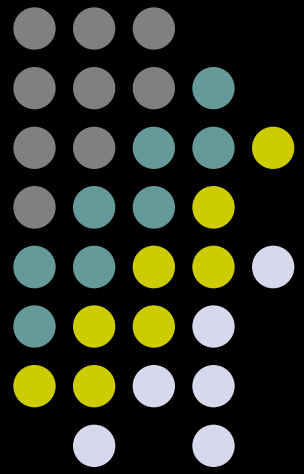
Pure Fabrication and "Gang of Four" Design Patterns

Larman, chapters 25 and 26



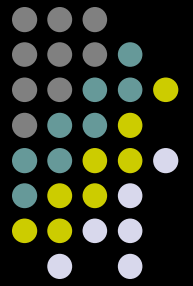
Majority of slides are taken from CSE 432: Object-Oriented Software Engineering class from Lehigh University (http://www.cse.lehigh.edu/~glennb/oose/oose.htm)

Pure Fabrication
Another GRASP design pattern

Problem: you must assign a responsibility to a class, but assigning it to a class that already represents a problem domain entity would ruin its low coupling and/or high cohesion.

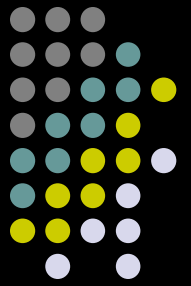
Solution: Assign a highly cohesive set of responsibilities to “made up,” fabricated class—it does not need to represent a problem domain concept—in order to support high cohesion, low coupling & reuse.

Pure Fabrication: Example

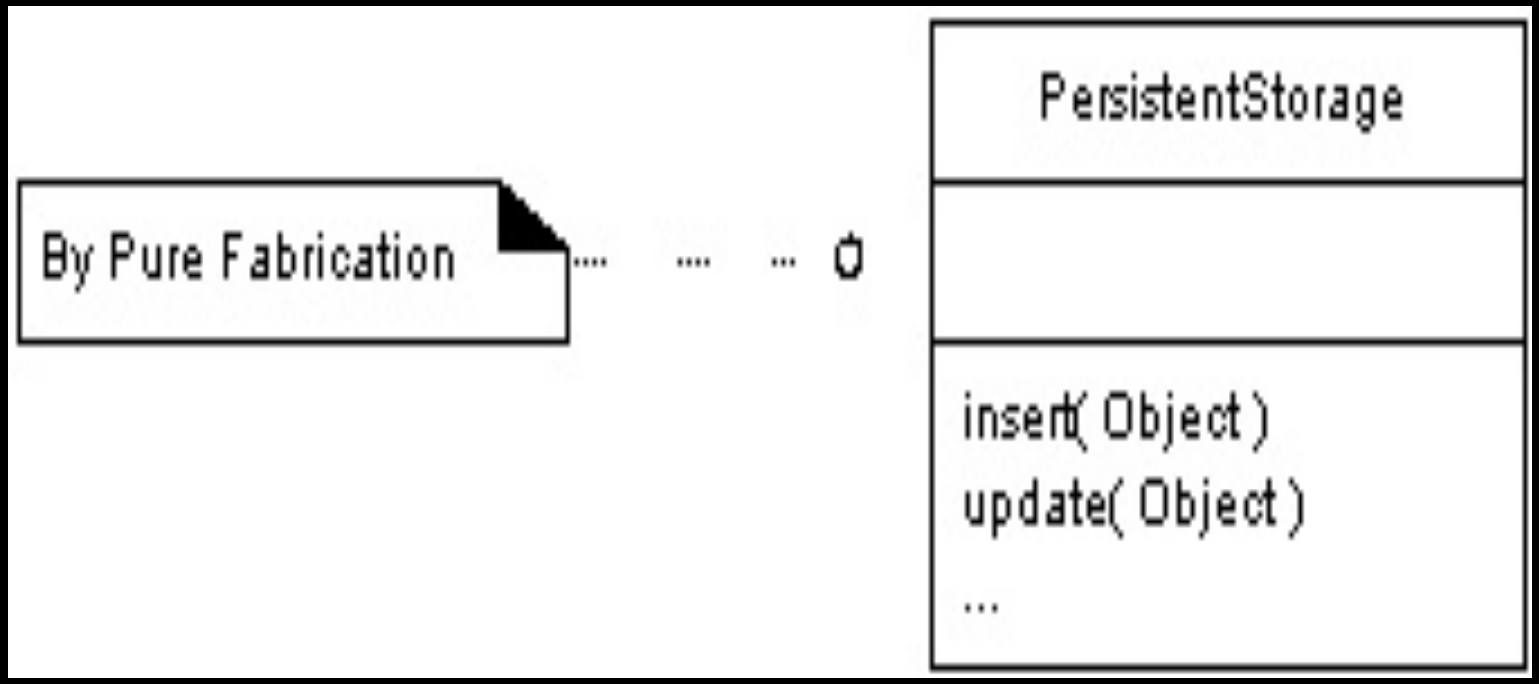


- For NextGen POS, there is a requirement to save the Sale instances in a database.
- What does Information Expert tell us to do?
- Assign this responsibility to the Sale class, since Sale has the data that needs to be saved.
- But the above solution leads to:
  - Low Cohesion: database tasks not related to Sale
    High Coupling: Should Sale interface with database?
  - Low Reusability: General task of saving to a database is one that other classes may share.

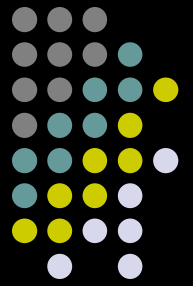
Instead, fabricate a new class, PersistentStorage



Invent a new class that is solely responsible for saving objects.



