Applying Some Gang of Four Design Patterns

Shawn Bohner
Office: Moench Room F212
Phone: (812) 877-8685
Email: bohner@rose-hulman.edu

ROSE-HULMAN
INSTITUTE OF TECHNOLOGY

Q1

Protected Variation

- **Problem:**
  How do we design objects and systems so that instability in them does not have undesirable effects on other elements?

- **Solution:**
  Identify points of predicted instability (variation) and assign responsibilities to create a stable interface around them

Protected Variations: Observations

- When to use it?
  - Variation point – a known area where variations in existing requirements or systems need to be supported
  - Evolution point – an anticipated area (speculative) where future variation may occur (not in current requirements)
- Investing in protection against future variation
  - How likely is it to occur? If it is, then should probably use PV now
  - If unlikely, then should probably defer using PV

Protected Variations by Other Names

- *Information hiding* [Parnas72]
  - "We propose instead that one begins with a list of difficult design decisions which are likely to change. Each module is then designed to hide such a decision from the others."
- *Open-Closed Principle* [Meyer88]
  - "Modules should be both open (for extension …) and closed (… to modification[s] that affect clients)"

Gang of Four (GoF)

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

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

Gang of Four Patterns

Behavioral

- *Interpreter*
- *Template Method*
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- ✓ Strategy
- Visitor

Creational

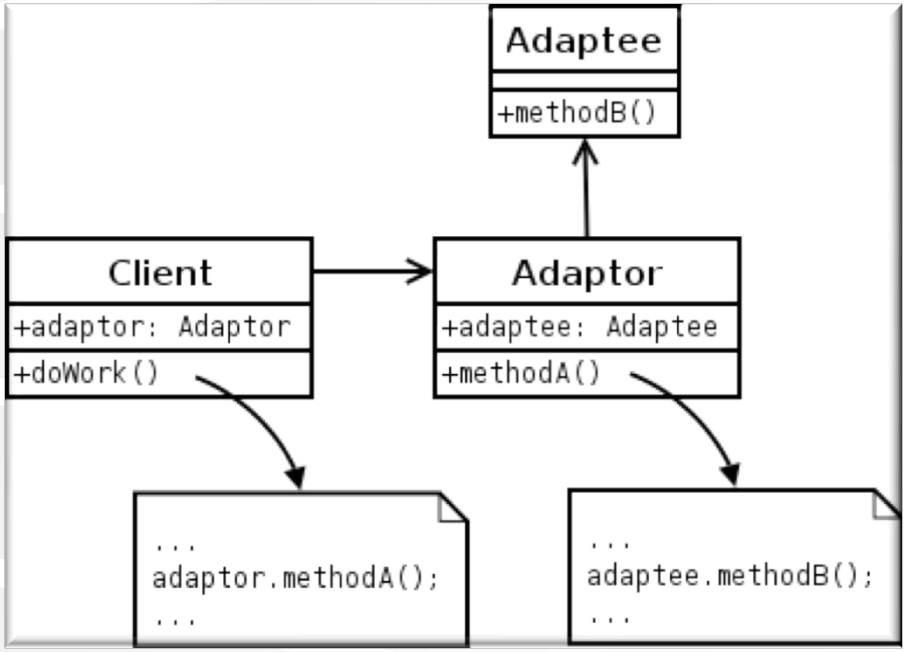
- *Factory Method*
- ✓ Abstract Factory
- Builder
- Prototype
- ✓ Singleton

