

SWEN30006

Software Modelling and Design

APPLYING GOF DESIGN PATTERNS

Larman Chapter 26

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

—Ward Cunningham and Ralph Johnson

Lectured by:
Peter Eze

A Brief History of Design Patterns

- ❑ **1977:** Christopher Alexander publishes: “A Pattern Language: Towns, Buildings, Construction”
- ❑ **1987:** Cunningham and Beck used Alexander’s ideas to develop a small pattern language for Smalltalk
- ❑ **1990:** The Gang of Four (Gamma, Helm, Johnson and Vlissides) begin work compiling a catalog of design patterns
- ❑ **1991:** Bruce Anderson gives first Patterns Workshop at OOPSLA
- ❑ **1993:** Kent Beck and Grady Booch sponsor the first meeting of what is now known as the [Hillside Group](#)
- ❑ **1994:** First Pattern Languages of Programs (PLoP) conference
- ❑ **1995: The Gang of Four (GoF) publish their ground breaking book: “Design Patterns: Elements of Reusable Software”**

Patterns in Architecture

"The **street cafe** provides a unique setting, special to cities: a place where people can sit lazily, legitimately, be on view, and watch the world go by...

Encourage local cafes to spring up in each neighborhood. Make them intimate places, with several rooms, open to a busy path, where people can sit with coffee or a drink and watch the world go by. Build the front of the cafe so that a set of tables stretch out of the cafe, right into the street."

— Christopher Alexander et al., *A Pattern Language*, pages 437,439



Gang of Four Design Patterns

Creational Patterns (5):

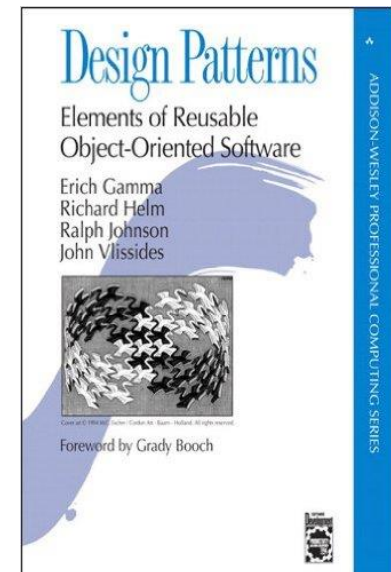
- abstract the object instantiation process.

Structural Patterns (7):

- describe how classes and objects can be combined to form larger structures.

Behavioural Patterns (11):

- are most specifically concerned with communication between objects.



Objectives

On completion of this topic you should be able to:

- ❑ Apply some GoF design patterns
 - Adapter
 - (Concrete) Factory
 - Singleton
 - Strategy
 - Composite
 - Façade
 - Observer
 - Decorator
- ❑ Recognise GRASP principles as a generalization of other design patterns.

Problem 1: External Services with Varying Interfaces

- ❑ POS system needs to support several kinds of external services, including:
 - Tax calculators
 - e.g., TaxMaster, GoodAsGoldTaxPro
 - Accounting systems
 - e.g., SAP, GreatNorthern
 - Credit authorization services
- ❑ Each service has a different API, which cannot be changed.

Adapter (GoF)

Problem:

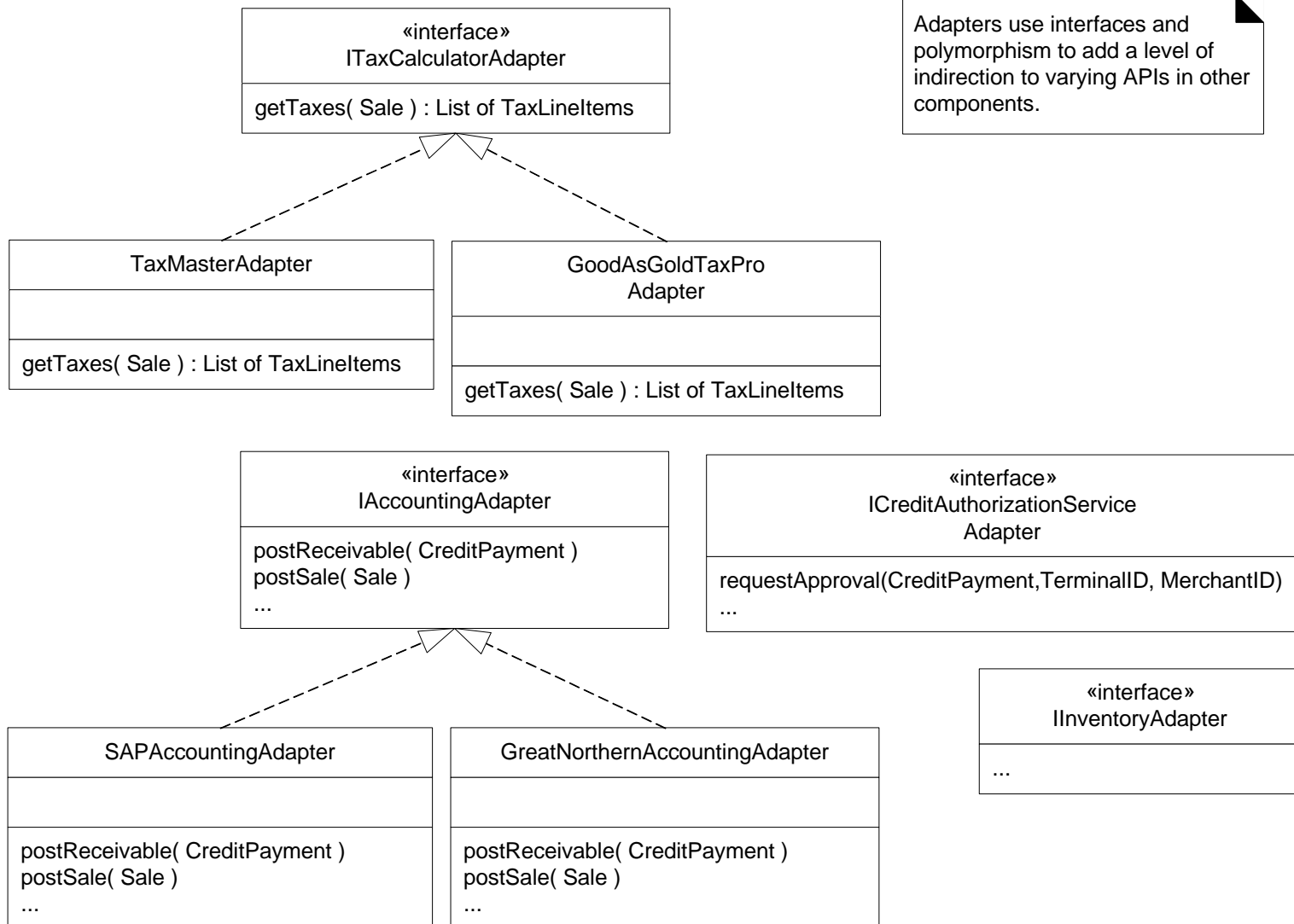
- ❑ How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

