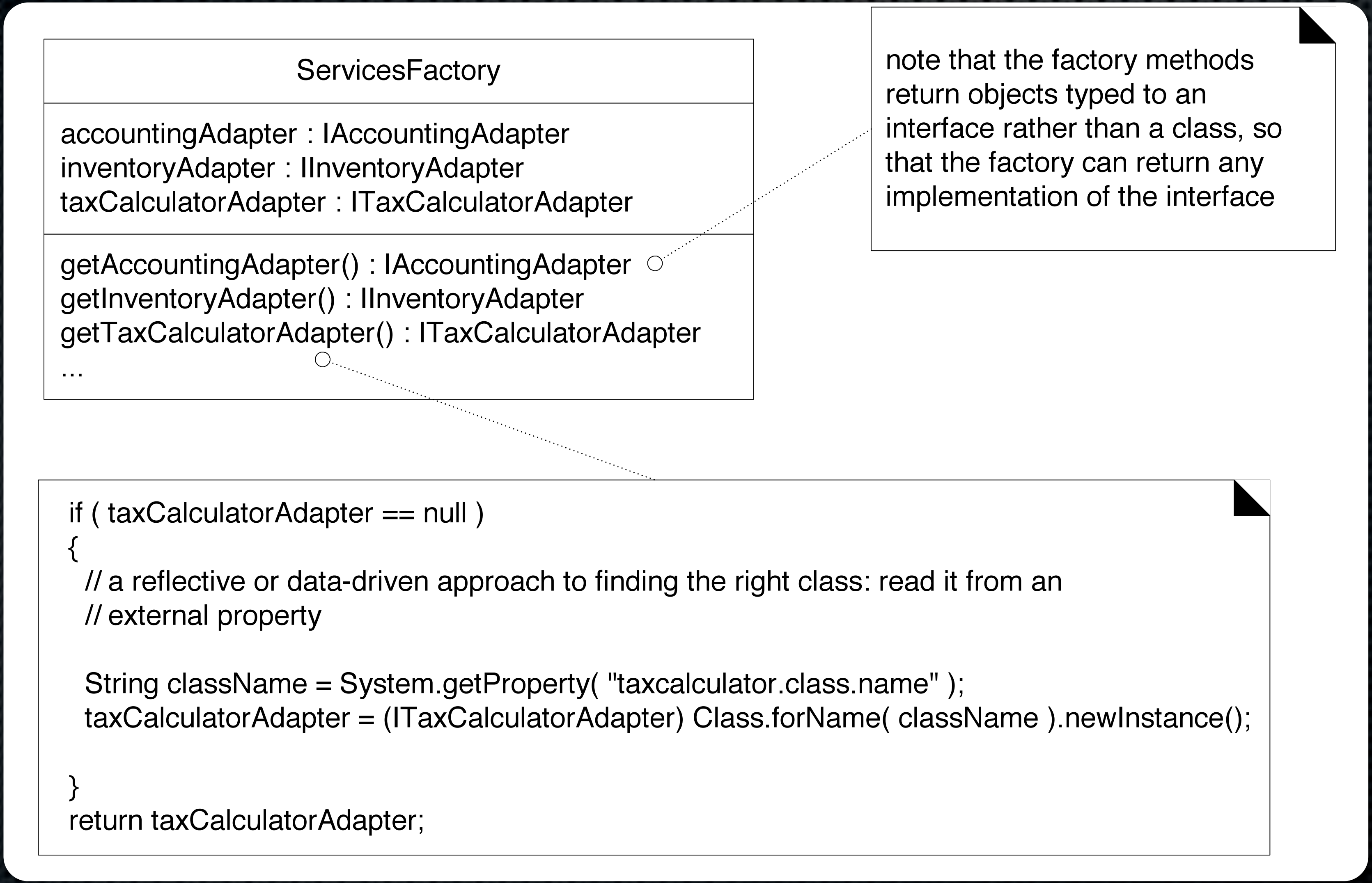
Q4

## Factory

- **Problem**: Who should be responsible for creating objects when there are special considerations like:
  - Complex creation logic
  - Separating creation to improve cohesion
  - A need for caching
- **Solution**: Create a Pure Fabrication called a Factory to handle the creation