Structural

- *Adapter*
- Bridge
- ✓ Composite
- Decorator
- Façade
- Flyweight
- Proxy

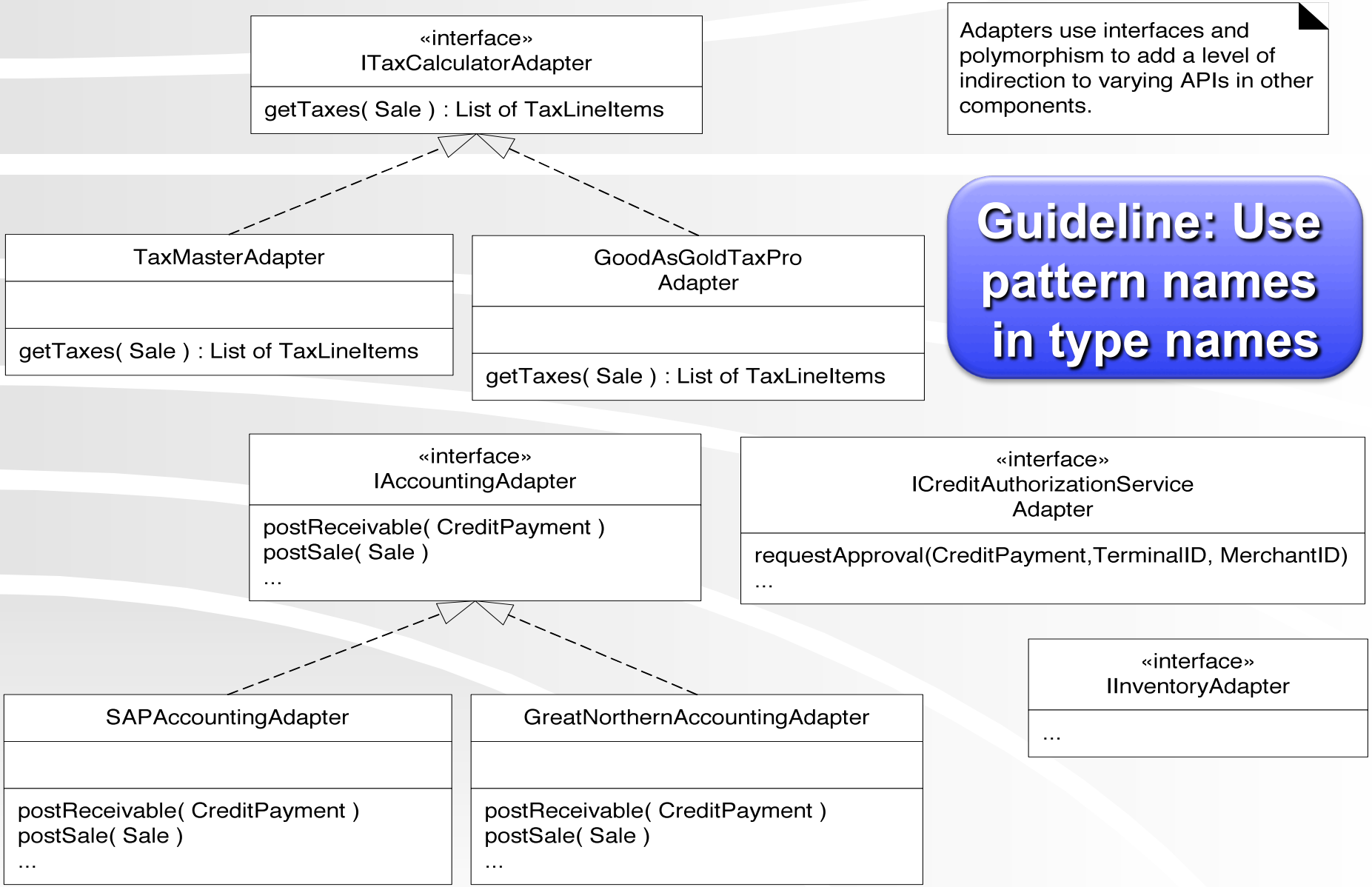
Adapter: Structural Pattern

- **Problem**: How do we provide a single, stable interface to similar components with different interfaces?
  - How do we resolve incompatible interfaces?



- **Solution**: Use an intermediate *adapter* object to convert calls to the appropriate interface for each component

Adapter Examples



GRASP Principles in Adapter?

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

So, why bother learning patterns?

Factory (Simplification of Abstract Factory)