Pure Fabrication Benefits

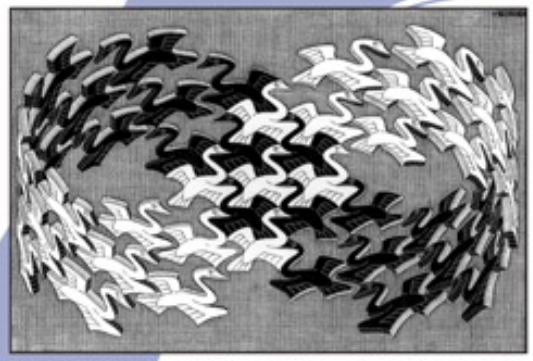


- High Cohesion: fabricated class focuses on a very specific responsibility
- Reuse: fine-grained pure fabrication classes with specific responsibilities are relatively easy to understand and reuse in other applications
- Pure fabrication principle leads to many other reusable design patterns, including most of the Gang of Four patterns

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

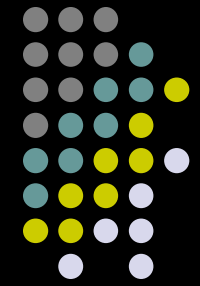


Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

1995

The “gang of four” (GoF)



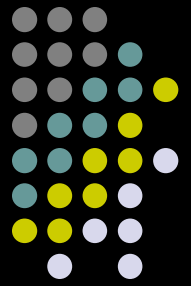
Design Patterns book catalogs 23 different patterns

- Solutions to different classes of problems, in C++ & Smalltalk
- Problems and solutions are broadly applicable, used by many people over many years
- Patterns suggest opportunities for reuse in analysis, design and programming
- GOF presents each pattern in a structured format
  - *What do you think of this format? Pros and cons?*

Elements of Design Patterns

- Design patterns have 4 essential elements:
  - Pattern name: increases vocabulary of designers
  - Problem: intent, context, when to apply
  - Solution: UML-like structure, abstract code
  - Consequences: results and tradeoffs

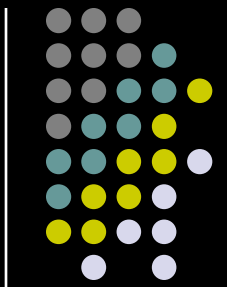
Design Patterns are NOT



- 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)
- If they are not familiar data structures or complex domain-specific subsystems, *what are they*?

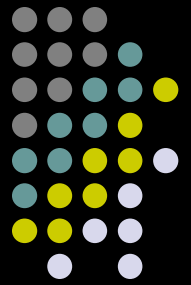
- They are:
  - "Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

Three Types of GoF Patterns



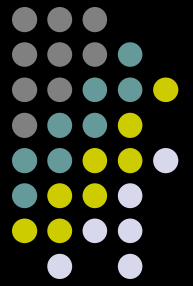
- **Creational patterns:**
  - Deal with initializing and configuring objects
- **Structural patterns**:
  - Composition of classes or objects
  - Decouple interface and implementation of classes
- **Behavioral patterns**:
  - Deal with dynamic interactions among societies of objects
  - How they distribute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

Structural patterns



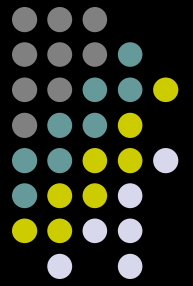
- Assemble objects to realize new functionality
  - Exploit flexibility of object composition at run-time
  - Not possible with static class composition
- Example: Proxy
  - ***Proxy*** acts as convenient surrogate or placeholder for another object.
  - Examples?
    - Remote Proxy: local representative 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

Structural Patterns



- **Adapter:**
  - Converts interface of a class into one that clients expect
- **Bridge**:
  - Links abstraction with many possible implementations
- **Composite**:
  - Represents part-whole hierarchies as tree structures
- **Decorator**:
  - Attach additional responsibilities to object dynamically
- **Facade**:
  - Simplifies the interface for a subsystem
- **Flyweight**:
  - Shares many fine-grained objects efficiently
- **Proxy**:
  - Provides a surrogate or placeholder for another object to control access to it

Adapter pattern

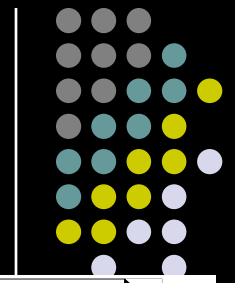


Problem: How to resolve incompatible interfaces or provide a stable interface to similar components with different interfaces?

Solution: Convert original interface component into another one through an intermediate adapter.