Solution (advice):

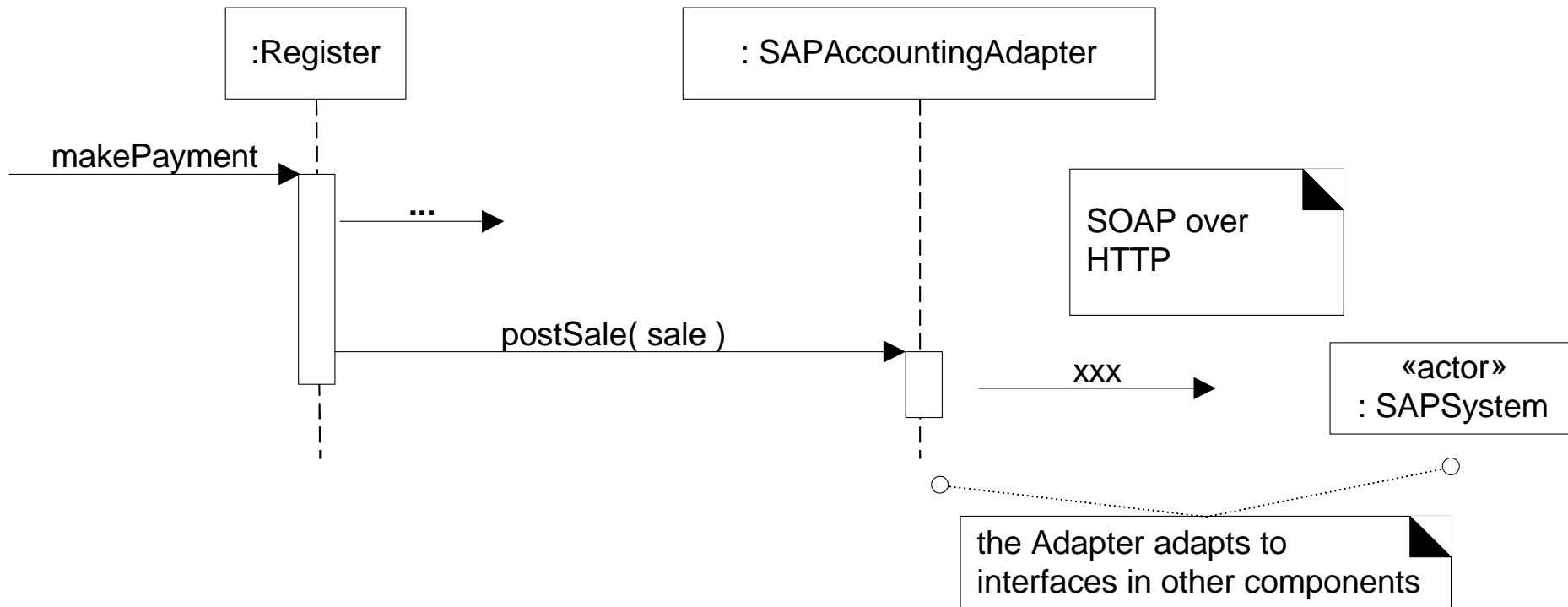
- ❑ Convert the original interface of a component into another interface, through an intermediate adapter object.