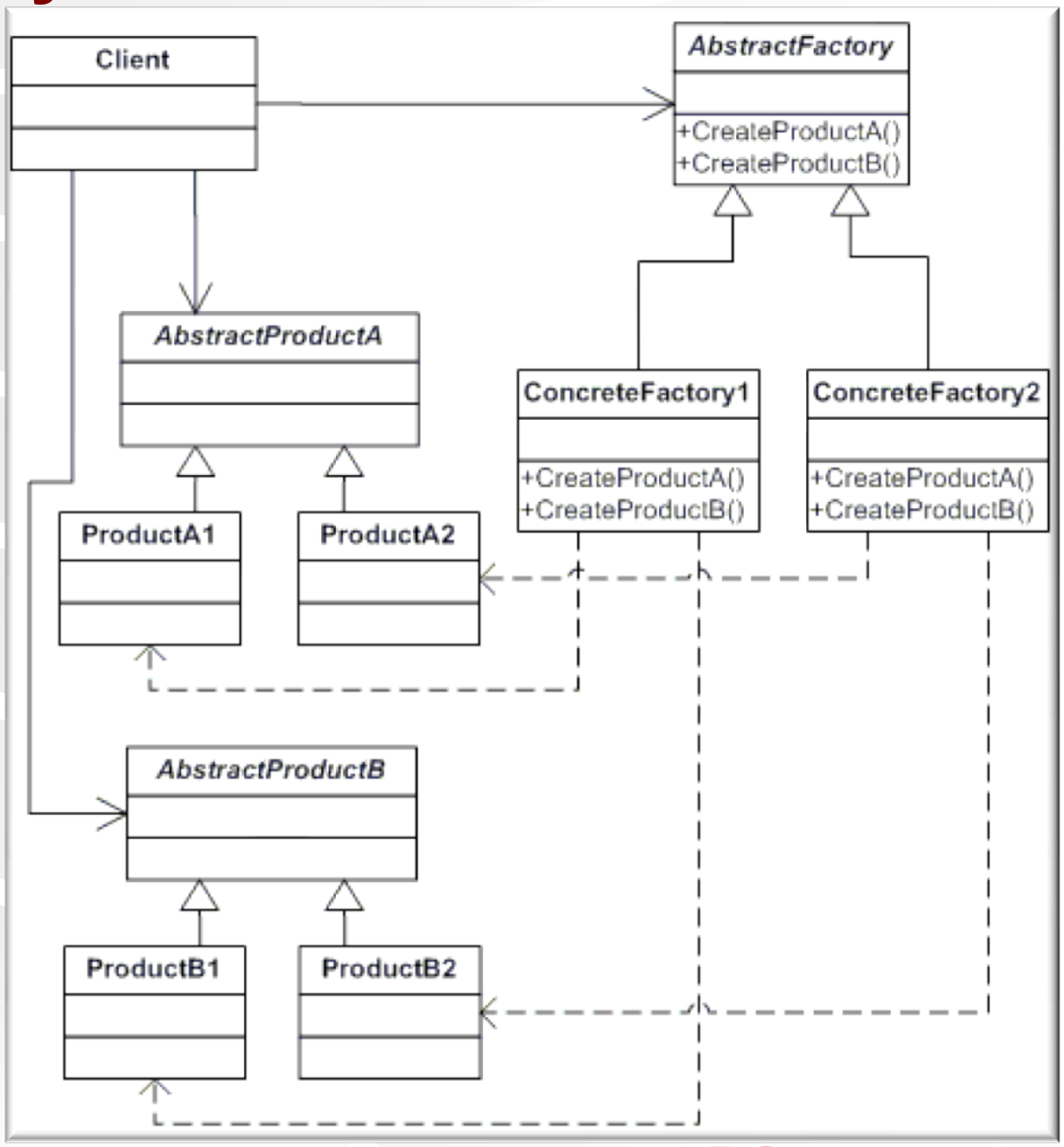
- **Problem**: Who should be responsible for creating objects when there are special considerations like:
  - Complex creation logic
  - Separating creation to improve cohesion
  - A need for caching

- **Solution**: Create a Pure Fabrication called a Factory to handle the creation

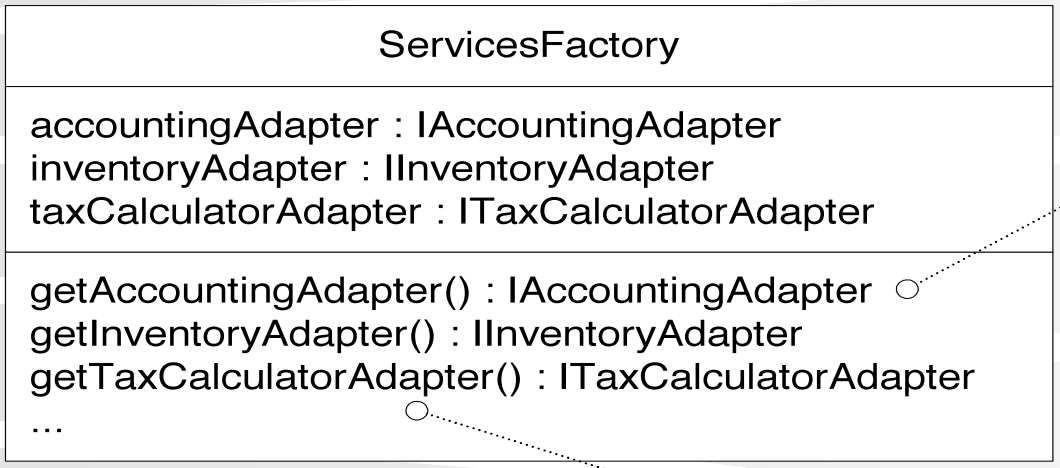
Also known as Simple Factory or Concrete Factory

Abstract Factory: Creational Pattern

- Provides an interface to create and return one of several families of related objects without needing to specify their concrete classes.



