Object Oriented Design and Modeling using UML

Design of an Object Oriented System

Introduction

- Object Oriented Design (OOD)
  - An approach used to specify the software solution in terms of collaborating objects, their attributes, and their methods.

Different types of Object classes

- Entity classes
  - An object oriented class that contains business related information and implements the analysis classes.
  - Correspond to items in real life (e. g. MEMBER / ORDER) and contain information known as attributes that describes the different instance of the entity.

Different types of Object classes

- Interface classes (Boundary Classes)
  - An object class that provides the means by which an actor can interface with the system.
  - e.g. a window, dialog box, screen
  - Also known as boundary class

Different types of Object classes

- Interface classes (Boundary Classes)
  - The responsibility of an interface class is twofold,
    - Translates the user's input into information that the system can understand and use to process the business event
    - Takes data pertaining to a business event and translates the data for appropriate presentation to the user.

Different types of Object classes

- Control classes
  - An object class that contains application logic
  - Implement the business logic or business rules of the system
  - Process messages from an interface class and respond to them by sending and receiving messages from the entity class.

Different types of Object classes

- Persistence Classes
  - –An object class that provides functionality to read and write attributes in a database. The functionality could be built into the entity classes.
  - –But if that functionality is put into a separate persistent (data access) classes, the entity classes are kept implementation independent.

Different types of Object classes

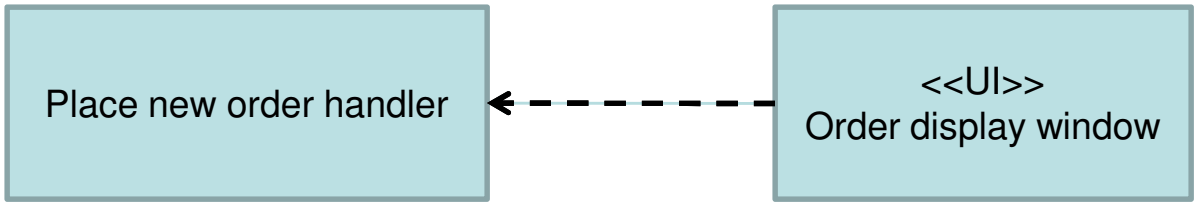
- System Classes
  - An object class that handles operating system specific functionality
  - If the system is ported to another operating system only the system classes and perhaps the interface classes have to be changed.

Design Relationships

- In object-oriented analysis, identify the most common object relationships.
- In object-oriented design, model more advanced relationships in order to accurately specify the software components.

Design Relationships

- Dependency Relationships
  - Used to model the association between two classes in two instances:
    - To indicate that when a change occurs in one class, it may affect the other classes
    - To indicate the association between a persistence class and an interface class
- e. g.



UCSC

Design Relationships

- Navigability
  - Illustrated with an arrow head pointing only to the direction a message can be sent.
  - By default associations between classes are bi-directional, meaning that classes of one kind can navigate to classes of the other class
  - However, there may be times when you want to limit the message sending to only one direction.

Attribute and Method visibility

- Visibility:
  - The level of access an external object has to an attribute or method
  - UML provides three levels of visibility
    - Public (+)
    - Protected (#)
    - Private (-)

UCSC

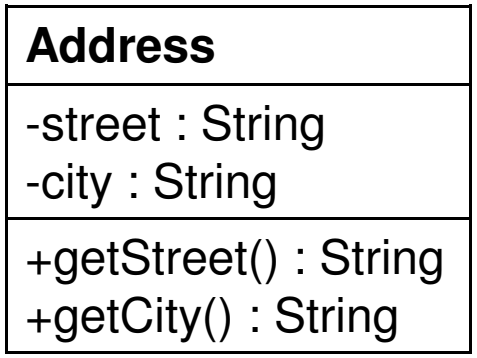
Attribute and Method visibility

- Public attributes can be accessed and public methods can be invoked by any other method in any other class.
- Protected attributes can be accessed and protected methods can be invoked by any method in the class in which the attribute or method is defined or in subclasses of that class.
- Private attributes can be accessed and private methods can be invoked only by any method in the class in which the attribute or method is defined.

Attribute and Method visibility

- If a method needs to be invoked in response to a message sent by another class, the method should be declared public.
- In most cases all attributes should be declared private to enforce encapsulation.

e.g:



UCSC

Object responsibilities

- The obligation that an object has to provide a service when requested
- Thus, collaborate with other objects to satisfy the request if required.
- Closely related to the concept of being able to sent and/or respond to messages.

The process of Object-Oriented design

- During OOA
  - Define use cases and identify objects based on ideal conditions
- During OOD
  - Refine the use cases and objects to reflect the actual environment of the proposed solution.

The process of Object-Oriented design

- OOD includes the following activities
  - Refining the use-case model to reflect the implementation environment
  - Modeling class interactions, behaviors, and states that support the use-case scenario
  - Updating the class diagram to reflect the implementation environment

Refining the use-case model

- Use-cases will be refined to include details of
  - how the actor will actually interface with the system
  - how the system will respond to that reaction to process the business event

Refining the use-case model

- The manner in which the user access the system should be described in detail.
  - e.g. Via a menu, window, button, barcode reader, printer
- The contents of windows, queries, and reports should be specified within the use case

Refining the use-case model

- Steps to adapt each use case to the implementation environment
  - Step1: Transforming the “Analysis” use cases to “Design” use cases
  - Step2: Updating the use-case model diagram and other documentation to reflect any new use cases

For more information refer ref1 pg 651-655

UCSC

Modeling class interactions, behaviors, and states that support the use-case scenario

- Step1: Identify and classify use-case design classes
- Step2: Identify class attributes
- Step3: Identify class behaviors and responsibilities

For more details refer ref1 pg 656-665

Updating the object model to reflect the implementation environment

- Once the objects and other interactions have been design, class diagram can be refined to represent software classes in the application
- Design class diagram : a diagram that depicts classes that correspond to software components that are used to build the software application.

Updating the object model to reflect the implementation environment

- A design class diagram includes the following
  - Classes
  - Associations , gen/spec and aggregation relationships
  - Attributes and attribute type information
  - Methods and parameters
  - Navigability
  - Dependencies

For more details refer ref1 pg 665-666

UCSC

Object reusability and Design patterns

- Two very important goals of OOD
  - Low coupling
  - High Cohesion
- Coupling : the degree to which one class is connected to or relies upon other classes
- Cohesion : the degree to which the attributes and behaviors of a single class are related to each other

Object reusability and Design patterns

- Reasons behind high cohesion and low coupling
  - Each class focuses on one thing
  - Classes are independent
  - object reusability: object classes created for one information system should be able to reused in other information systems.

Design Patterns

- Object oriented developers look for the same reuse opportunities through the use of design patterns
- Design patterns: a common solution to a given problem in a given context, which supports reuse of proven approaches and techniques.

Object Reusability and Design Patterns

Design Patterns

Definition:

- A *pattern* is a recurring solution to a standard problem, in a context.(Alexander et al., 1997)
- A **pattern describes a problem which occurs over and over again** in our environment, and then
- Describes the core of the solution to that problem, in such a way that
  - you can use this solution a million times over,
  - without ever doing it the same way twice."