The Adapter Pattern



Using an Adapter



Example: Disney Madness

- ❑ Using an external libraries for Disney characters
 - APIs cannot be modified

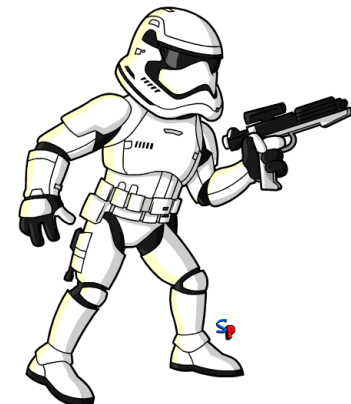
DisneyCharacters::Sloth
+ wake(): void + pushToMove() : void



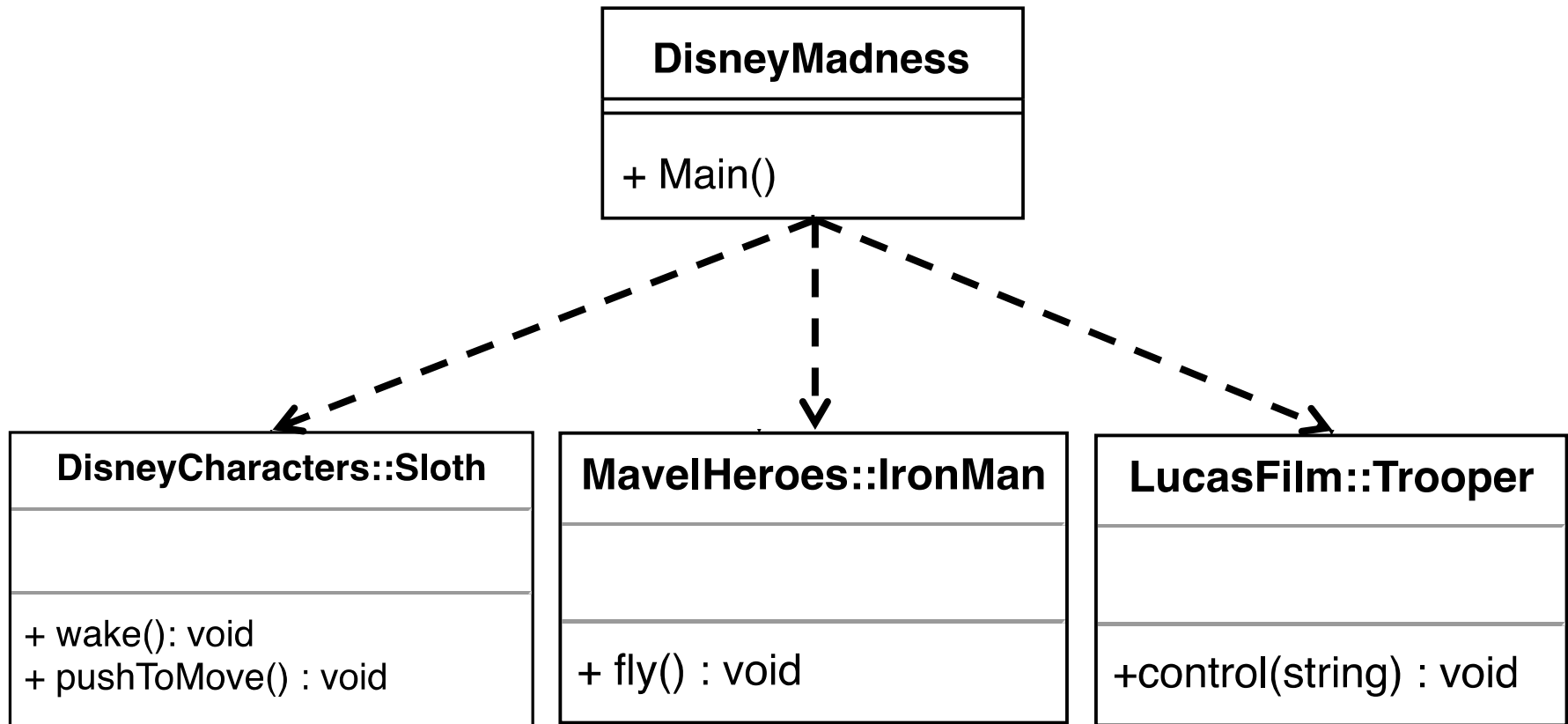
MavelHeroes::IronMan
+ fly() : void



LucasFilm::Trooper
+control(string) : void



Disney Madness: Design *without* Adapter



Disney Madness: Design *with* Adapter

DisneyMadness

+ Main()

Create a common interface for the Main class and
create an adapter for each library