Use interfaces and polymorphism to add indirection to varying APIs

POS example: Instantiate adapters for external services



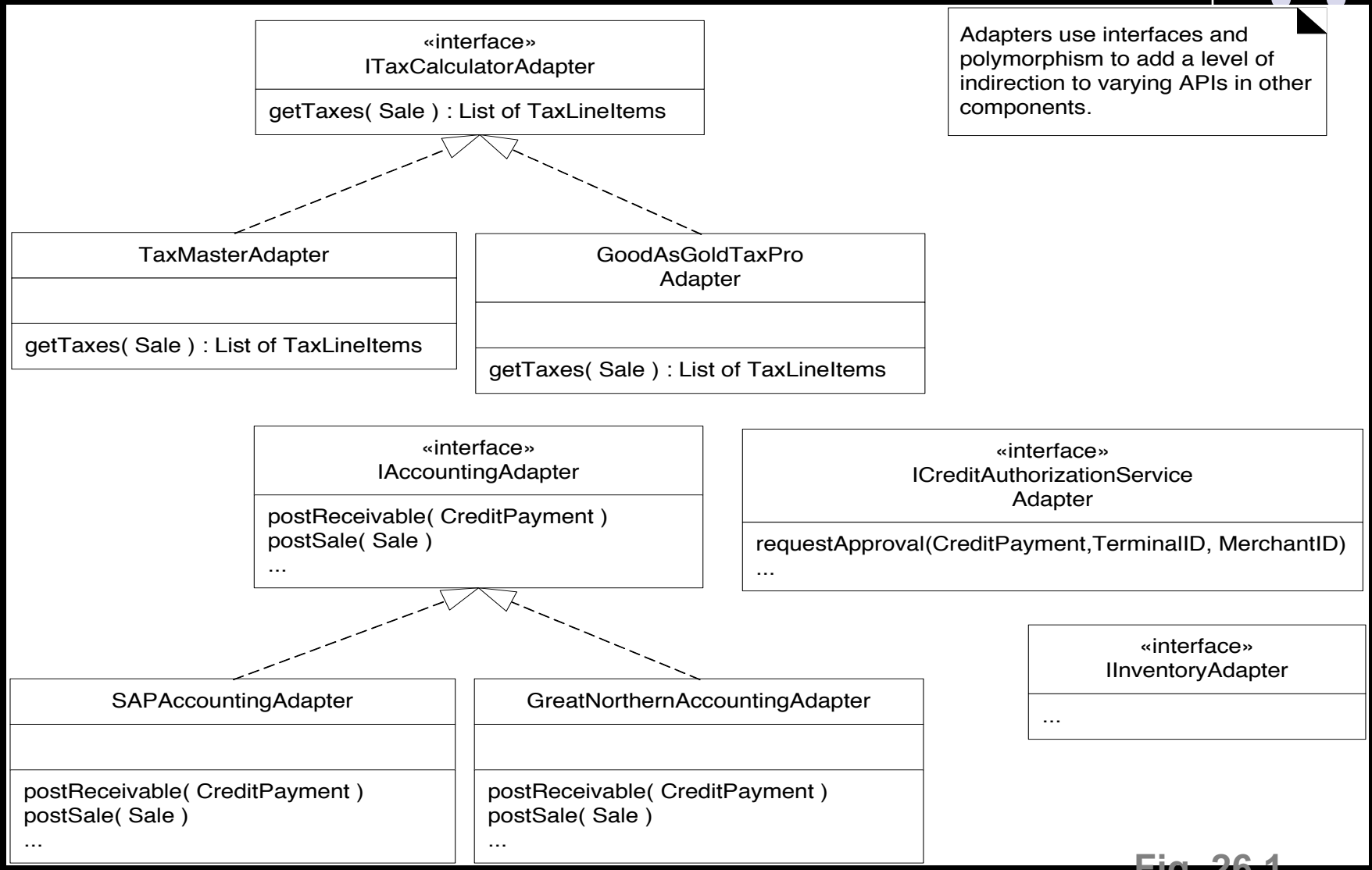


Fig. 26.1

Using an Adapter: adapt postSale request to SOAP XML interface

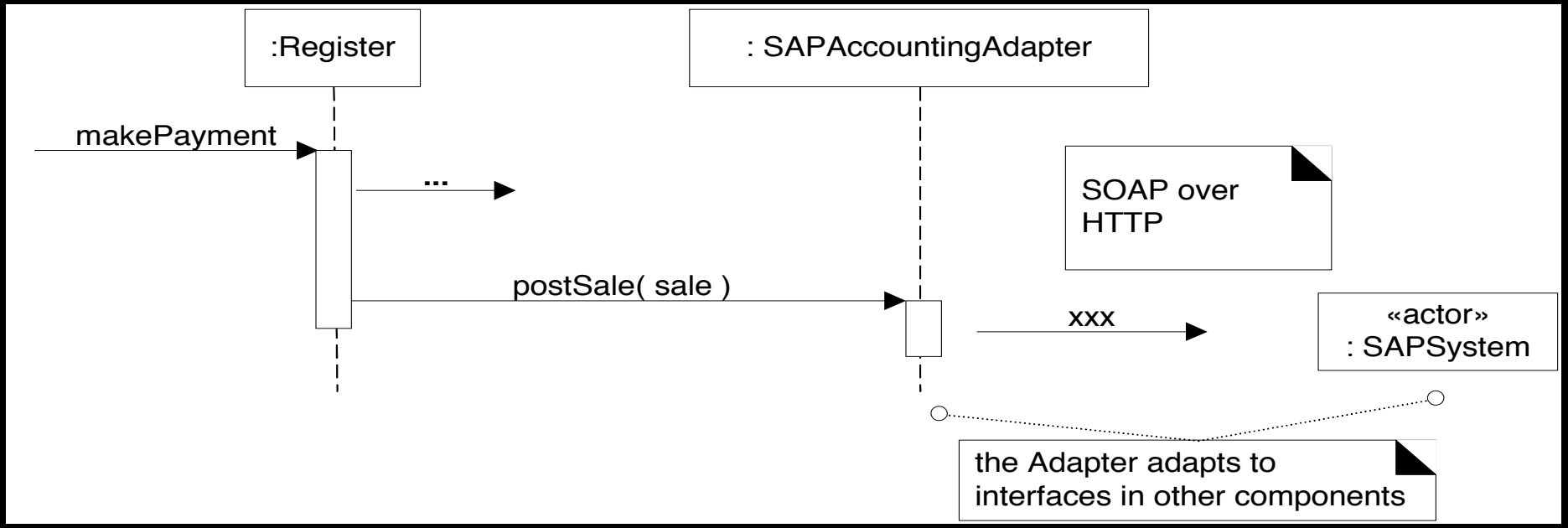
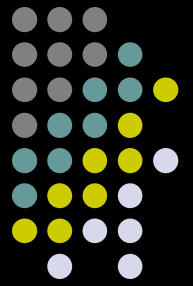


Fig. 26.2

Benefits of Adapter pattern



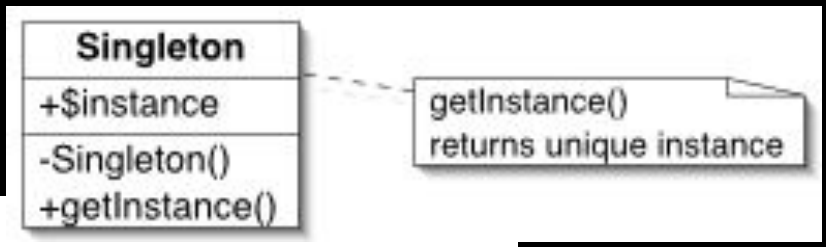
- Reduces coupling to implementation specific details
- Polymorphism and Indirection reveals essential behavior provided
- Including name of design pattern in new class (e.g., TaxMasterAdapter) in class diagrams and code communicates to other developers in terms of known design patterns

Creational Patterns

- **Singleton**: Guarantee access to a singular (sole) instance
- **Simple Factory:** Create specialized, complex objects
- **Abstract Factory**: Create a family of specialized factories
- **Factory Method**: Define an interface for creating an object, but let subclasses decide which class to instantiate
- **Builder**: Construct a complex object step by step
- **Prototype**: Clone new instances from a prototype
- **Lazy initialization**: Delay costly creation until it is needed

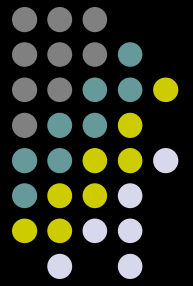
Singleton pattern (creational)

- A class with just instance and provide a global point of access
  - *Global Variables can be dangerous!*
    (side effects, break information hiding)