Factory Example



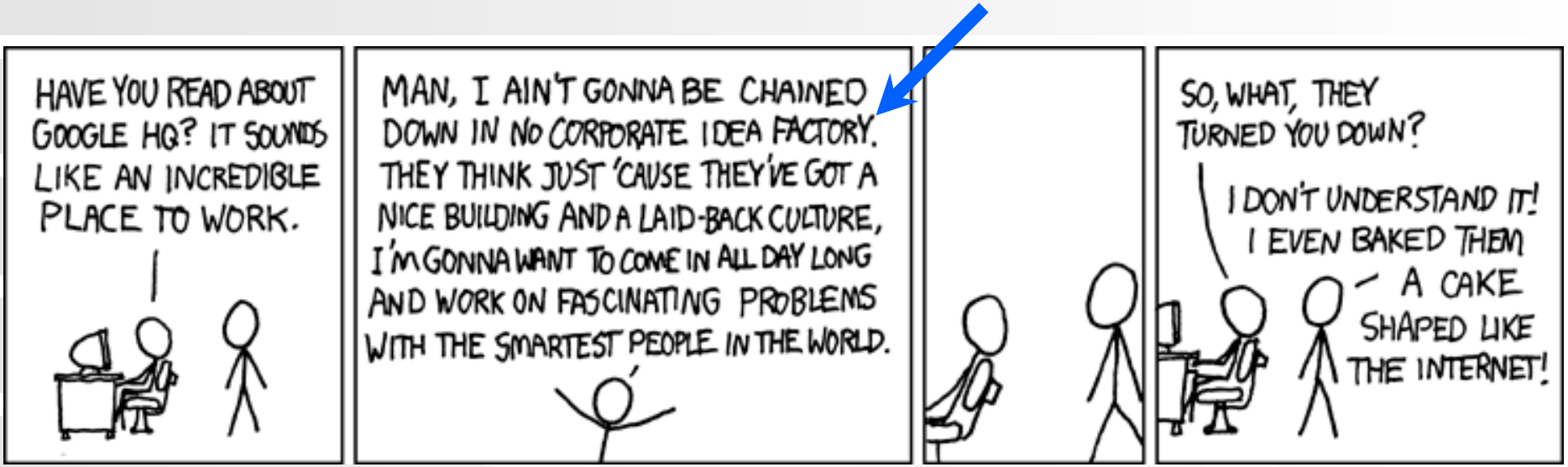
note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```
if ( taxCalculatorAdapter == null )
{
  // a refl ective or datadriven approach to fi nding the right class read it from an
  // external property