DisneyCharacters::Sloth

+ wake(): void
+ pushToMove() : void

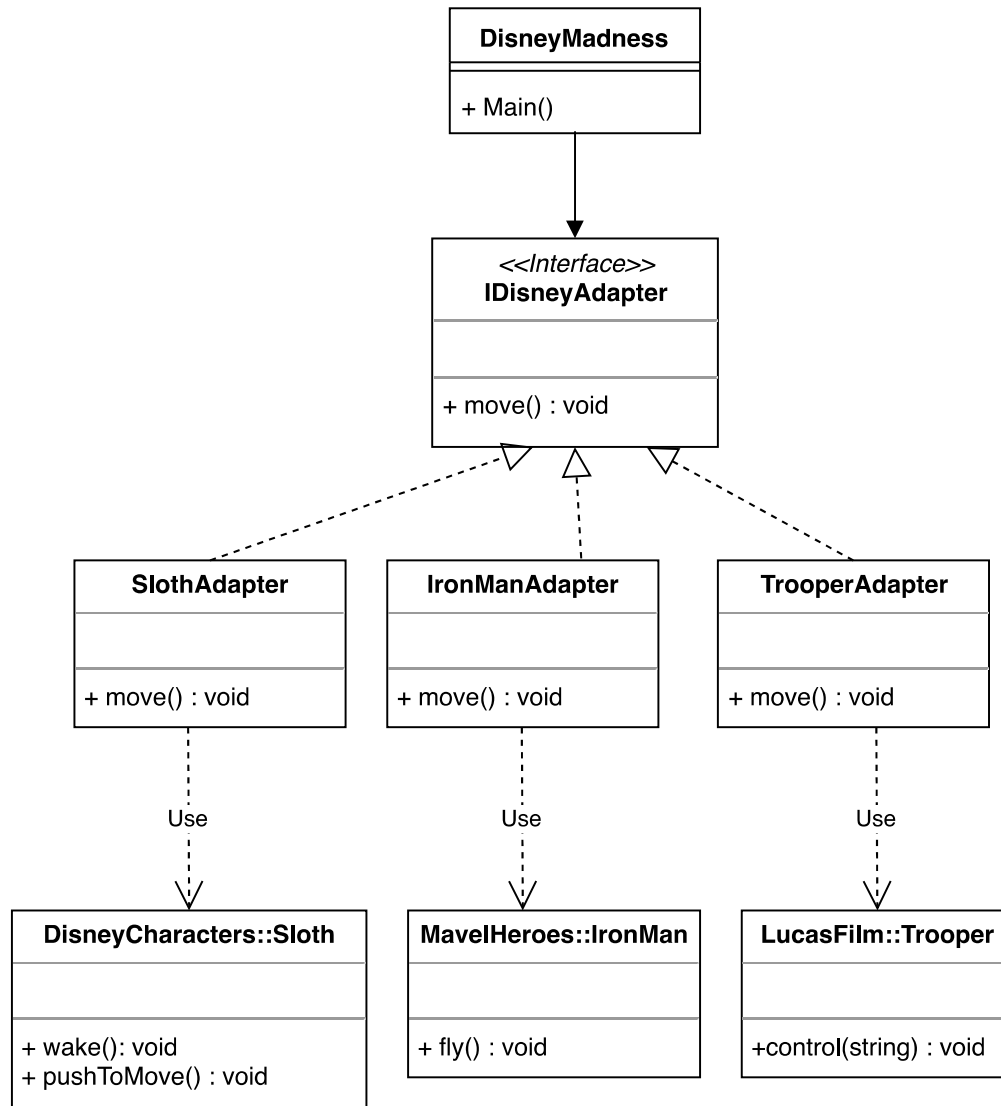
MavelHeroes::IronMan

+ fly() : void

LucasFilm::Trooper

+control(string) : void

Disney Madness: Design *with* Adapter



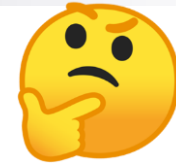
Disney Madness: Implement *with* Adapter

```
public interface IDisneyAdapter {  
    public void move();  
}
```

```
public class SlothAdapter implements IDisneyAdapter {  
    private Sloth theSloth;  
    public SlothAdapter(String name) {  
        theSloth = new Sloth(name);  
    }  
    public void move() {  
        theSloth.wake();  
        theSloth.pushToClimb();  
    }  
}
```

Disney Madness: Implement *with* Adapter

```
public class DisneyMadess {  
    public static void main(String[] args) {  
        //With adapter  
        IDisneyAdapter slothAdt = new SlothAdapter("Flash");  
        slothAdt.move();  
  
        IDisneyAdapter ironManAdt = new IronManAdapter("Tony");  
        ironManAdt.move();  
  
        IDisneyAdapter trooperAdt = new TrooperAdapter("Storm");  
        trooperAdt.move();  
    }  
}
```



Thinking Time

Which of the following GRASP principle(s) are used by the Adapter pattern?