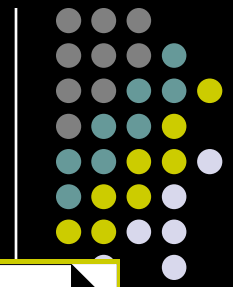

```
public class Singleton {
    private String objectState;
    private static final Singleton instance = new Singleton();
    private Singleton() {
        this.objectState = "ABCD...XYZ";
        // Setting some random object objectState
    }
    public static Singleton getInstance() {
        return instance;
    }
    public String getObjectState() {
        return objectState;
    }
    public void setObjectState(String objectState) {
        this.objectState = objectState;
    }
}
```

Simple Factory pattern



- Context/Problem
  - Who should be responsible for creating objects when there are special considerations, such as complex logic,a desire to separate the creation responsibilities for better cohesion, and so forth
- Solution
  - Create a Pure Fabrication to handle the creation

Factory can create different objects from a file



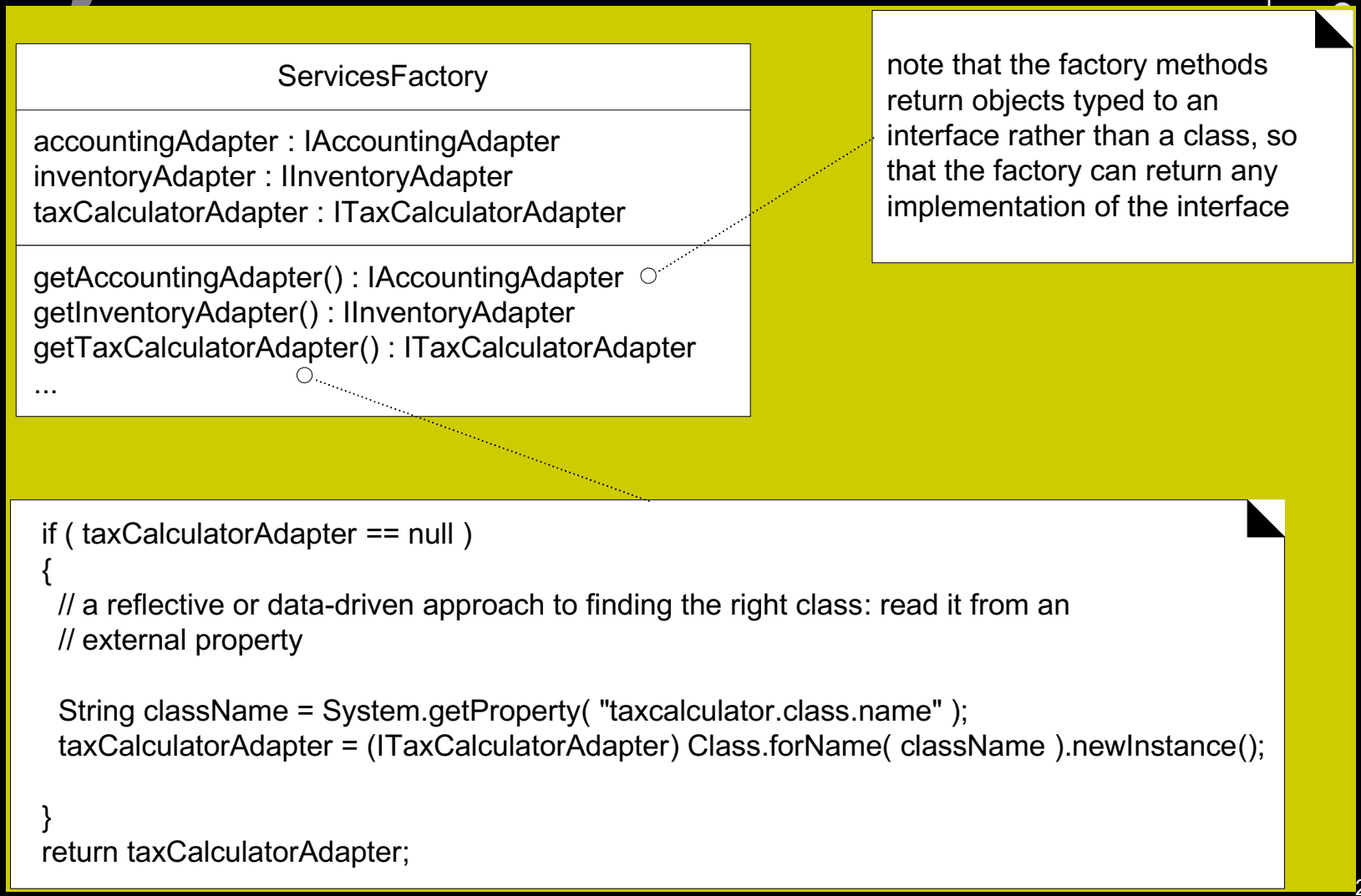
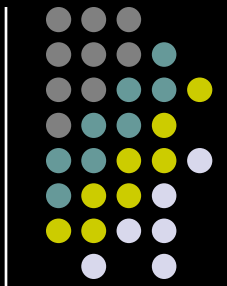


Figure 26.5

Advantages of Factory Objects?



- Separates responsibility of complex creation into cohesive helper classes
- Hides complex creation logic, such as initialization from a file
- Handles memory management strategies, such or recycling or caching