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

}
return taxCalculatorAdapter;
```

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

Working for Google



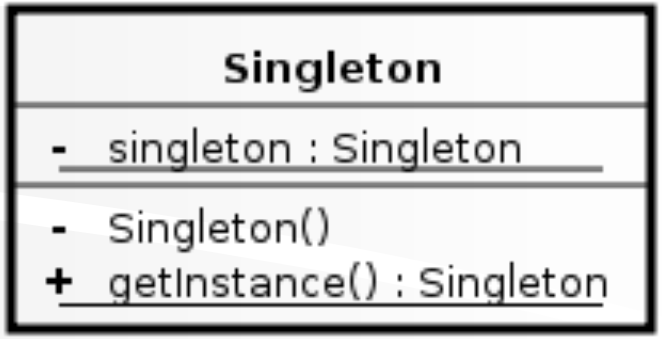
I hear once you've worked there for 256 days they teach you the secret of levitation.

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

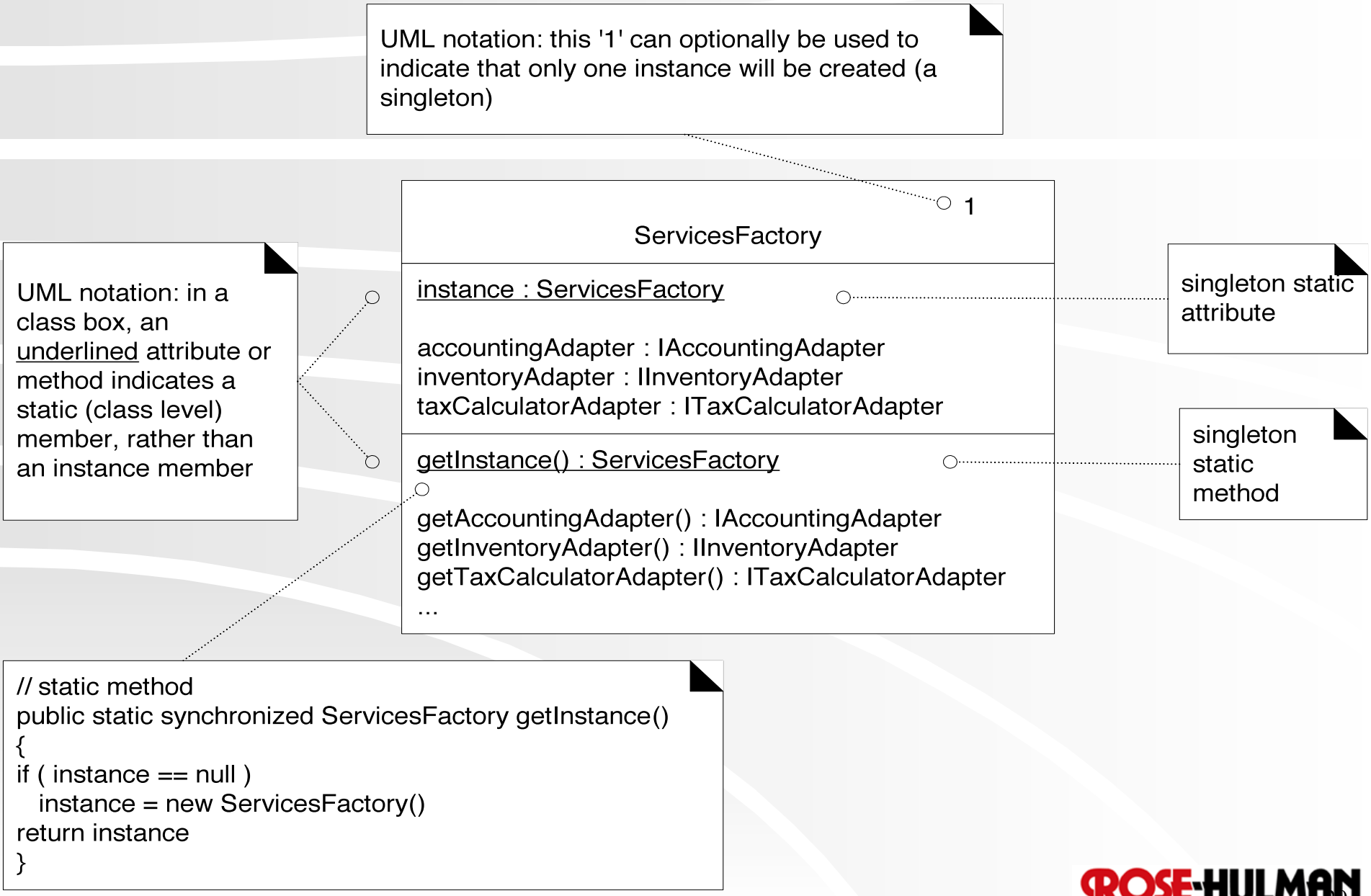
Singleton

- **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
  - Created only once for the life of the program (a non-creational pattern?)
  - Provides single global point of access to instance
    - Similar to a static or global variable variable

Singleton Example



Lazy vs. Eager Initialization

- Lazy:

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

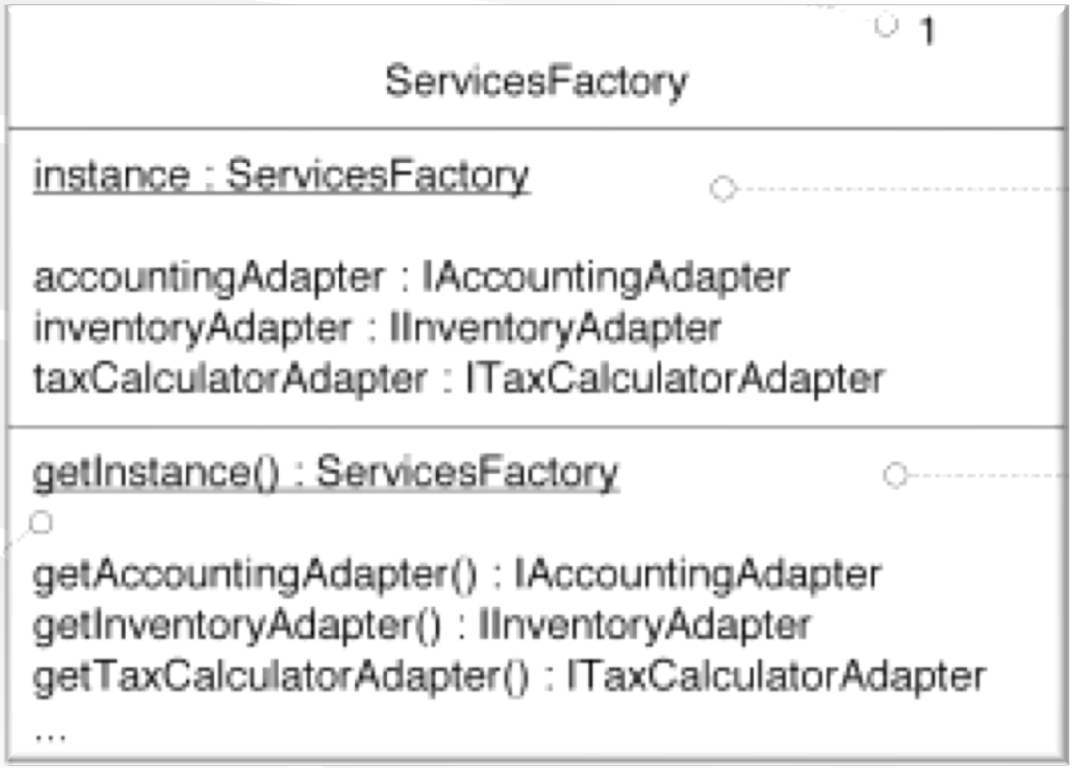
- Eager:

```
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



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

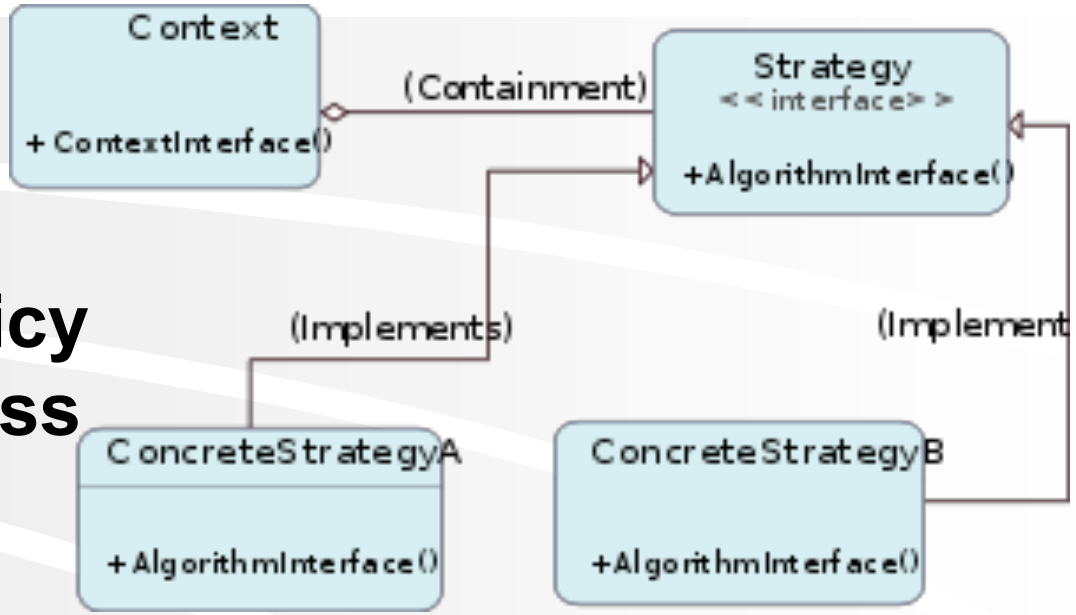
http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx

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

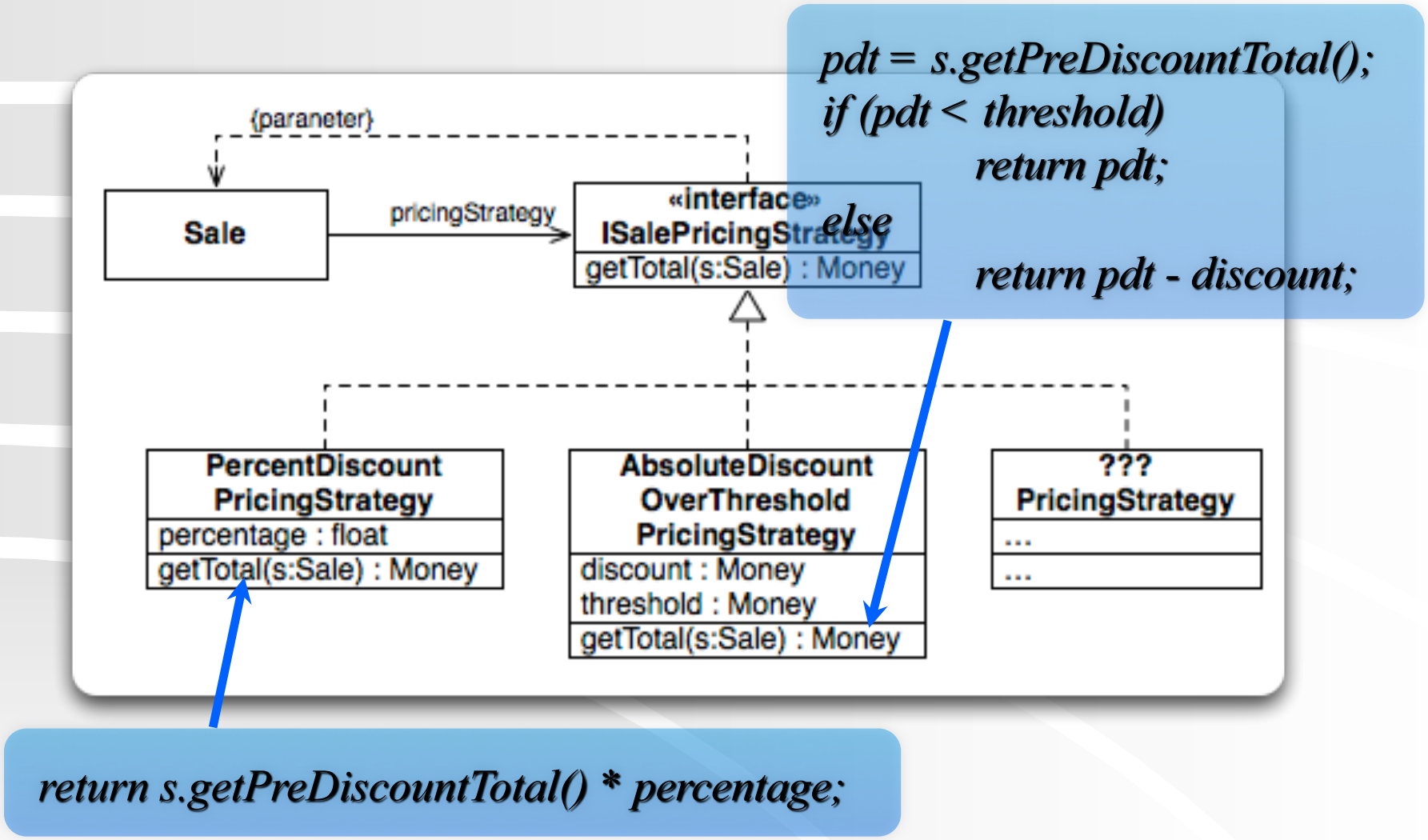
Strategy

- **Problem**: How do we design for varying, but related, algorithms or policies?

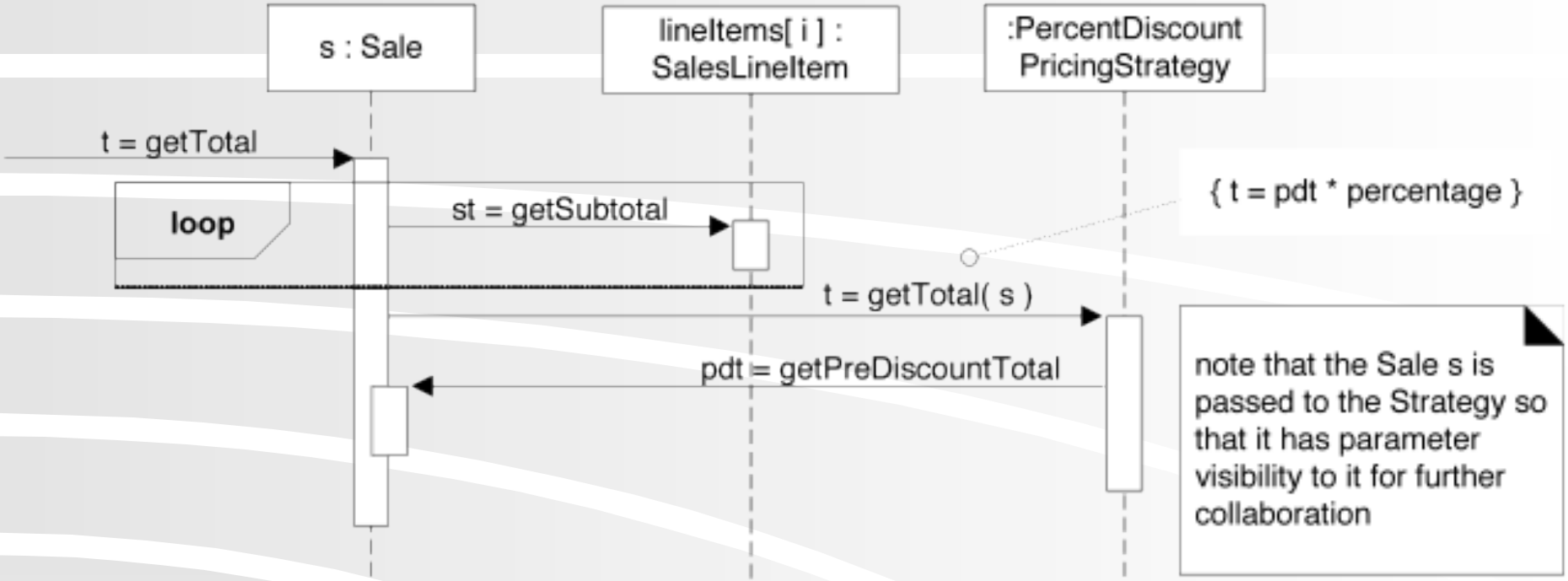
- **Solution**: Define each algorithm or policy in a separate class with a common interface.