Polymorphism

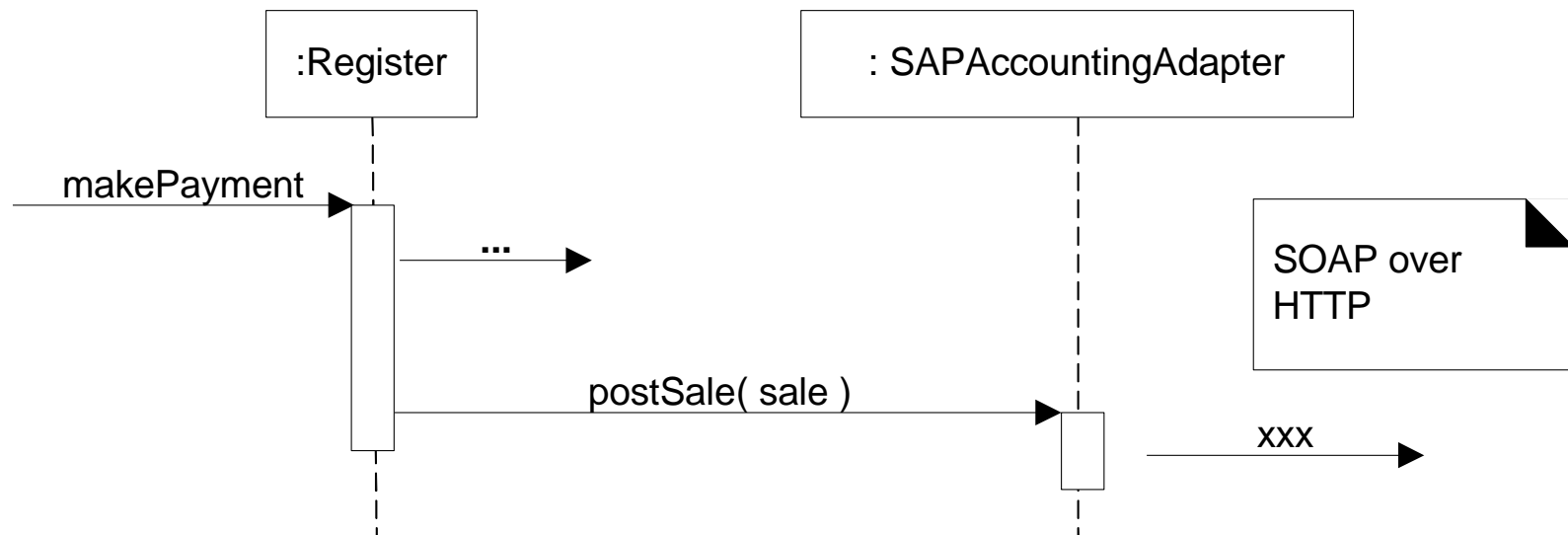
Creator Information
Expert

Indirection

Pure Fabrication

Issues arising from Adapter

- ❑ Who creates the adapters?
- ❑ Which class of adapter should be created?
 - Should the Register object create SAPAccountingAdapter?



Issues arising from Adapter

- ❑ Who creates the adapters?
- ❑ Which class of adapter should be created?
 - Should the Register object create SAPAccountingAdapter?

Partial Answer:

- ❑ Separation of Concerns/High Cohesion
 - Creating adapters is not pure application logic
 - Adapter object is not a domain object
- ❑ **Use GRASP Pure Fabrication pattern**

Factory

Problem:

- ❑ Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?

Solution (advice):

- ❑ Create a Pure Fabrication object called a Factory that handles the creation.

(Concrete) Factory

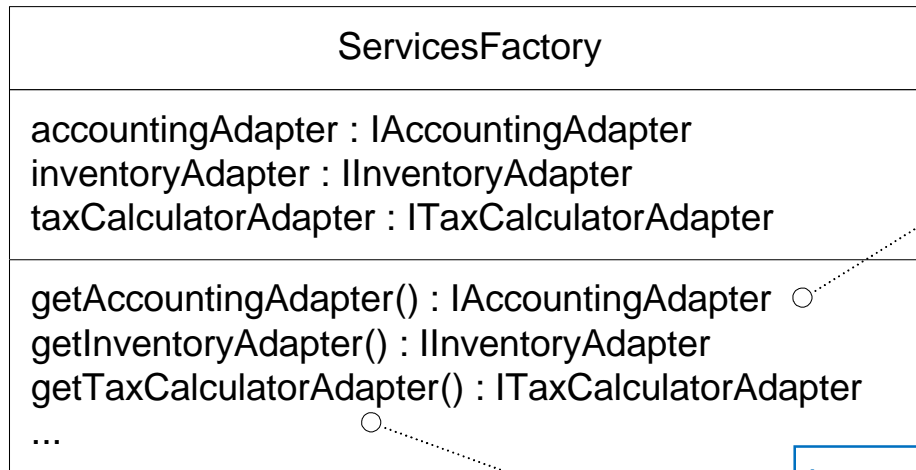
Aka **Simple Factory**

- ❑ Simplified GoF Abstract Factory pattern

Advantages:

- ❑ Separate responsibility of complex creation into cohesive helper objects.
- ❑ Hide potentially complex creation logic.
- ❑ Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

The Factory Pattern



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

Properties File Entry

```
! taxcalculator.class.name=TaxMasterAdaptor  
taxcalculator.class.name=GoodAsGoldTaxProAdaptor
```

```
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;
```

Example: Disney Madness

- ❑ Using an external libraries for Disney characters
 - APIs cannot be modified

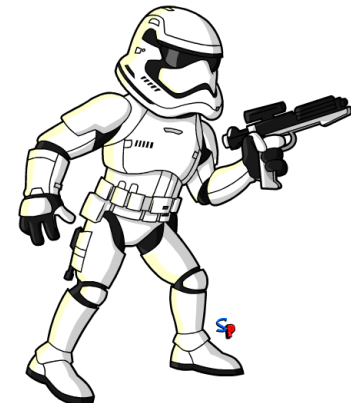
DisneyCharacters::Sloth
+ wake(): void + pushToMove() : void