Use Singleton to create a Factory

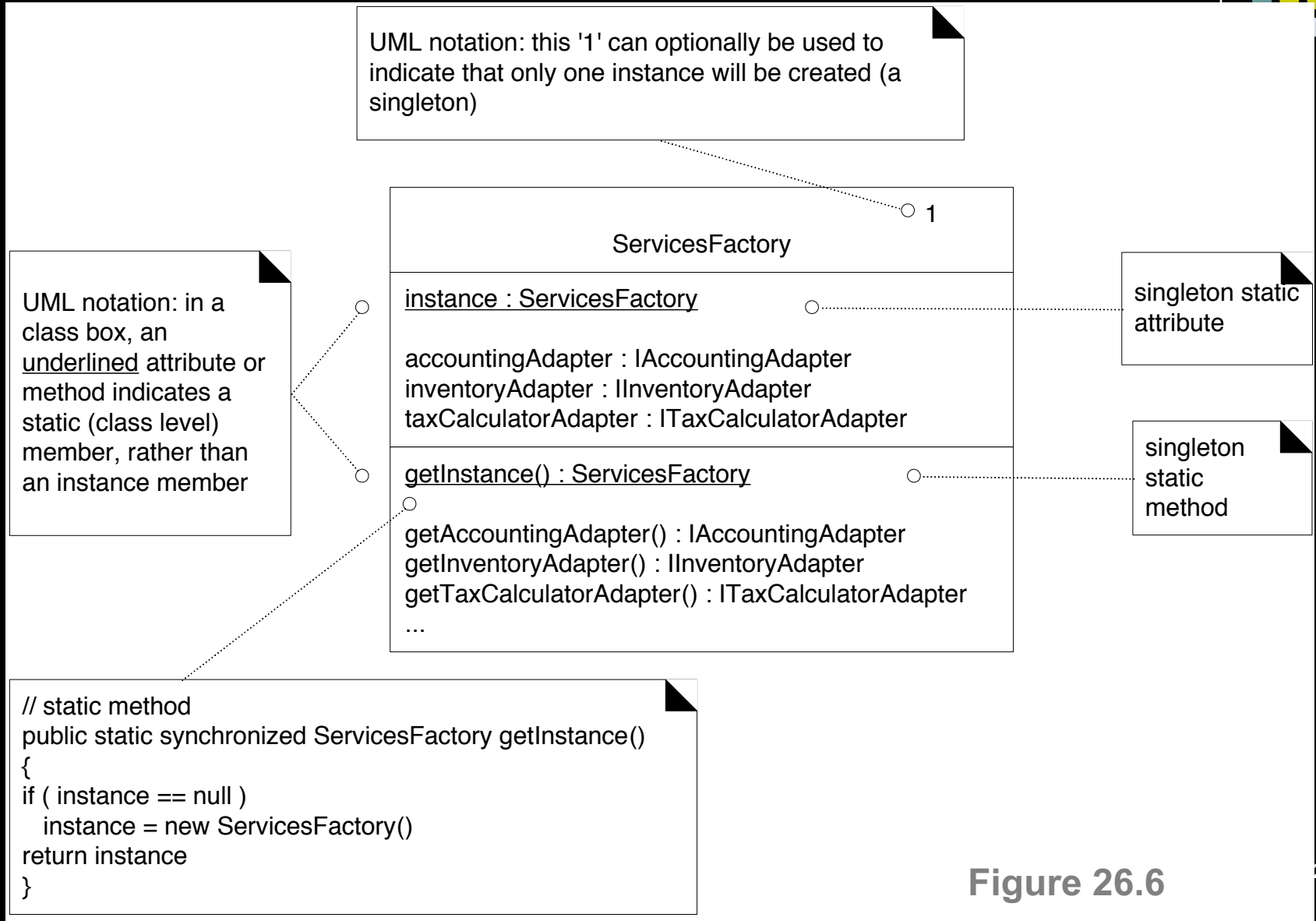


Figure 26.6

Adapter, Factory and Singleton working together

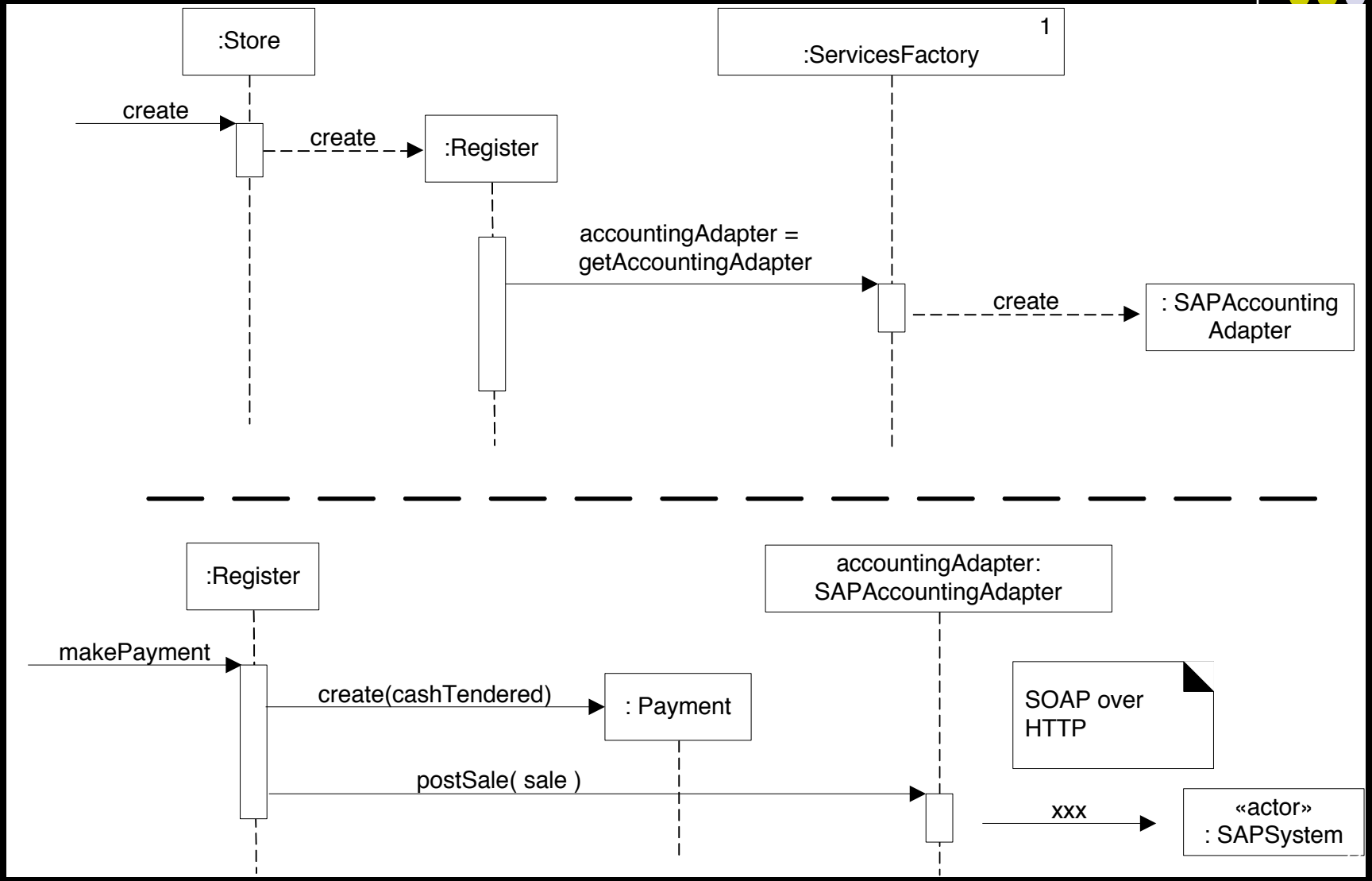
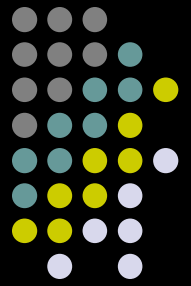