Strategy Example

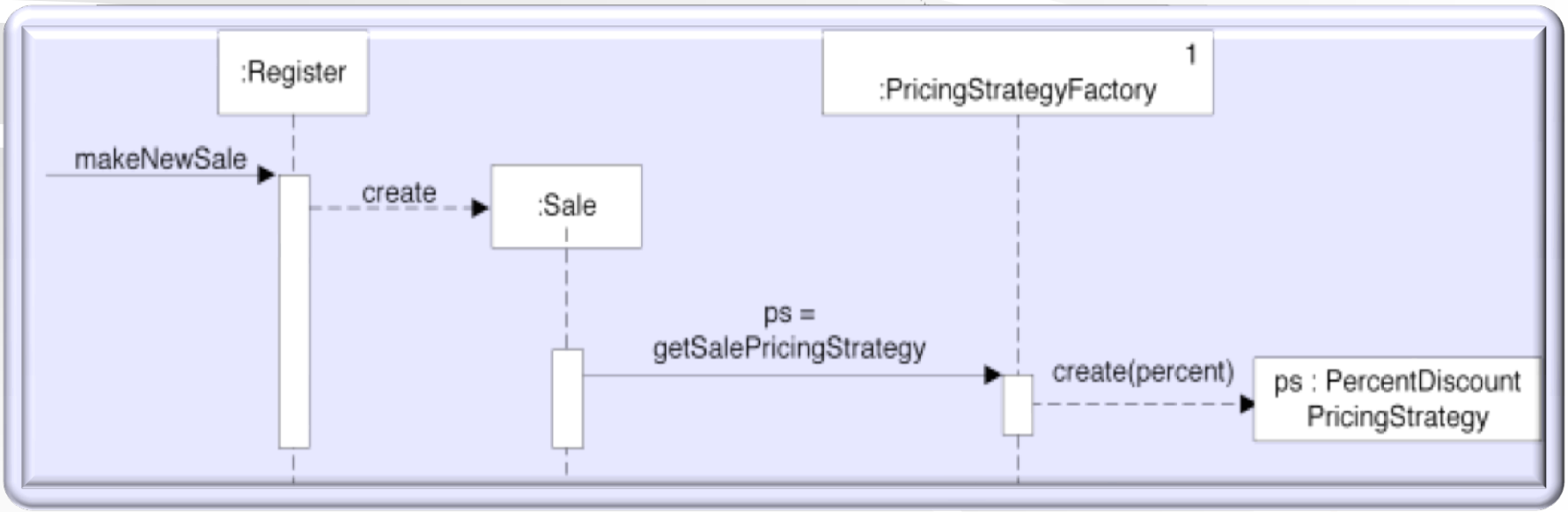


Strategy Example (cont.)



Where does the *PricingStrategy* come from?

What about with Dependency Injection?



Examples of Change and Patterns

What Varies	Design Pattern
Algorithms	Strategy, Visitor
Actions	Command
Implementations	Bridge
Response to change	Observer
Interactions between objects	Mediator
Object being created	Factory Method, Abstract Factory, Prototype
Structure being created	Builder
Traversal Algorithm	Iterator
Object interfaces	Adapter
Object behavior	Decorator, State

Homework and Milestone Reminders

- Homework 6 – More GRASP on Video Store Design
  - Due by 5:00pm Tuesday, January 26th, 2010
- Milestone 4: Patterns and Detailed Design, with some Iteration 2 on the Side
  - Due by 11:59pm Friday, January 29th, 2010