UCSC

Patterns in engineering

- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch using the laws of physics

    Instead, they reuse standard designs with successful track records, learning from experience
  - *Why should software engineers make use of patterns?*
- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture and design

UCSC

The "gang of four" (GoF)

- Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (Addison-Wesley, 1995)
  - *Design Patterns* book **catalogs 23 different patterns** as solutions to different classes of problems, in C++ & Smalltalk
  - The problems and solutions are broadly applicable, used by many people over many years
  - Why is it useful to learn about this pattern?
    - Patterns suggest opportunities for reuse in analysis, design and programming

UCSC

GoF says, In general, a pattern has four essential elements:

1. **Pattern name**
2. **Problem:** describes when to apply the pattern
3. **Solution** :describes the elements that make up the design, their relationships, responsibilities, and collaborations
4. **Consequences** : are the results and trade-offs of applying the pattern.

UCSC

D1

Object Reusability and Design Patterns

Ref: SAD Methods, Whitten, Bentley 7th Ed, 2007

- Consider **Place New Order** sequence diagram in an Order Processing System

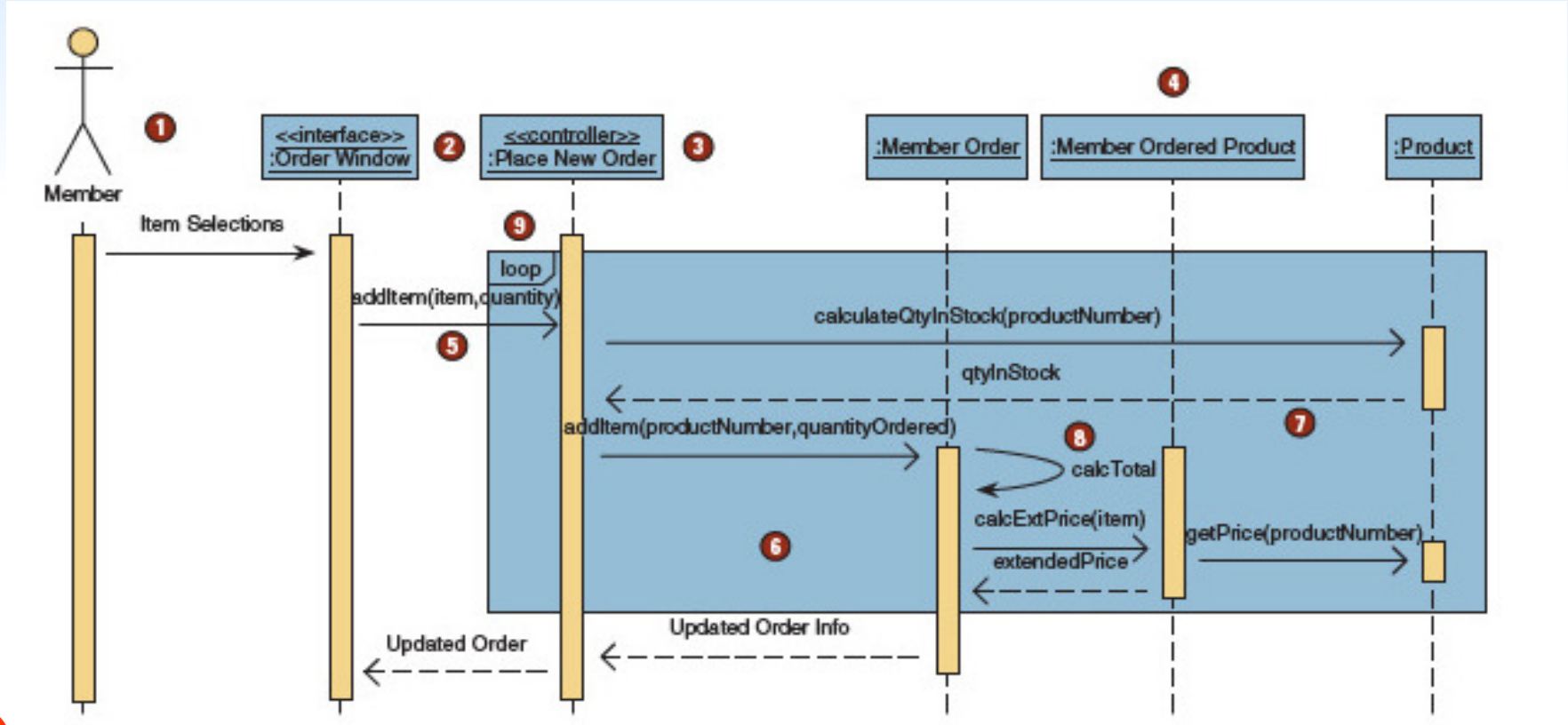


Figure 1

Slide 31

D1 DELL, 4/9/2014

Object Reusability and Design Patterns

- Alternative way - **Place New Order** sequence diagram

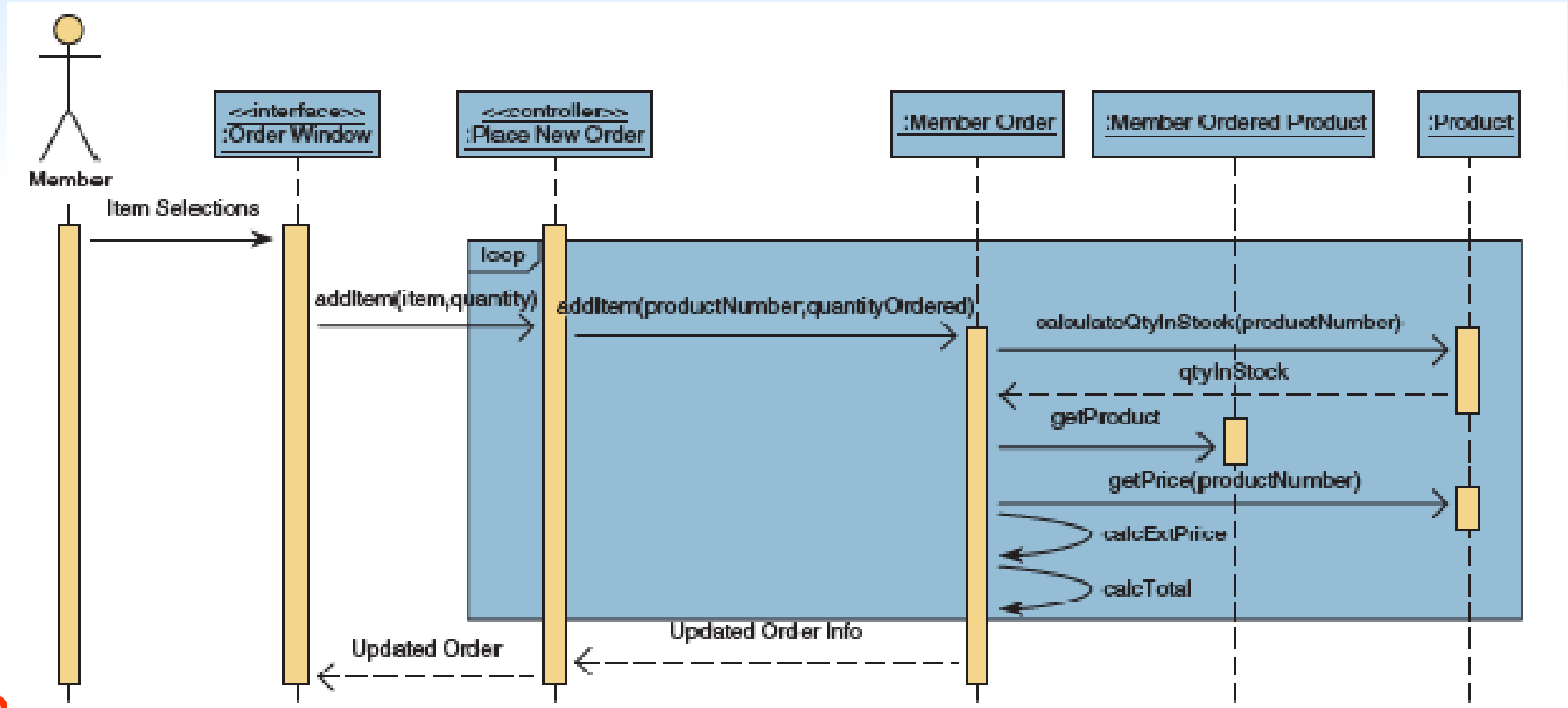
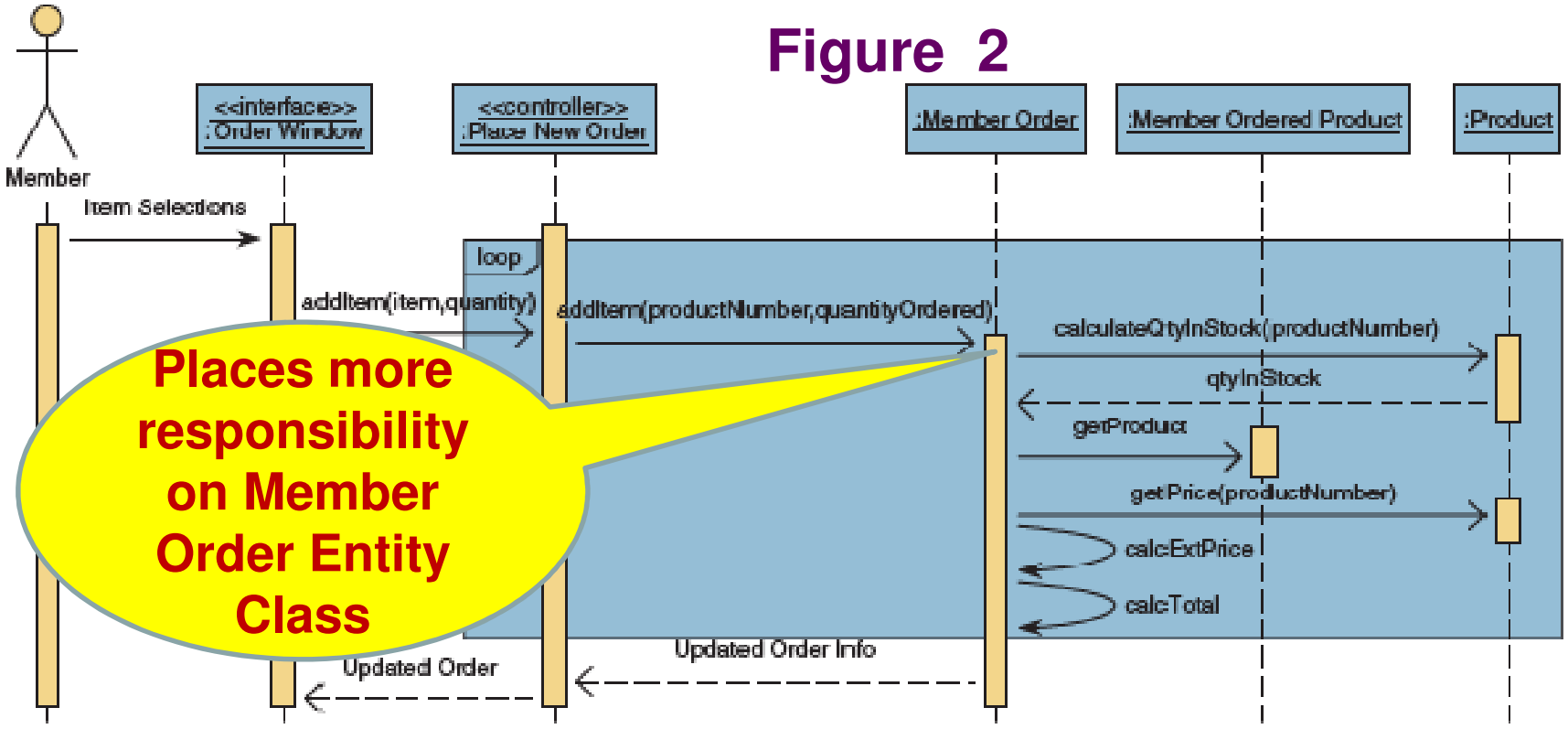


Figure 2

Object Reusability and Design Patterns

Figure 2

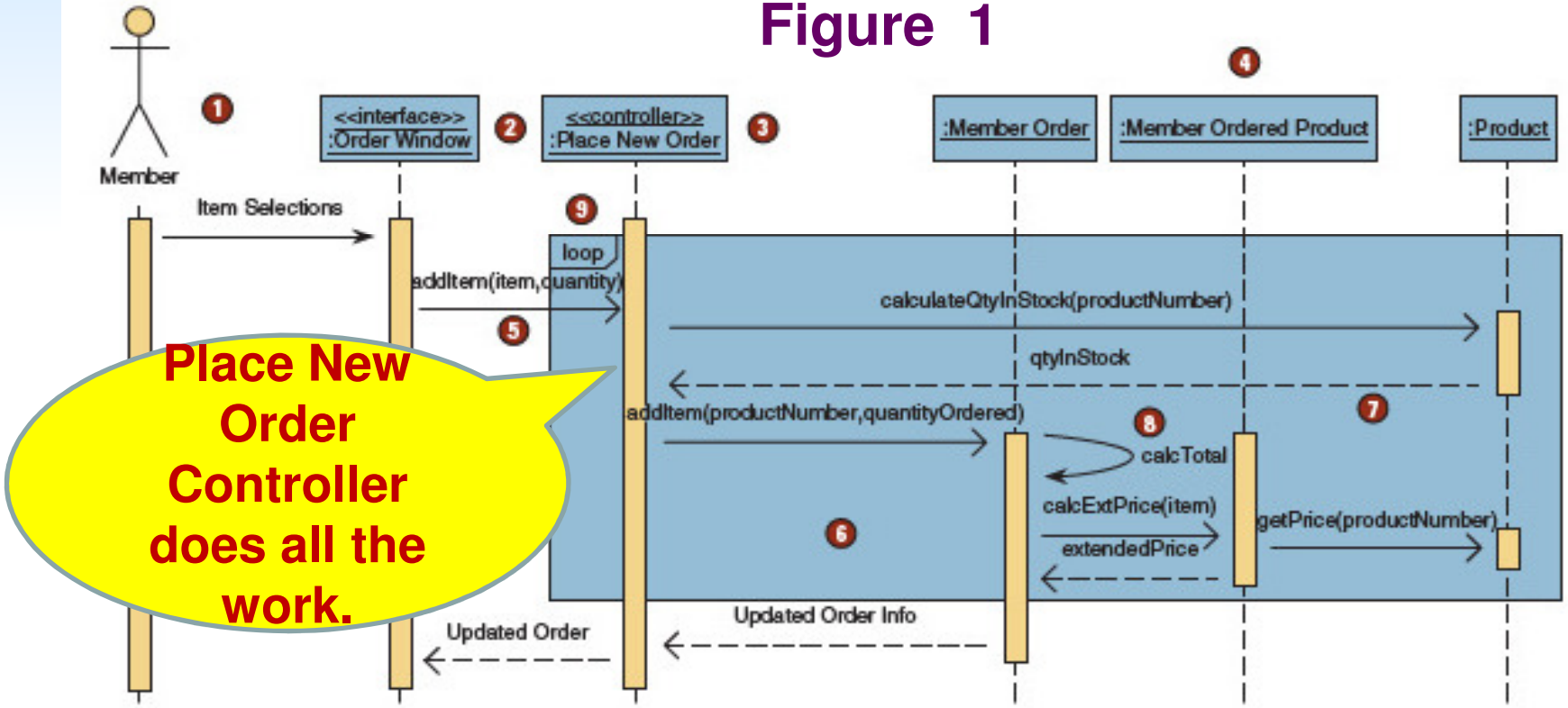


D1

Object Reusability and Design Patterns

- Consider **Place New Order** sequence diagram in an Order Processing System

Figure 1



Slide 34

D1 DELL, 4/9/2014

Coupling and Cohesion

- Two overall goals of OOD are
  - LOW Coupling
  - HIGH Cohesion
- **Coupling** : The degree to which one class is connected to or relies upon other classes.
- **Cohesion** : The degree to which all of the attributes and behavious of a single class are related to each other.