Figure 26.8

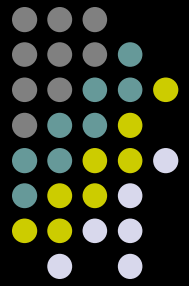
Behavioral Patterns



- **Chain of Responsibility**:
  - Request delegated to the responsible service provider
- **Command**:
  - Request or Action is first-class object, hence storable
- **Iterator**:
  - Aggregate and access elements sequentially
- **Interpreter**:
  - Language interpreter for a small grammar
- **Mediator**:
  - Coordinates interactions between its associates
- **Memento**:
  - Snapshot captures and restores object states privately

Which ones do you think you have seen somewhere?

Behavioral Patterns (cont.)



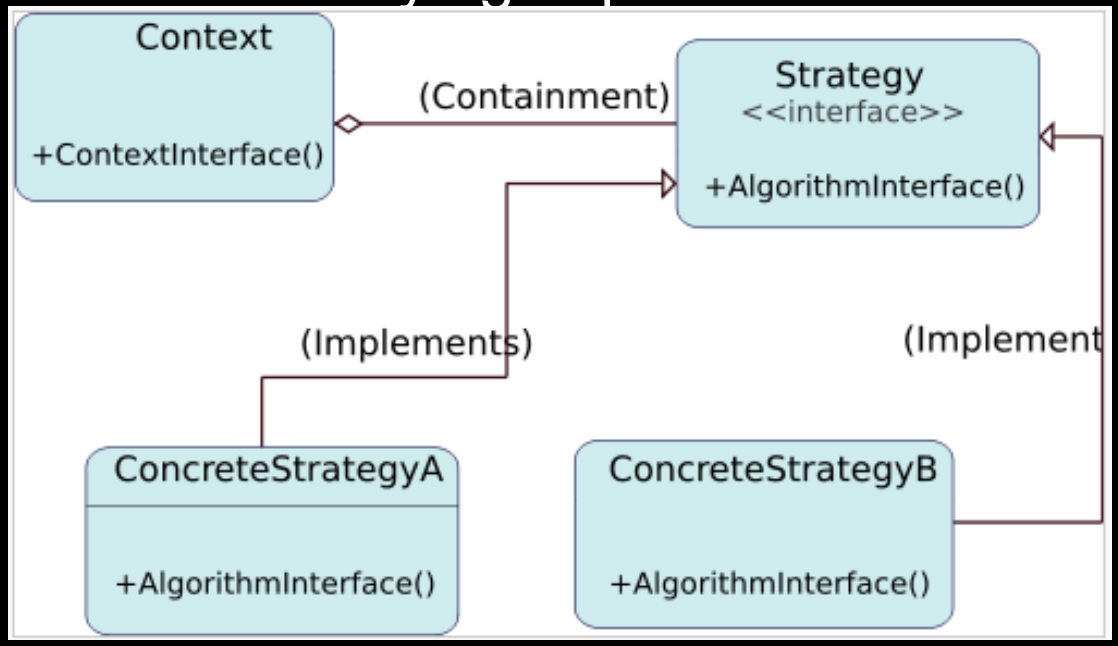
- **Observer**:
  - Observers update automatically when observed object changes
- **State**:
  - Object whose behavior depends on its state
- **Strategy**:
  - Abstraction for selecting one of many algorithms
- **Template Method**:
  - Algorithm with some steps supplied by derived class
- **Visitor**:
  - Operations applied to elements of a heterogeneous object structure

Strategy design pattern

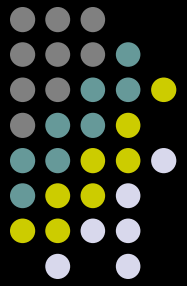
Problem: How to design a family of algorithms or policies that are essentially the same but vary in details?

Solution: "Define a family of algorithms, encapsulate each one, and make them interchangeable." [Gamma, p315]

- Use abstraction and polymorphism to show high level algorithm and hide varying implementation details