MavelHeroes::IronMan
+ fly() : void



LucasFilm::Trooper
+control(string) : void

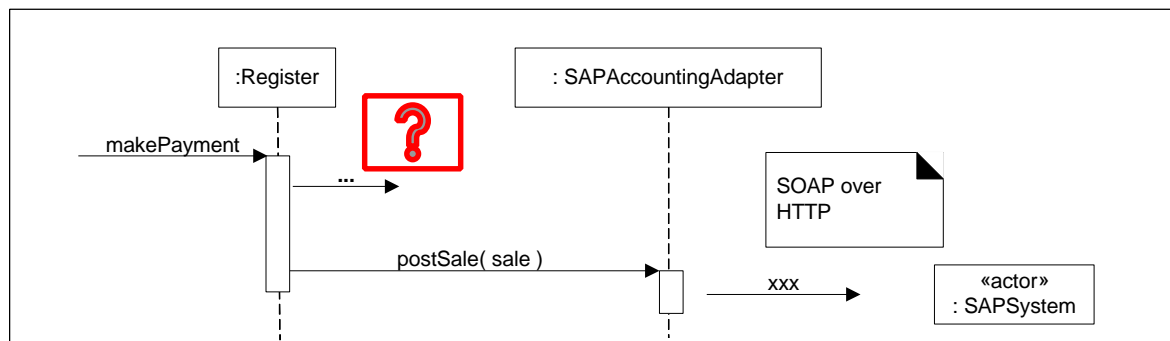


Implement DisneyMadness *with* Adapter and Factory

```
public class DisneyFactory {  
    private IDisneyAdapter characterAdapter = null;  
  
    public IDisneyAdapter getDisneyAdapter(String character, String name){  
        if(character.equals("ironman")) {  
            characterAdapter = (IDisneyAdapter) new IronManAdapter(name);  
        }else if(character.equals("sloth")) {  
            characterAdapter = (IDisneyAdapter) new SlothAdapter(name);  
        }else if(character.equals("trooper")) {  
            characterAdapter = (IDisneyAdapter) new TrooperAdapter(name);  
        }  
        return characterAdapter;  
    }  
}
```

Issues arising from Factory

- ❑ Who creates the Factory and how is it accessed?
 - It may need to be called from various places
- ❑ Oftentimes only one instance of this factory is needed.
- ❑ How to get visibility to this single **ServicesFactory** instance?



Issues arising from Factory

- ❑ How is Factory accessed?
 - It may need to be called from various places
- ❑ How to get visibility to this single ServicesFactory instance?

Partial Answer:

- ❑ Only one instance of the factory is needed
- ❑ Where required, pass through to methods or initialise objects with a ref? *No.*
- ❑ Use **Singleton pattern** to provide a single access point through global visibility

Singleton (GoF)

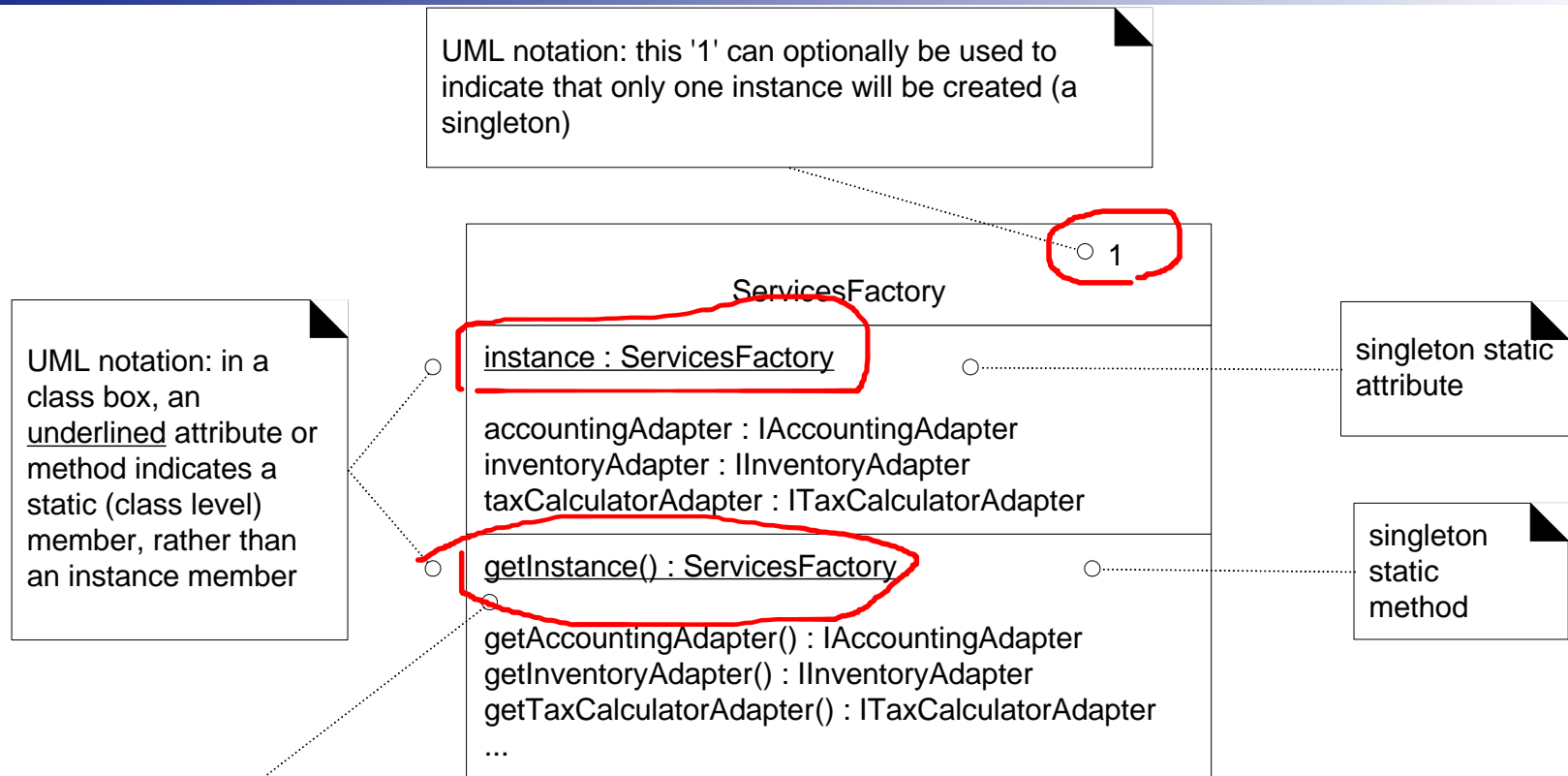
Problem:

- ❑ Exactly one instance of a class is allowed—it is a “singleton.” Objects need a global and single point of access.

Solution (advice):

- ❑ Define a static method of the class that returns the singleton.

Singleton Pattern in *ServicesFactory* Class



```

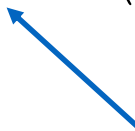
// static method
public static synchronized ServicesFactory getInstance()
{
    if ( instance == null )
        instance = new ServicesFactory()
    return instance
}

```

Lazy Initialization

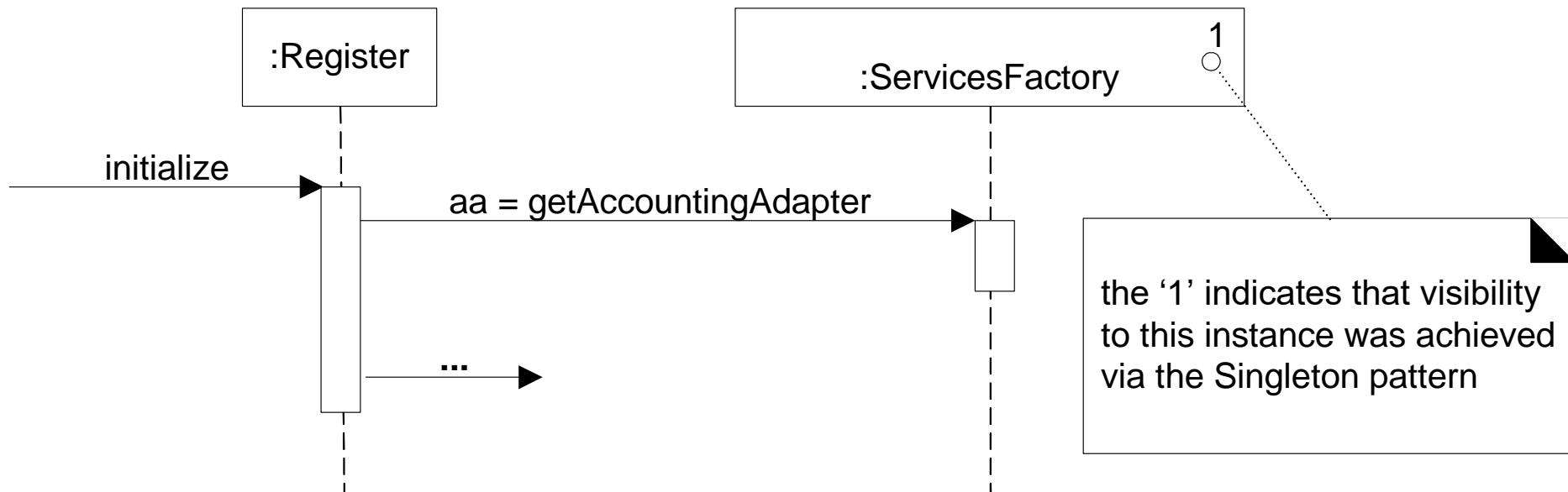
Accessing the Singleton Factory

```
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
```



SingletonClass.getInstance()
is globally visible

Implicit *getInstance* Singleton Pattern Message



```
aa = ServicesFactory.getInstance().getAccountingAdapter();
```

Static: why some and not all?

Why aren't all Singleton service methods *static*?