Also known as Simple Factory or Concrete Factory

# Factory Example


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

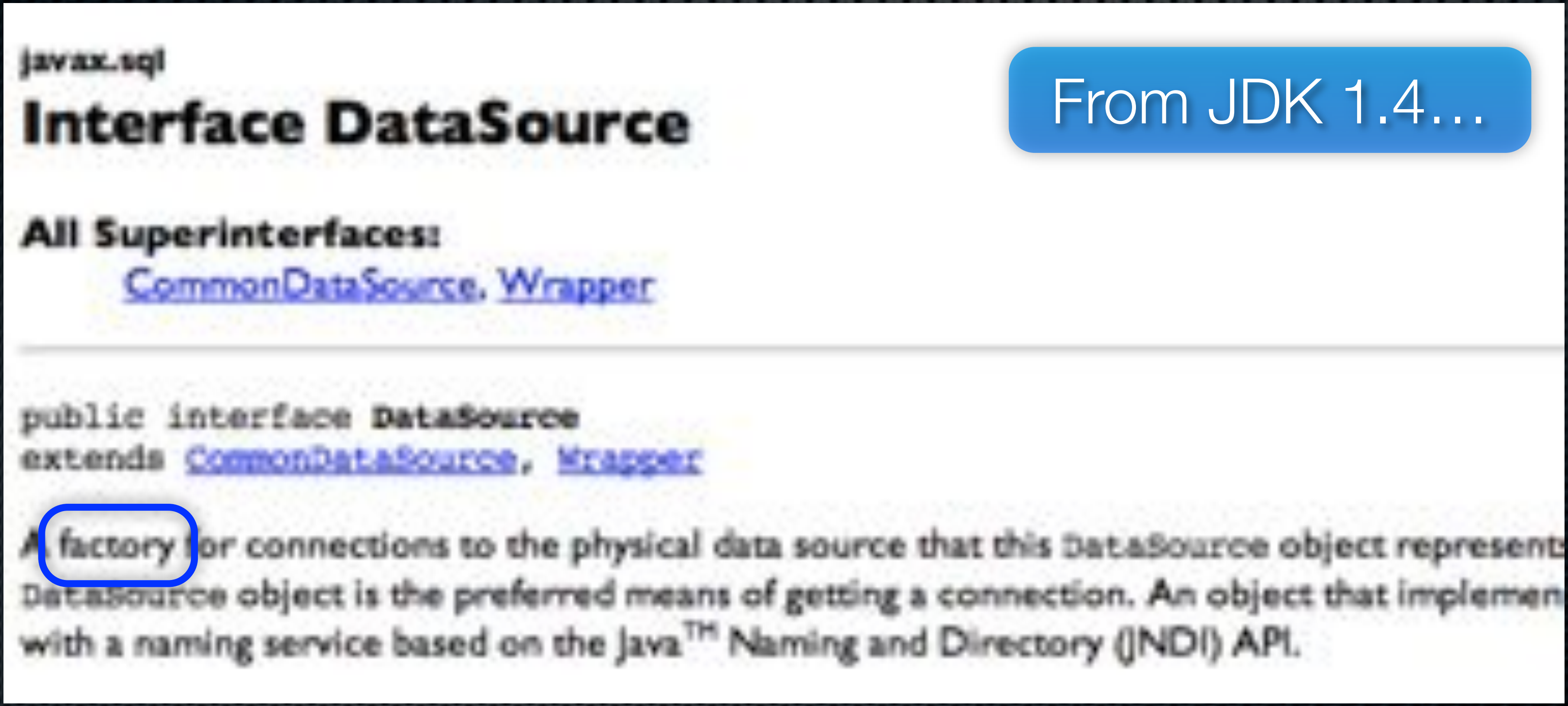
note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```
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;
```

# Another Factory Example



From JDK 1.4...

javax.sql

Interface DataSource

All Superinterfaces:
CommonDataSource, Wrapper