Multiple *SalePricingStrategy* classes with polymorphic *getTotal* method



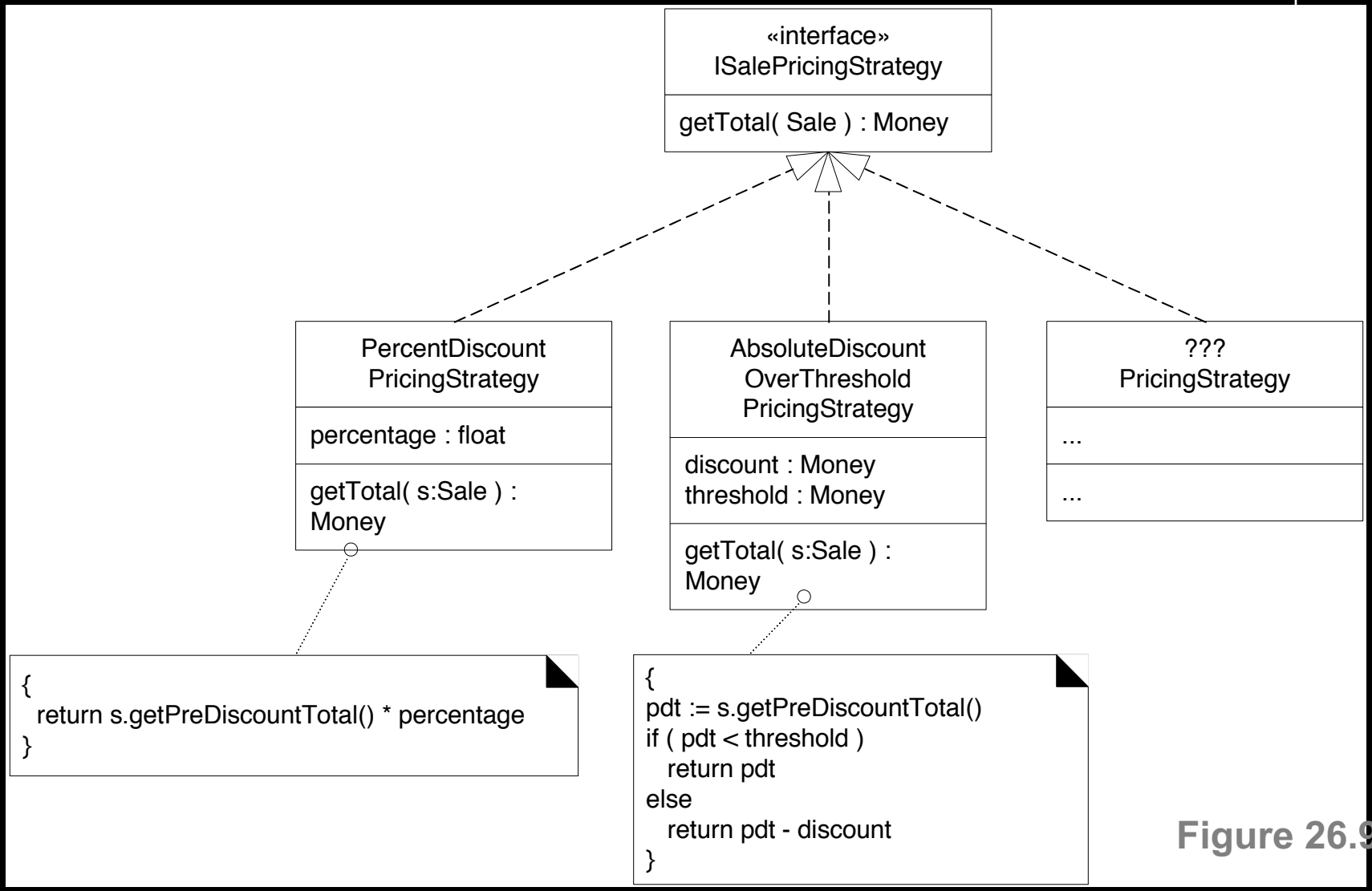
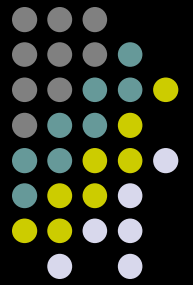


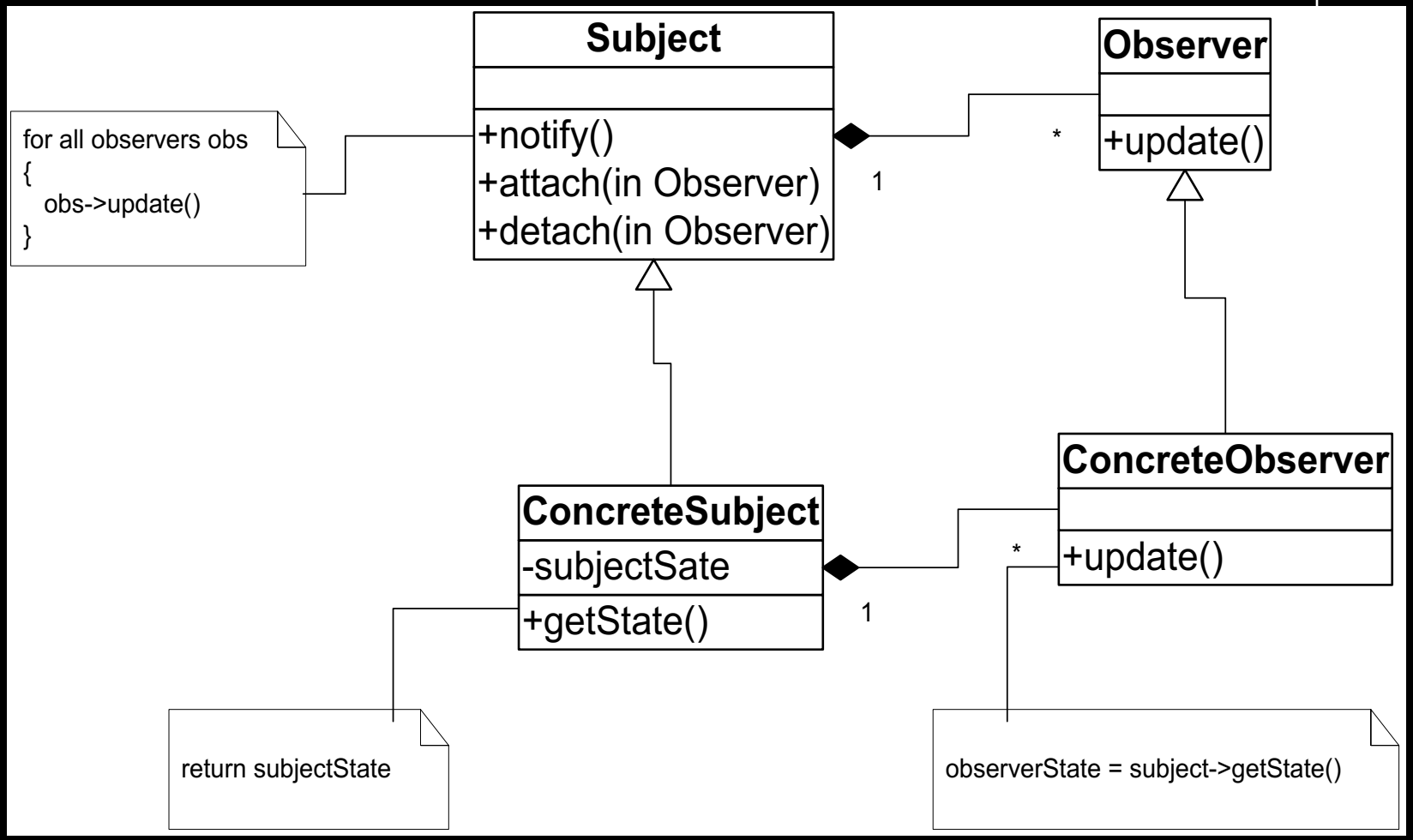
Figure 26.9

Observer pattern

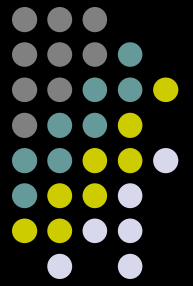


- Intent:
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Used in Model-View-Controller framework
  - Model is problem domain
  - View is windowing system
  - Controller is mouse/keyboard control
- *How can Observer pattern be used in other applications?*
- JDK's Abstract Window Toolkit (listeners)
- Java's Thread monitors, notify(), etc.