Coupling and Cohesion cont..

- The reason behind the goals of low coupling and high cohesion is **Object Reusability**.

Object Reuse

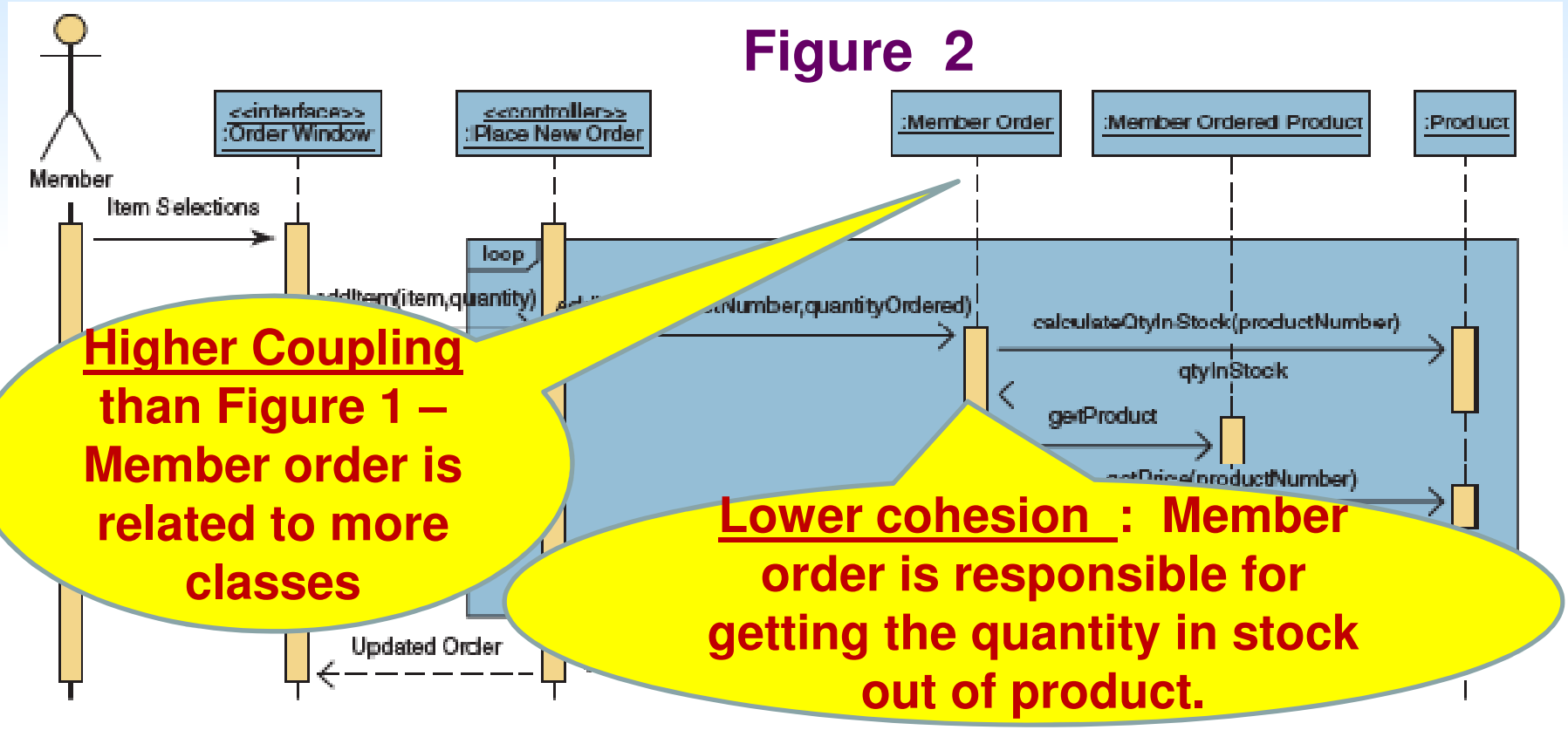
- Several studies have documented the success of object Reuse. [1]

[1] – " White paper on Object Technology : A Key software technology for 90s "- Computer World , May 1992

Electronic Data Systems initiated two projects to develop the same system using two different languages.

Programming Language	Project Duration (calendar months)	Level of Effort (person-months)	Software Size (lines of code)
PL/1	19	152	265,000
Smalltalk	3.5	10.4	22,000

Cohesion and Coupling in Place Order example.



Design Patterns

- "Don't reinvent the Wheel" : it means do not write software to solve a problem that someone else has already written to solve correctly and efficiently.
- Many companies take this approach for developing new applications.
- This will save time and money.
- OO developers look for the same reuse opportunities through the use of **Design Patterns.**

UCSC

Design Patterns

- The goal of a pattern is not to discover or invent a new solution to a problem.
- But to formally structure an existing solution to a common problem.
  - So that others may use it and take advantage of it.
- They are not :
  - Data structures that can be encoded in classes and reused *as is* (i.e., linked lists, hash tables)
  - Complex domain-specific designs (for an entire application or subsystem)

UCSC

Three Types of Patterns (GoF)

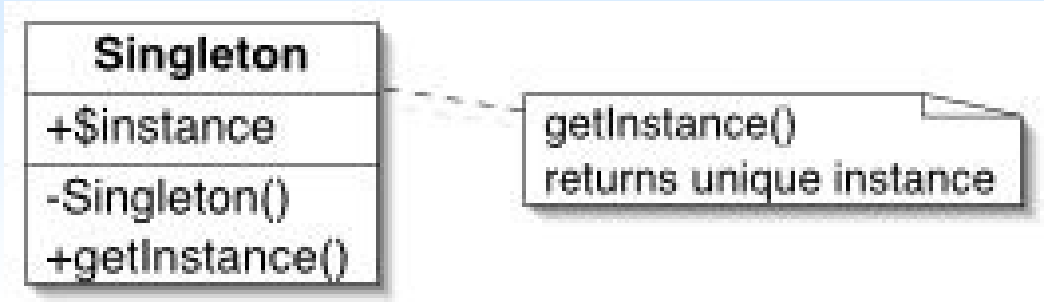
- **Creational patterns:**
  - Deal with initializing and configuring classes and objects
- **Structural patterns**:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- **Behavioral patterns**:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Three Types of Patterns (GoF)

Gang-of-Four Patterns		
Creational	**Structural**	**Behavioral**
Abstract factor	Adapter	Chain of responsibility
Builder	Bridge	Command
Factory method	Composite	Flyweight
Prototype	Decorator	Interpreter
Singleton	Façade	Iterator
	Proxy	Mediator
		Memento
		Observer
		State
		Strategy
		Template method
		Visitor