```
public interface DataSource
extends CommonDataSource, Wrapper
```

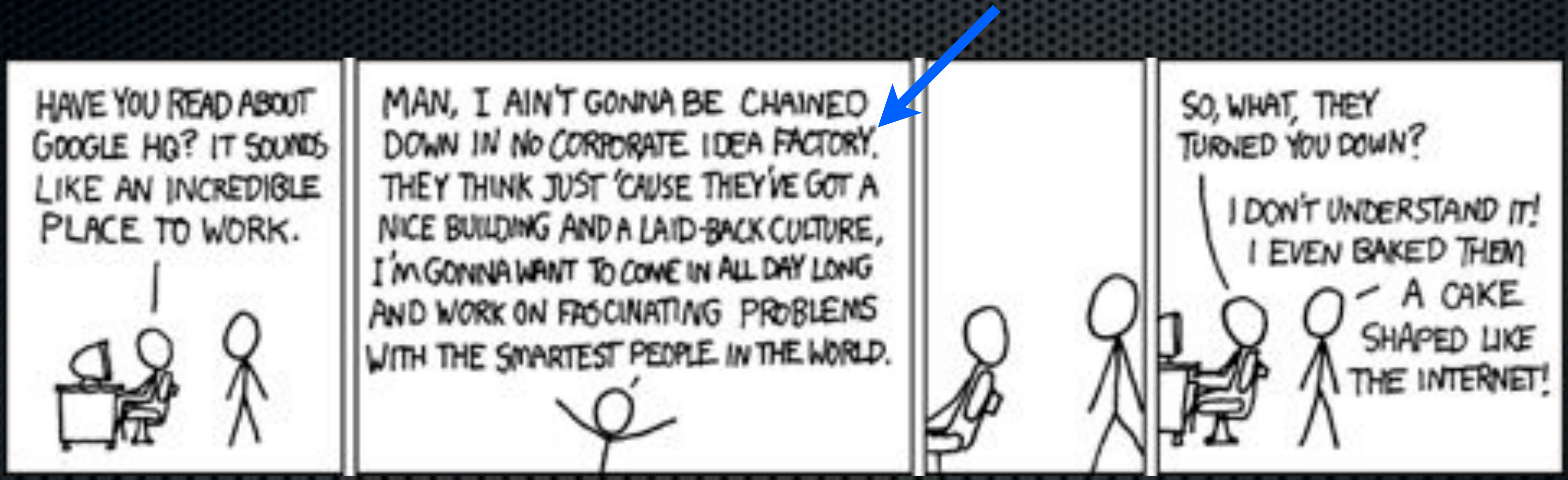
A factory for connections to the physical data source that this `DataSource` object represent
`DataSource` object is the preferred means of getting a connection. An object that implemen
with a naming service based on the Java™ Naming and Directory (JNDI) API.

## Advantages of Factory

- Puts responsibility of creation logic into a separate, cohesive class—*separation of concerns*
- Hides complex creation logic
- Allows performance enhancements:
  - Object caching
  - Recycling

Q5

## Working for Google


HAVE YOU READ ABOUT GOOGLE HQ? IT SOUNDS LIKE AN INCREDIBLE PLACE TO WORK.
MAN, I AIN'T GONNA BE CHAINED DOWN IN NO CORPORATE IDEA FACTORY. THEY THINK JUST 'CAUSE THEY'VE GOT A NICE BUILDING AND A LAID-BACK CULTURE, I'M GONNA WANT TO COME IN ALL DAY LONG AND WORK ON FASCINATING PROBLEMS WITH THE SMARTEST PEOPLE IN THE WORLD.