Structure of Observer Pattern

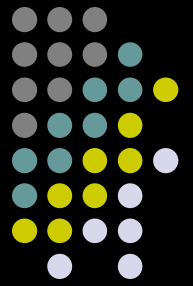


Patterns in software libraries



- AWT and Swing use Observer pattern
- Iterator pattern in C++ template library & JDK
- Façade pattern used in many student-oriented libraries to simplify more complicated libraries!
- Bridge and other patterns recurs in middleware for distributed computing frameworks
- …

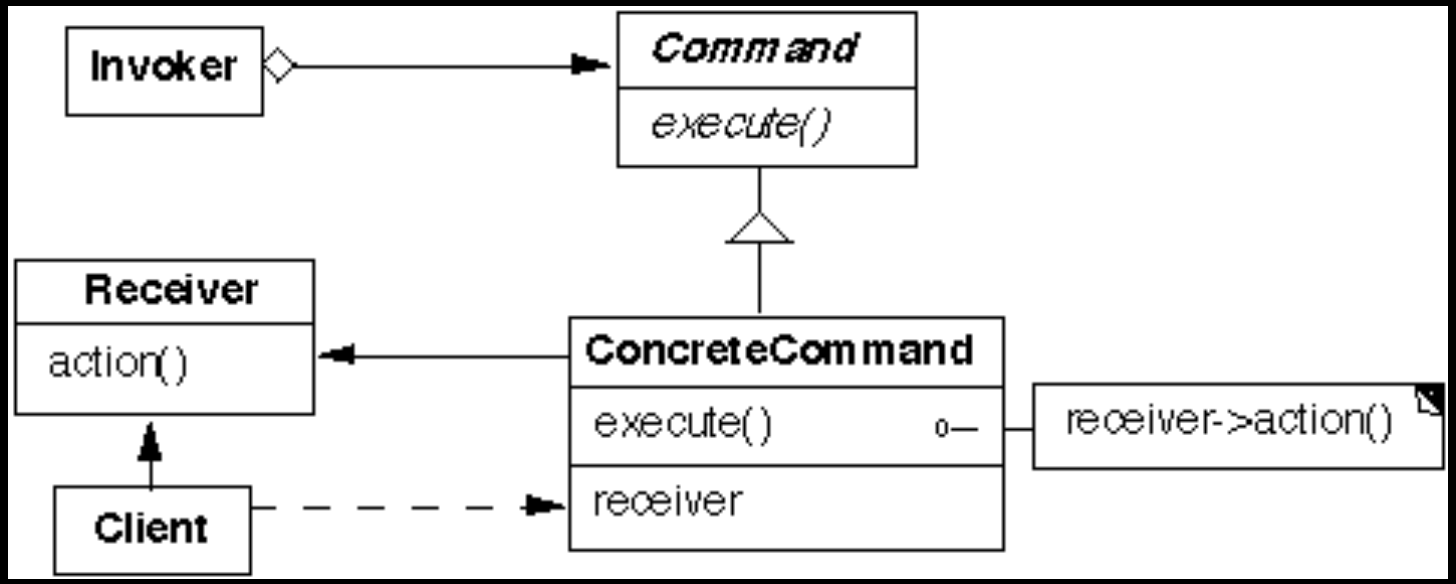
Command pattern



- **Synopsis** or **Intent**: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- **Context**: You want to model the time evolution of a program:
  - What needs to be done, e.g. queued requests, alarms, conditions for action
  - What is being done, e.g. which parts of a composite or distributed action have been completed
  - What has been done, e.g. a log of undoable operations
- *What are some applications that need to support undo?*
  - Editor, calculator, database with transactions
  - Perform an execute at one time, undo at a different time
- **Solution**: represent units of work as Command objects
  - Interface of a Command object can be a simple execute() method
  - Extra methods can support undo and redo
  - Commands can be persistent and globally accessible, just like normal objects

Command pattern, continued

- **Structure**:



Participants (the classes and/or objects participating in this pattern):

Command (Command) declares an interface for executing an operation

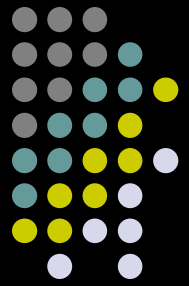
ConcreteCommand defines a binding between a Receiver object and an action
implements Execute by invoking the corresponding operation(s) on Receiver

Invoker asks the command to carry out the request

Receiver knows how to perform operations associated with carrying out the request

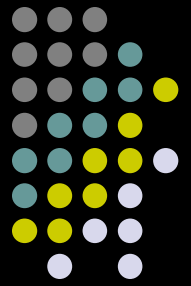
Client creates a ConcreteCommand object and sets its receiver

Command pattern, continued



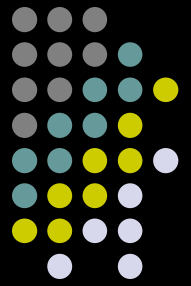
- **Consequences:**
  - You can undo/redo any Command
    - Each Command stores what it needs to restore state
  - You can store Commands in a stack or queue
    - Command processor pattern maintains a history
  - It is easy to add new Commands, because you do not have to change existing classes
    - Command is an abstract class, from which you derive new classes
    - execute(), undo() and redo() are polymorphic functions

More software patterns



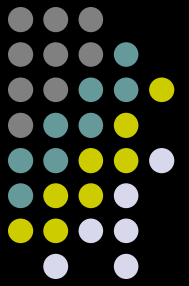
- Language idioms (low level, C++): Jim Coplein, Scott Meyers
  - I.e., when should you define a virtual destructor?
- Architectural (systems design): layers, reflection, broker
  - Reflection makes classes self-aware, their structure and behavior accessible for adaptation and change: Meta-level provides self-representation, base level defines the application logic
- *Java Enterprise Design Patterns* (distributed transactions and databases)
  - E.g., ACID Transaction: *A*tomicity (restoring an object after a failed transaction), *C*onsistency, *I*solation, and *D*urability
- **Analysis patterns** (recurring & reusable analysis models, from various domains, i.e., accounting, financial trading, health)
- **Process patterns** (software process & organization)

Benefits of Design Patterns



- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary

Web Resources



- http://home.earthlink.net/~huston2/dp/
- http://www.dofactory.com/
- http://hillside.net/patterns/