Singleton pattern (creational)

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




```
class Singleton
{ public:
      static Singleton* getInstance();
  protected:
Singleton();
      Singleton(const Singleton&);
      Singleton& operator= (const Singleton&);
 private: static Singleton* instance;
};
Singleton *p2 = p1->getInstance();
```

Ref:
https://www.youtube.com/watch?v=nh1oDnzn5yM&list=PLq1I1RpjIS59ziAx7YBqQ0rtyAcpgdsjm&index=40

Structural patterns

- Describe ways to assemble objects to realize new functionality
  - Added flexibility inherent in object composition due to ability to change composition at run-time
  - not possible with static class composition
- Example: Proxy
  - ***Proxy***: acts as convenient surrogate or placeholder for another object.
    - Remote Proxy: local representative for object in a different address space
    - Virtual Proxy: represent large object that should be loaded on demand
    - Protected Proxy: protect access to the original object

Adapter Pattern- Another Example

- SoundStage MembervServices system has to calculate sales tax on orders.
- Keeping up on all the varying laws in country or province is a difficult task.
- Therefore SoundStage will want to buy prewritten tax calculation classes and plug them into the member service system.
- They found more than one vendor who could supply them with the classes.
- Each vendors classes provides different set of methods to call.
- Design the system : if they change vendors, very few modifications need to be done.

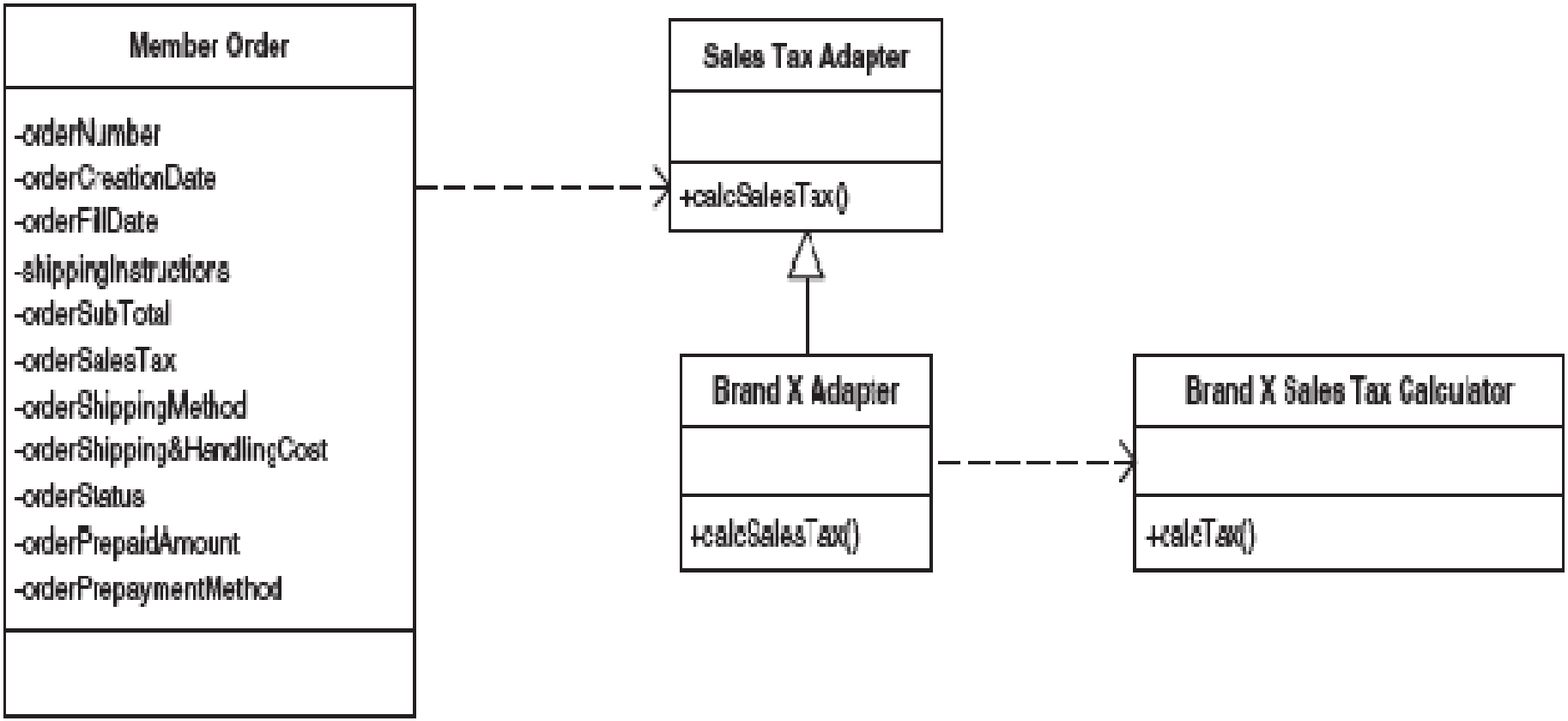
Adapter Pattern- Example cont..

- Given example : **SoundStage Member Services system**

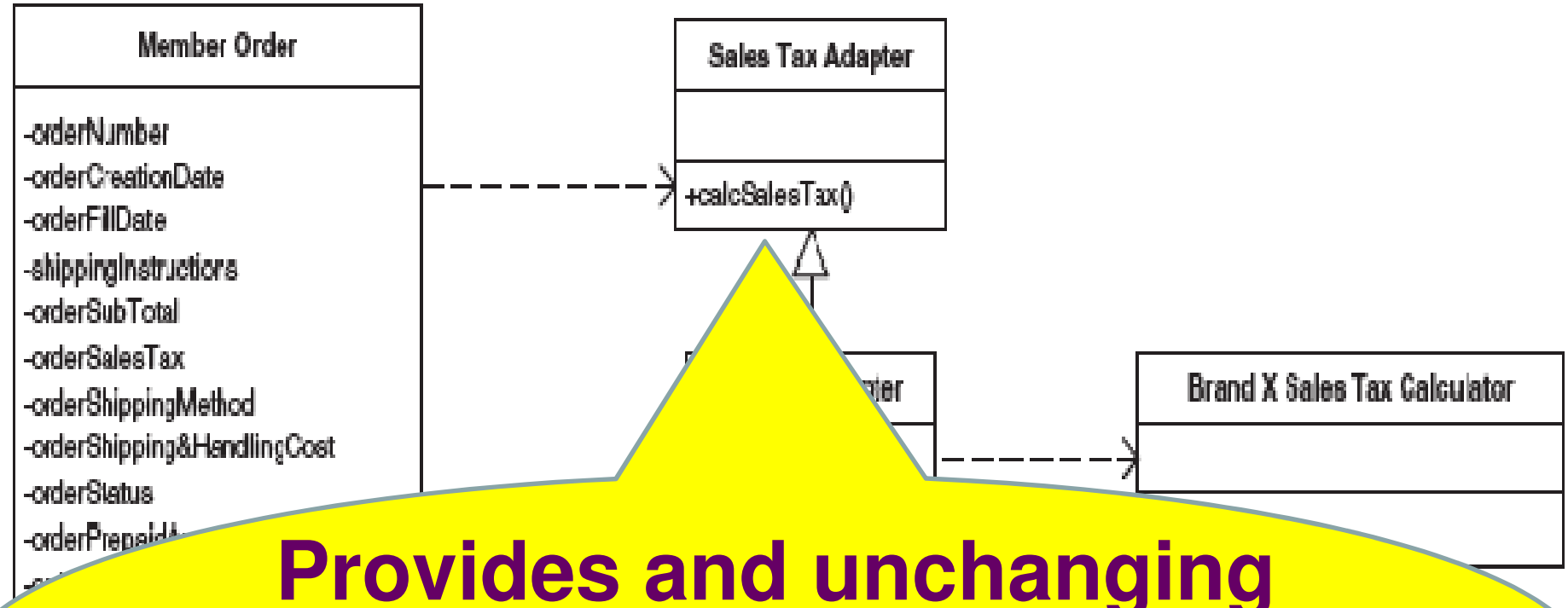
Pattern:	Adapter
Category:	Structural
Problem:	How to provide a stable interface to similar classes with different interfaces.
Solution:	Add a class that acts as an adapter to convert the interface of a class into another interface that the client classes expect.