SO, WHAT, THEY TURNED YOU DOWN?
I DON'T UNDERSTAND IT! I EVEN BAKED THEM A CAKE SHAPED LIKE THE INTERNET!

http://xkcd.com/192/

I hear once you've worked there for 256 days they teach you the secret of levitation.

## Singleton

## Who creates the Factory?

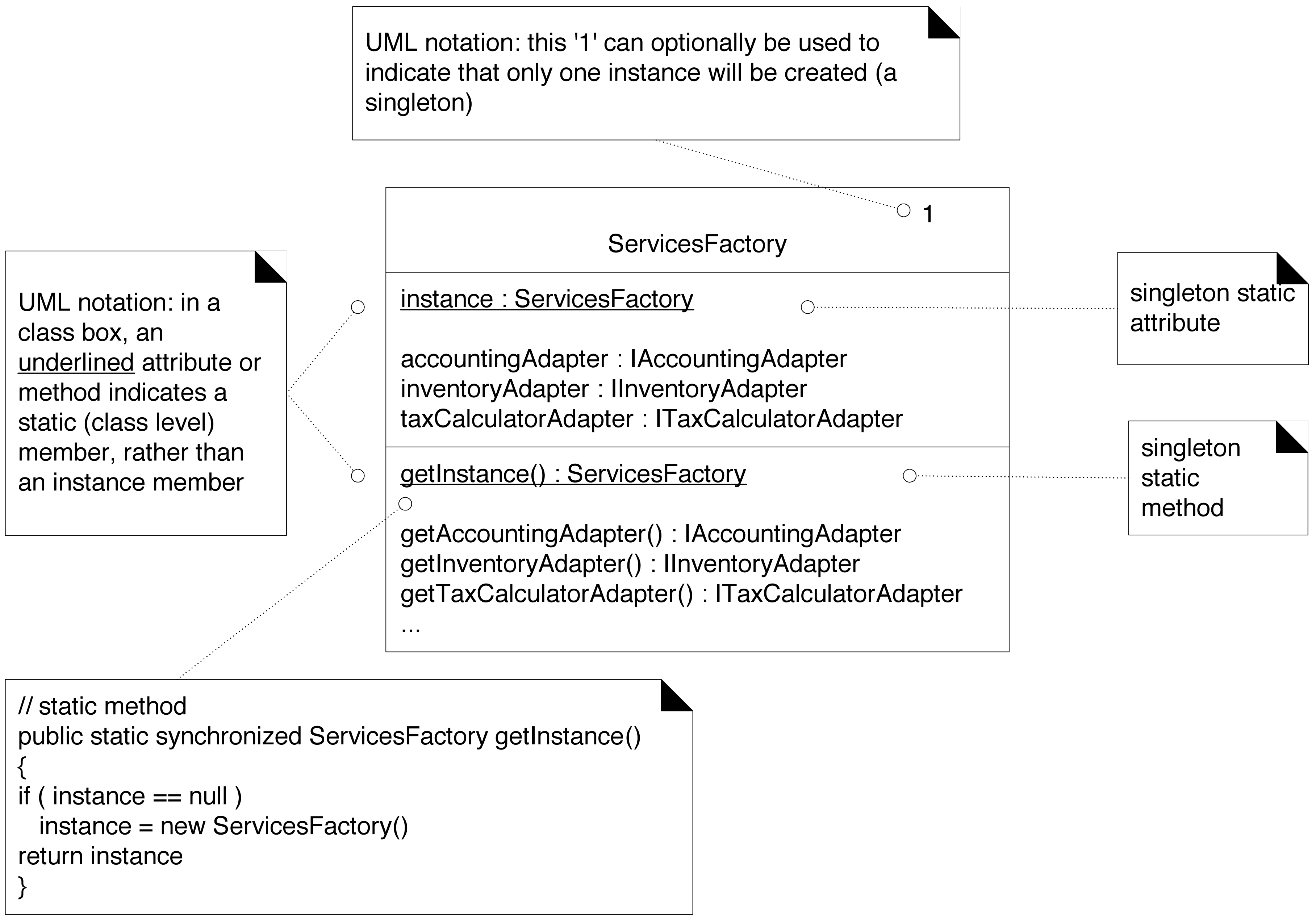
- Several classes need to access Factory methods
- Options:
    - Pass instance of Factory to classes that need it
    - Provide global visibility to a Factory instance

Dependency Injection

Singleton

## Singleton

- **Problem**: How do we ensure that exactly one instance of a class is created and is globally accessible?
- **Solution**: Define a static method in the class that returns the *singleton* instance


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
}
```