E.g. static `getAccountingAdapter()`: ...

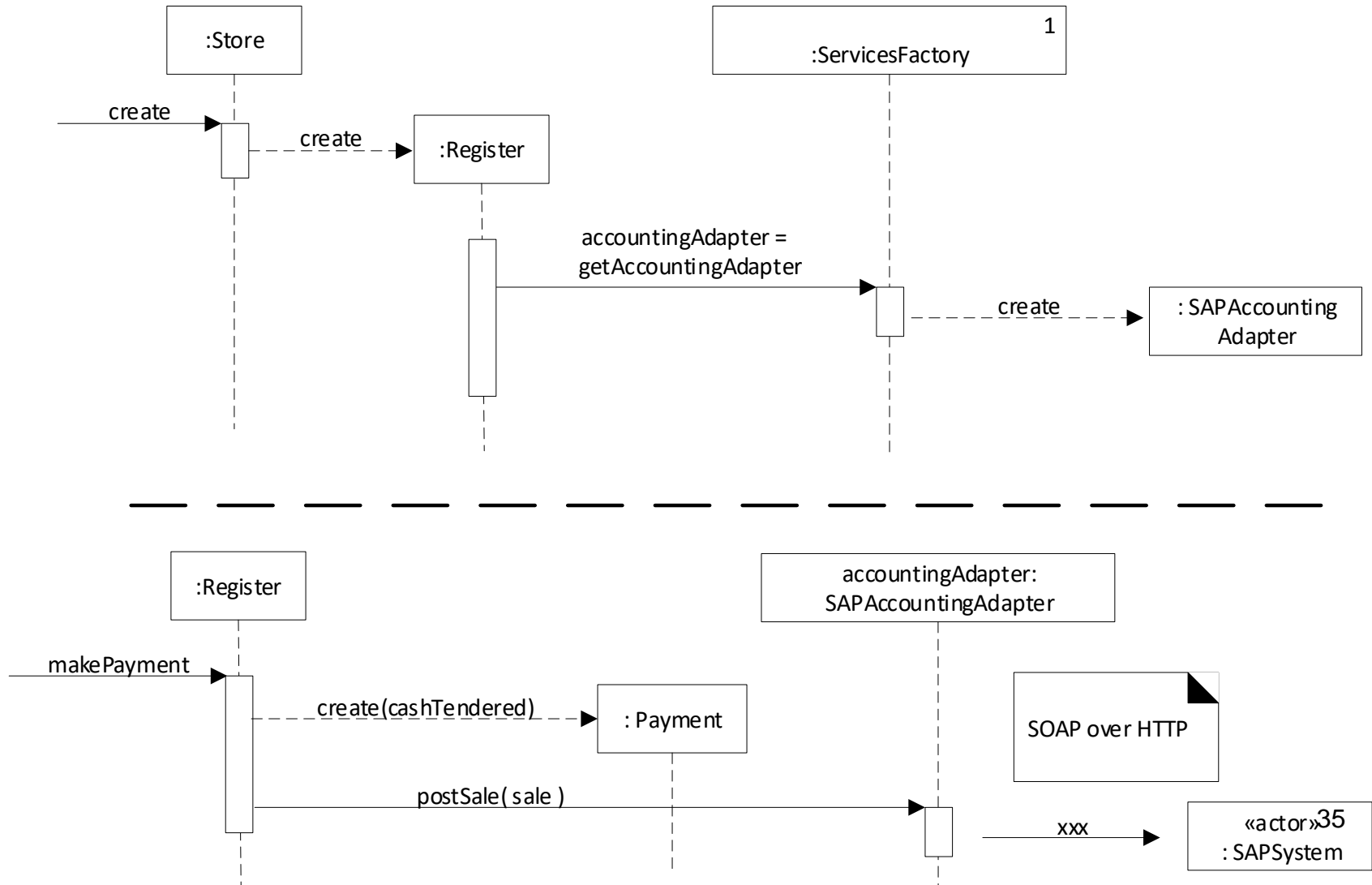
- ❑ May want subclasses: static methods are not polymorphic (virtual); no overriding in most languages
- ❑ Most remote communication mechanisms (e.g. Java's RMI) don't support remote-enabling of static methods.
- ❑ A class is not always a singleton in all application contexts.

Problem 1 Revisited: External Services with Varying Interfaces

Possible solution:

- ❑ “To handle this problem, let’s use Adapters generated from a Singleton Factory.”
- ❑ Design reasoning based on:
 - Controller
 - Creator
 - Protected Variations
 - Low Coupling
 - High Cohesion
 - Indirection
 - Polymorphism
 - Adaptor
 - Factory
 - Singleton

Adapter, Factory, and Singleton Patterns



More examples, from YouTube

- ❑ [Adaptor Design Pattern](#)
- ❑ [Factory Design Pattern \(1\)](#)
- ❑ [Factory Design Pattern \(2\)](#)
- ❑ [Singleton Design Pattern](#)

Lecture Identification

Coordinator: Patanamon Thongtanunam

Lecturer: Peter Eze

Semester: S2 2020

© University of Melbourne 2020

These slides include materials from:

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, by Craig Larman, Pearson Education Inc., 2005.