Adapter Pattern describe ways to assemble objects to realize new functionality

Implementation of Adapter Pattern for SoundStage



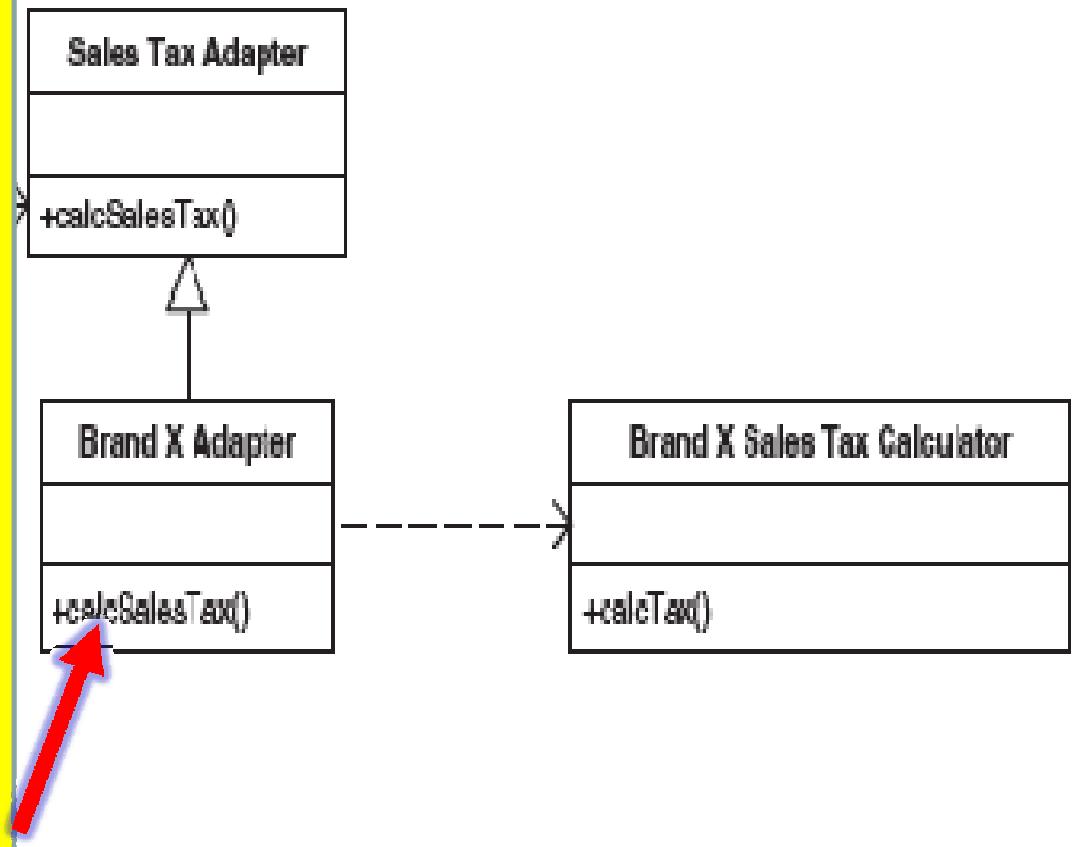
Implementation of Adapter Pattern for SoundStage



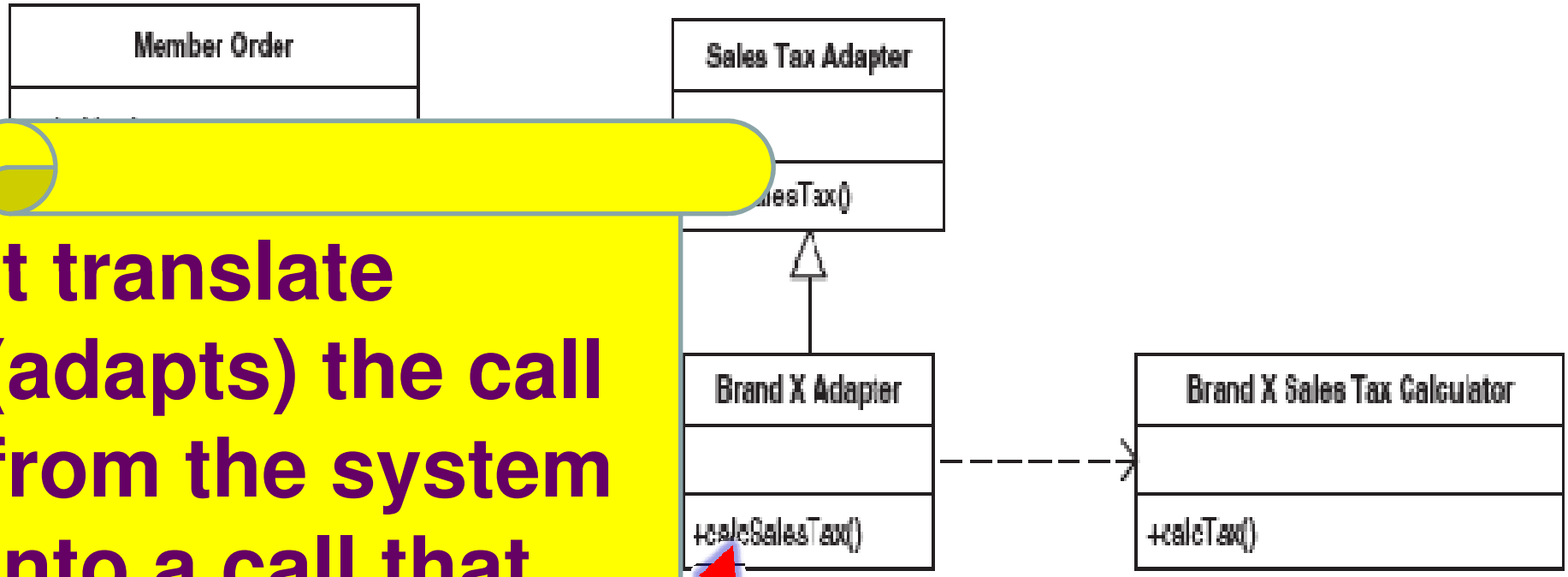
Provides and unchanging method (calcSalesTax) for the rest of the system to call

Implementation of Adapter Pattern for SoundStage

To integrate in the purchased class(Brand X Sales Tax Calculator) We write a new class (Brand X Adapter) that inherits from Sales Tax Adapter class