## Lazy vs. Eager Initialization

- Lazy:

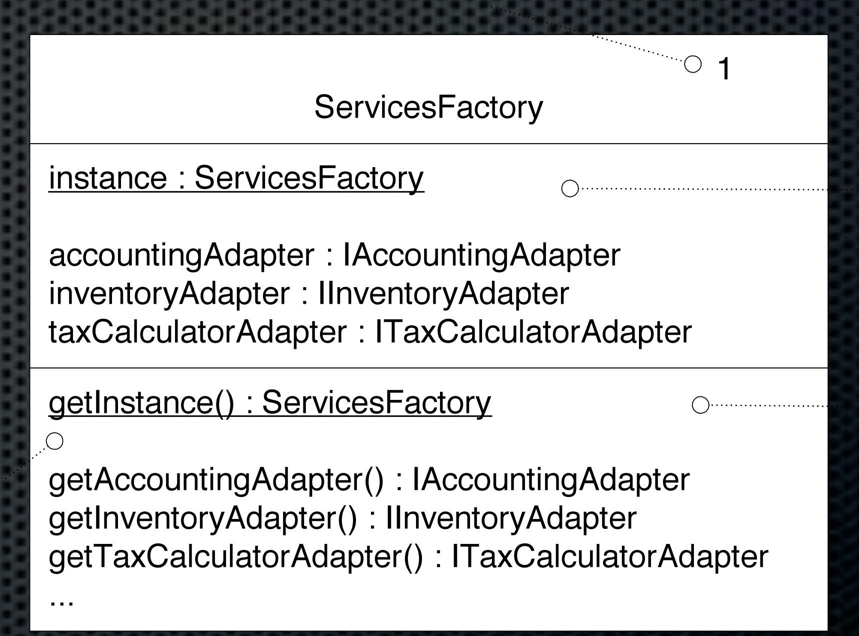
  -
    ```
    private static ServicesFactory instance;
    public static synchronized Services Factory getInstance() {
       if (instance == null)
             instance = new ServicesFactory();
       return instance;
    }
    ```
- Eager:

  -
    ```
    private static ServicesFactory instance = new ServicesFactory();
    public static Services Factory getInstance() {
       return instance;
    }
    ```

Pros and cons?

## Why don’t we just make all the methods static?

- Instance methods permit subclassing
- Instance method allow easier migration to “multi-ton” status


1
ServicesFactory
instance : ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getInstance() : ServicesFactory
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

Q6

## Singleton Considered Harmful?

Favor Dependency Injection

- Hides dependencies by introducing global visibility
- Hard to test since it introduces global state (also leaks resources)
- A singleton today is a multi-ton tomorrow
- Low cohesion — class is responsible for domain duties *and* for limiting number of instances

Instead, use Factory to control instance creation

http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx

http://tech.puredanger.com/2007/07/03/pattern-hate-singleton/

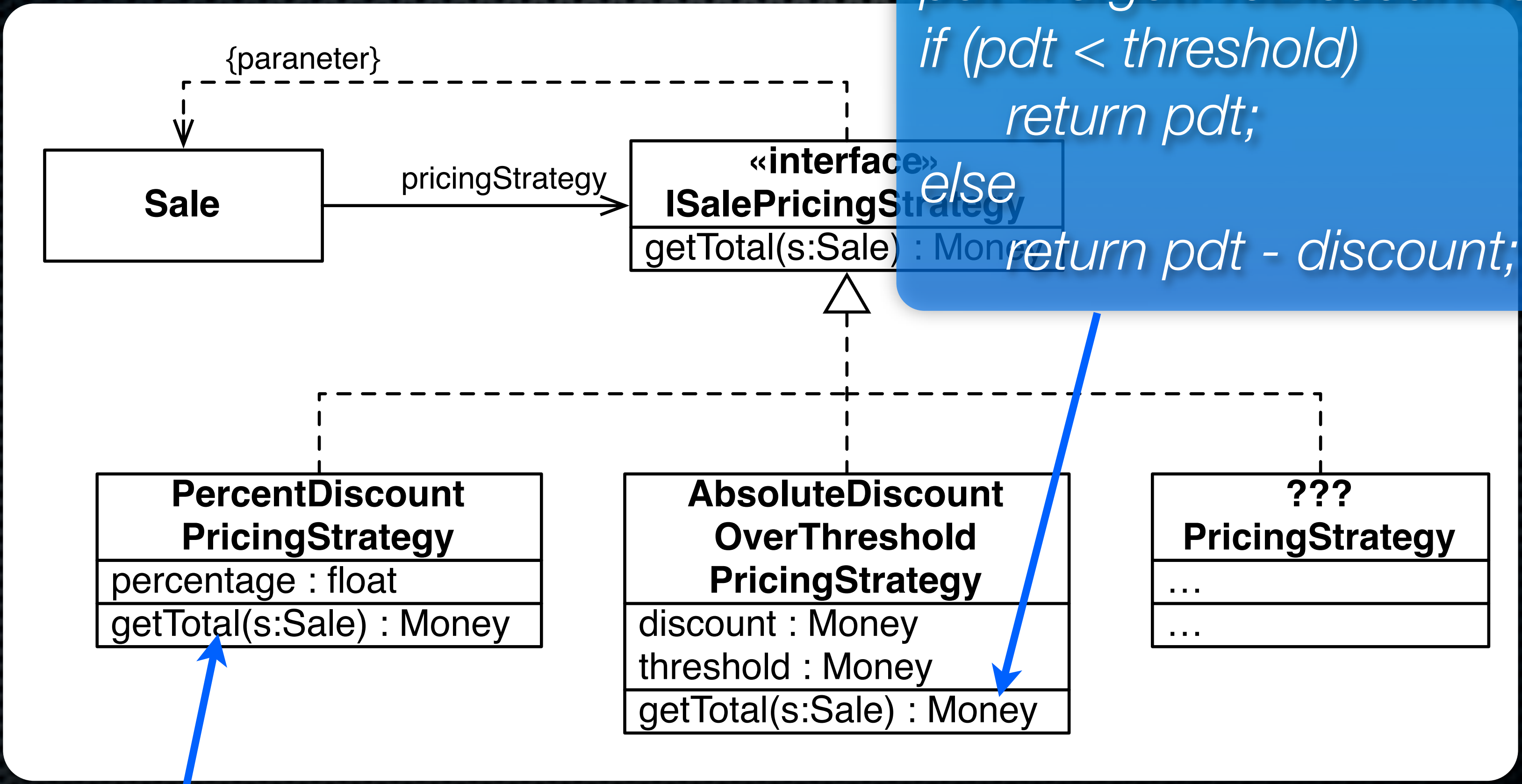
Q7

## Strategy

- **Problem**: How do we design for varying, but related, algorithms or policies?
- **Solution**: Define each algorithm or policy in a separate class with a common interface.