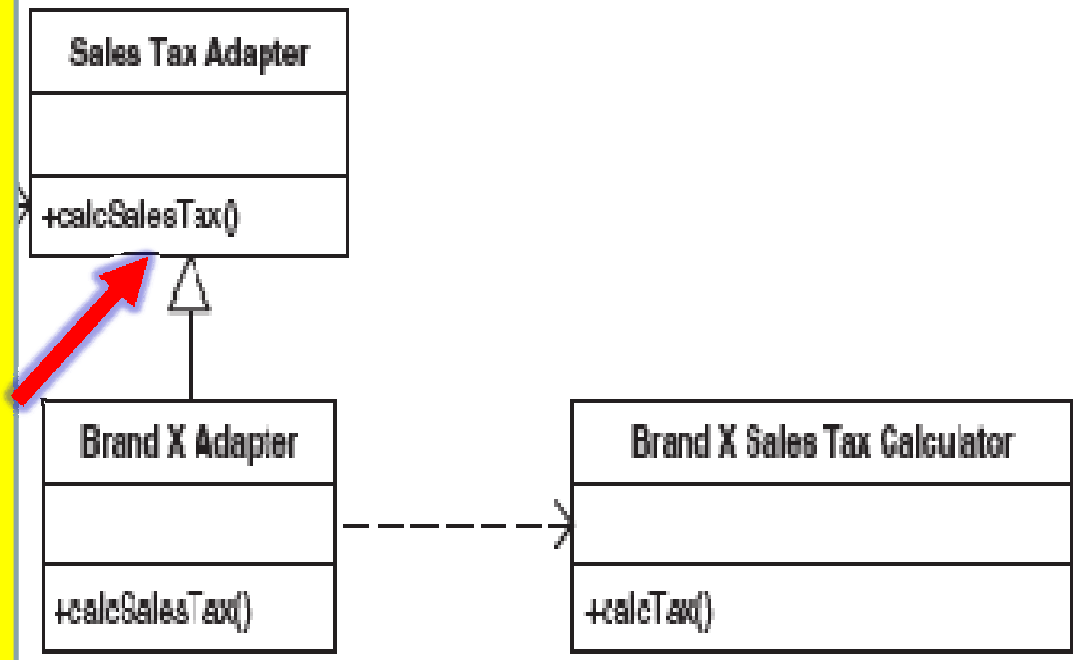
Implementation of Adapter Pattern for SoundStage



It translate (adapts) the call from the system into a call that purchased class can accept.

Implementation of Adapter Pattern for SoundStage

If we ever change the vendors, then we have only to write a new adapter subtype; everything else stays the same.



Behavioral patterns:

- Deal with dynamic interactions among societies of classes and objects
- Provide guidance on the way in which classes interact to distribute responsibility.

The Strategy Pattern

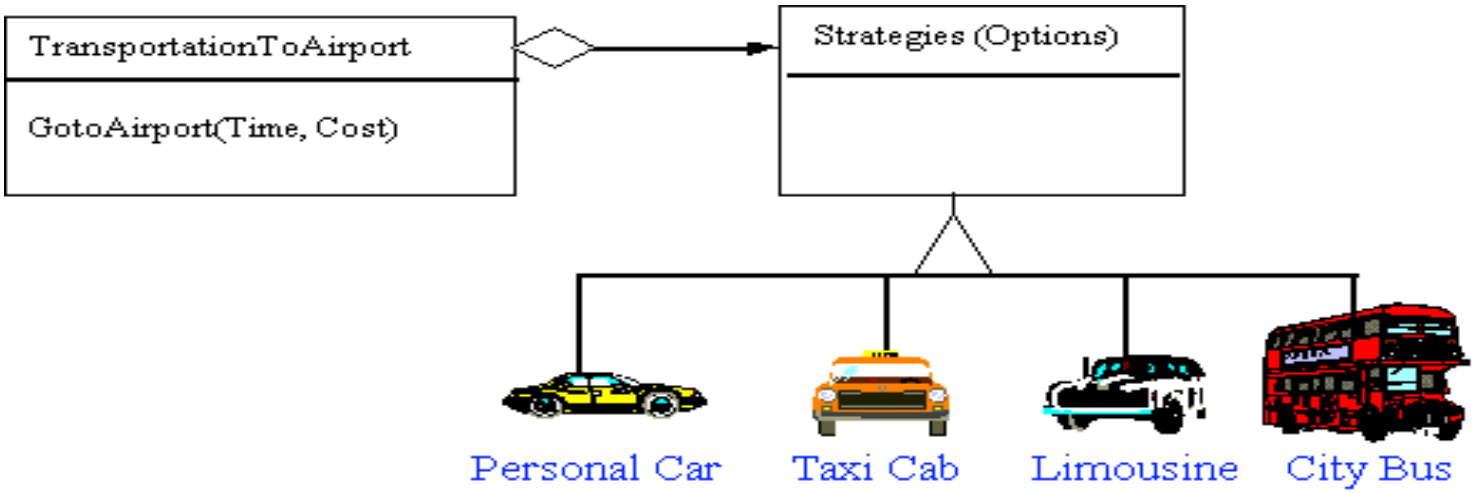
Pattern:	Strategy
Category:	Behavioral
Problem:	How to design for varying and changing policy algorithms.
Solution:	Define each algorithm in a separate class with a common interface.

Example

Modes of transportation to an airport is an example of a Strategy.

Several options exist, such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service.

Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must choose the Strategy based on tradeoffs between cost, convenience, and time.



UCSC

Another Example

- Say a company X is running **Promotions**
- **When a member places an order may use any number of promotions**
  - Based on total $ amount of the order
  - Based on number units they purchased
  - Based on the kind of the product ordered.
  - Some provide a % discount
  - Some $ amount discount etc.

Strategy Pattern : Promotion Example cont..

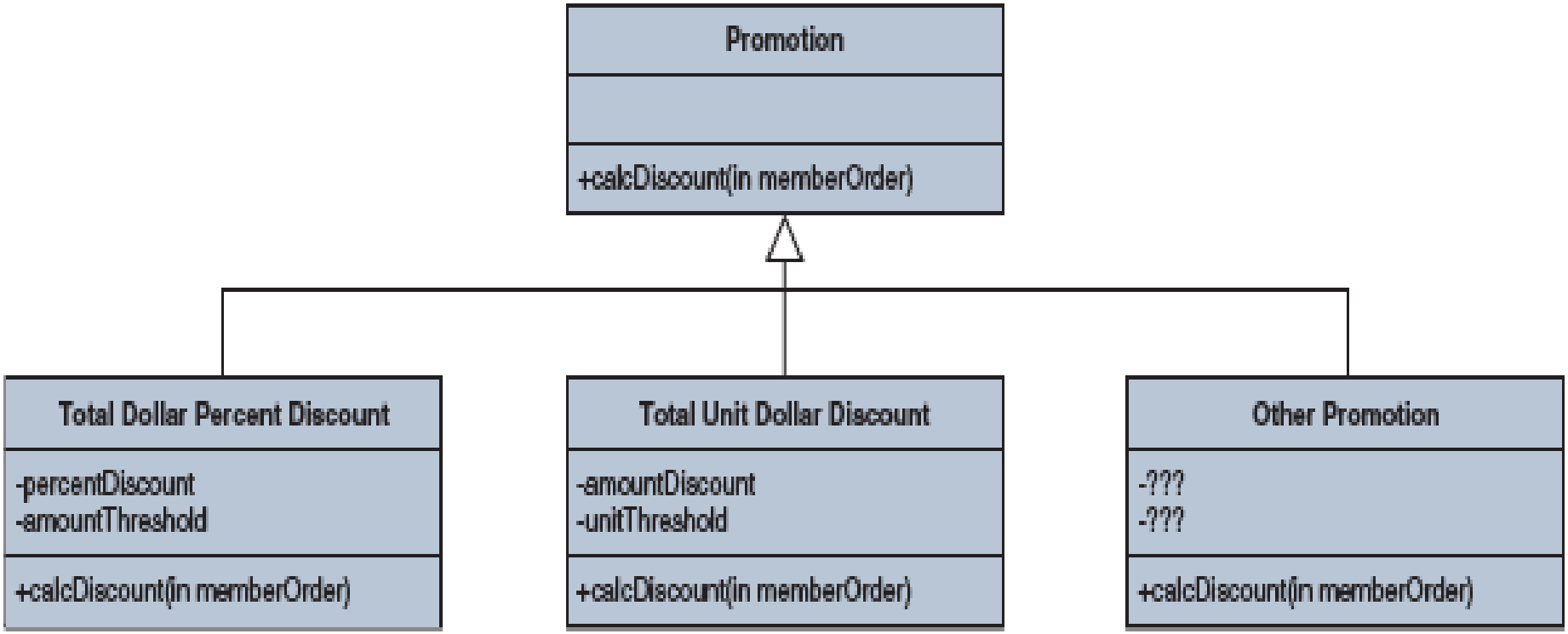
- Programming code to apply to each kind of promotion is significant.
- More importantly it is constantly changing – (new promotions from marketing people)