Q8

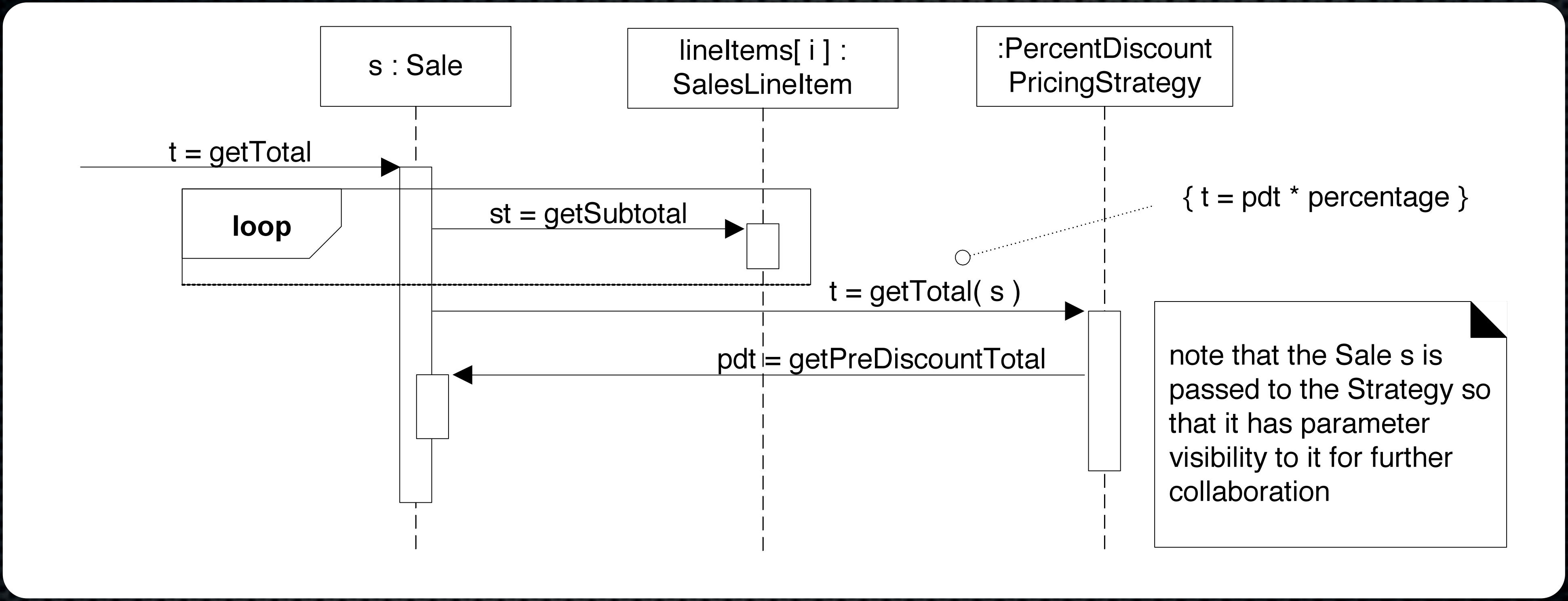
# Strategy Example



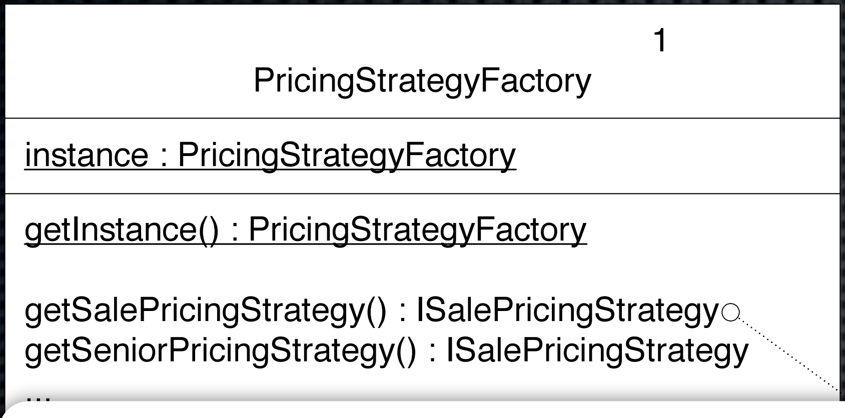
```
pdt = s.getPreDiscountTotal();
if (pdt < threshold)
     return pdt;
else
     return pdt - discount;
```

```
return s.getPreDiscountTotal() * percentage;
```

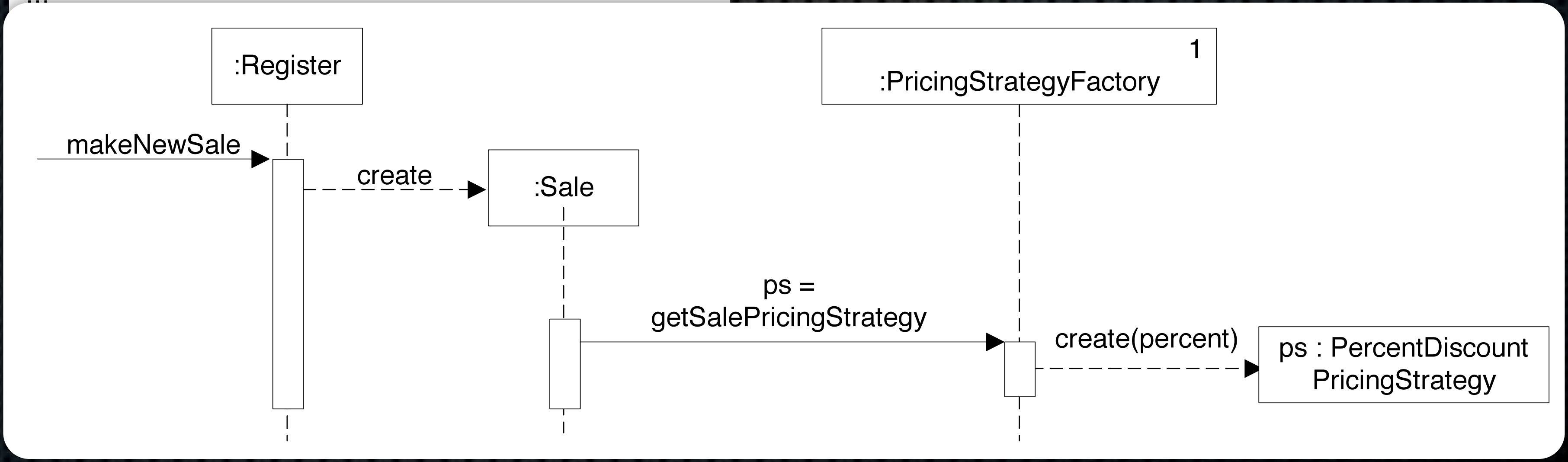
# Strategy Example (cont.)


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

# Where does the *PricingStrategy* come from?


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

What about with Dependency Injection?



Q9