Strategy Pattern : Promotion Example cont..

- How can the system incorporate existing and new promotions without constantly rewriting code/classes?

Pattern:	Strategy
Category:	Behavioral
Problem:	How to design for varying and changing policy algorithms.
Solution:	Define each algorithm in a separate class with a common interface.

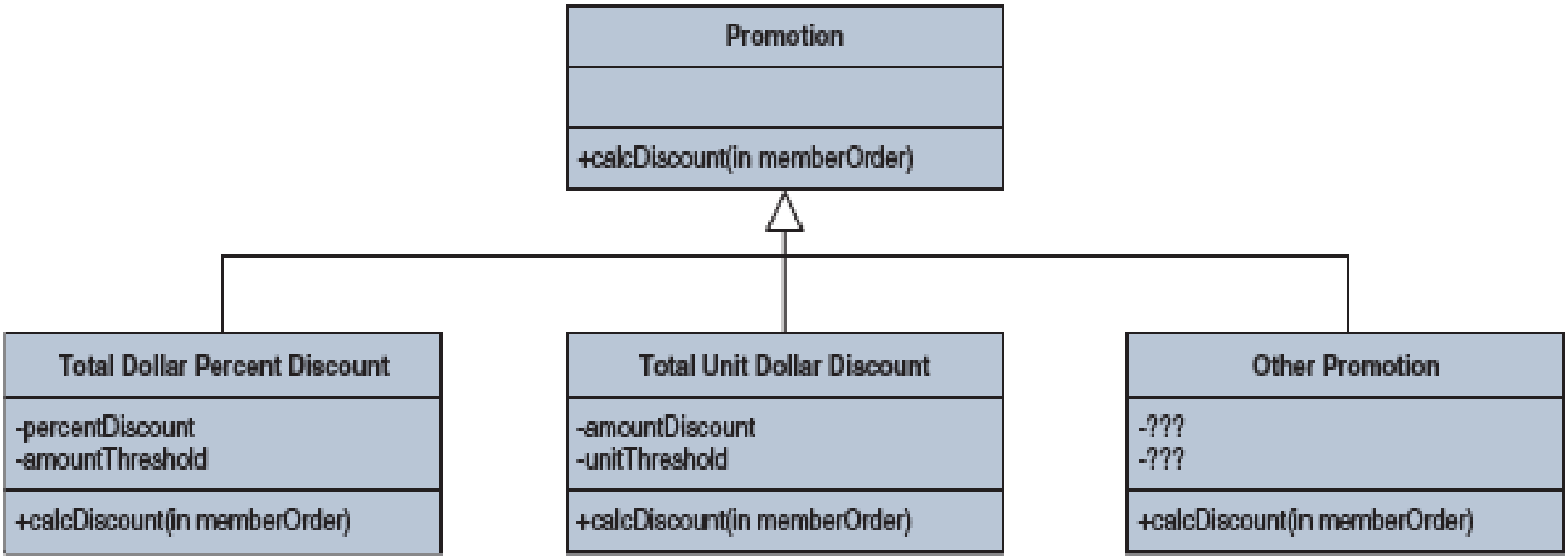
Promotion Example cont..



Each class has a standard interface method called **calcDiscount to return the $ amount.**

UCSC

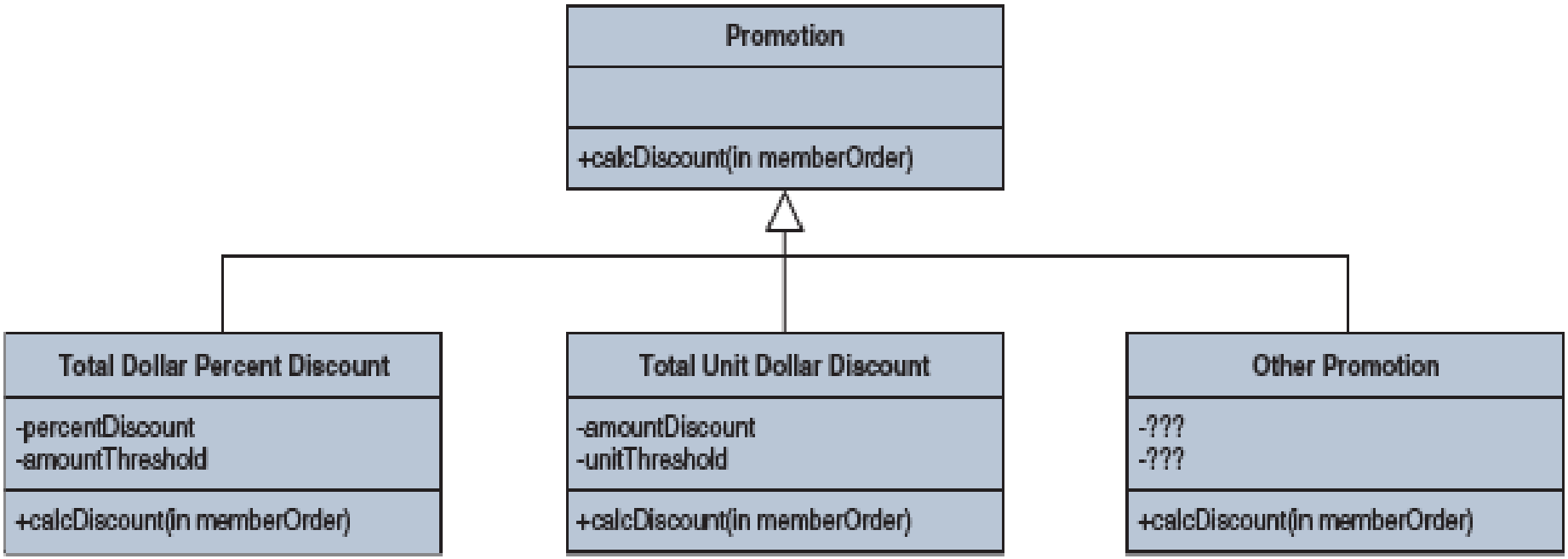
Strategy Pattern : Promotion Example cont..



Internal code to calculate each promotion will be entirely different for each promotion class.

UCSC

Strategy Pattern : Promotion Example cont..



CalcDiscount method is designed to be passed to the entire member order instance as a parameter.