

# **CSSE2003**

## **Software Engineering Studio**

Semester 2, 2009

### **12: Software Design Patterns -2**

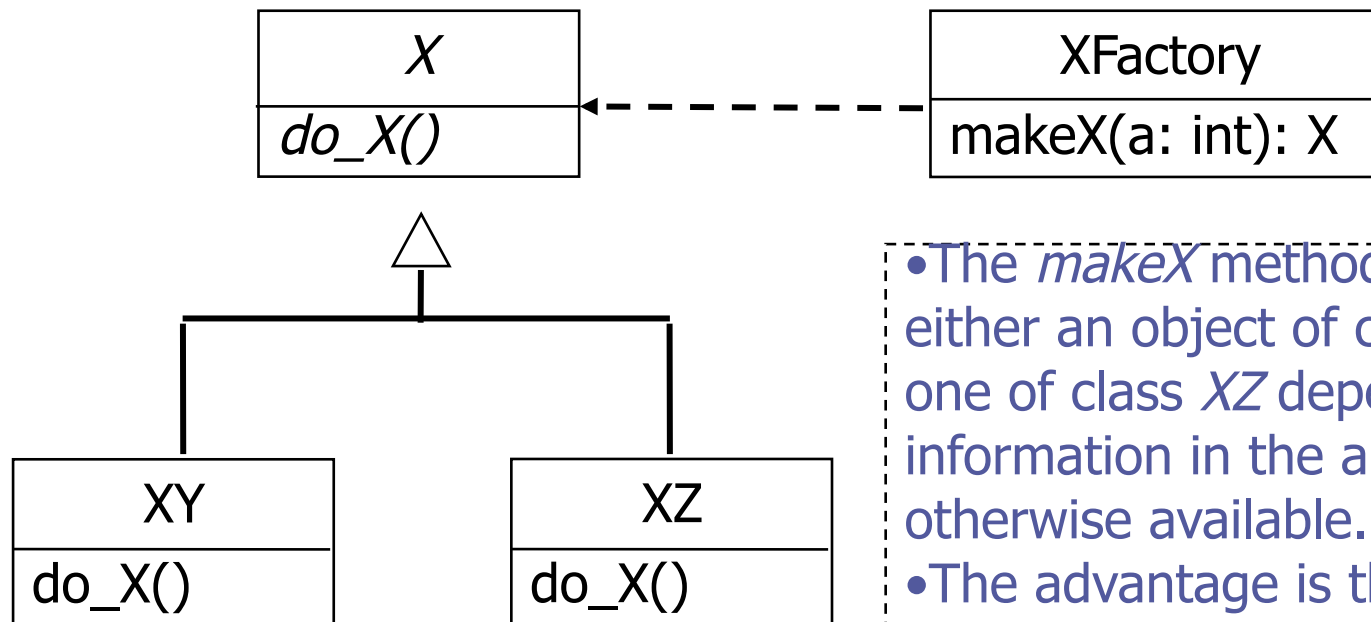
# Lecture Summary

- ◆ GoF creational patterns
  - Builder pattern
  - Factory Method pattern
  - Abstract Factory pattern
  - Singleton pattern
- ◆ GoF behavioural patterns
  - Operation Patterns
    - ◆ State
    - ◆ Strategy

# GoF: Creational Patterns

- ◆ **Factory method** - defer the decision about which class to instantiate
- ◆ **Abstract factory** - construct a family of objects that share some commonality
- ◆ **Singleton** - ensure class has only one instance
- ◆ **Builder** - gather information for an object before requesting its construction

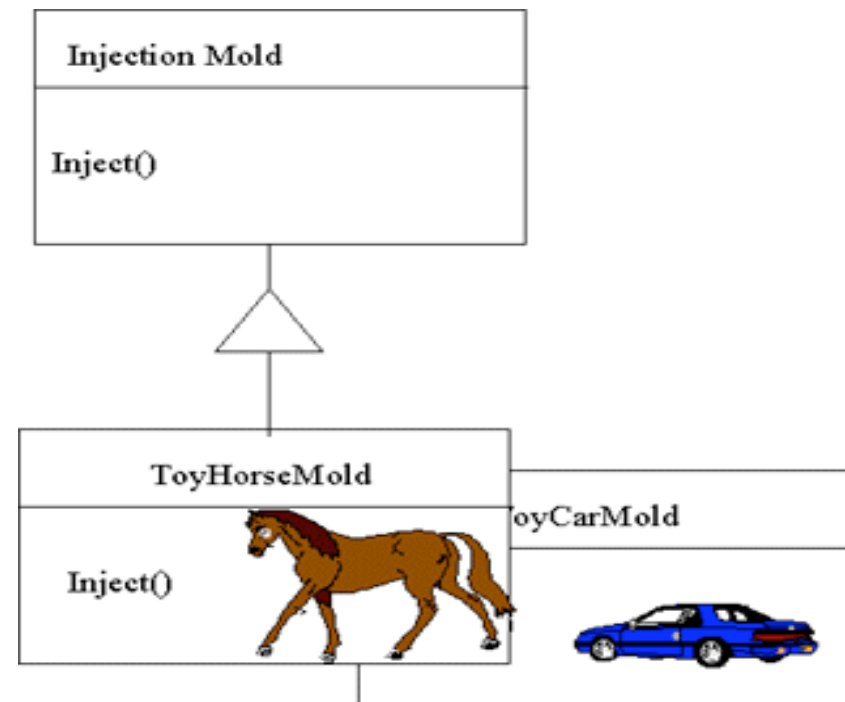
# Simple Factory Example



- The *makeX* method returns either an object of class *XY* or one of class *XZ* depending on information in the argument or otherwise available.
- The advantage is that the client of the *XFactory* class does not need to know how the decision is made, merely that it gets back an object conforming to the class (or interface) *X*.

# Factory Method Pattern - 1

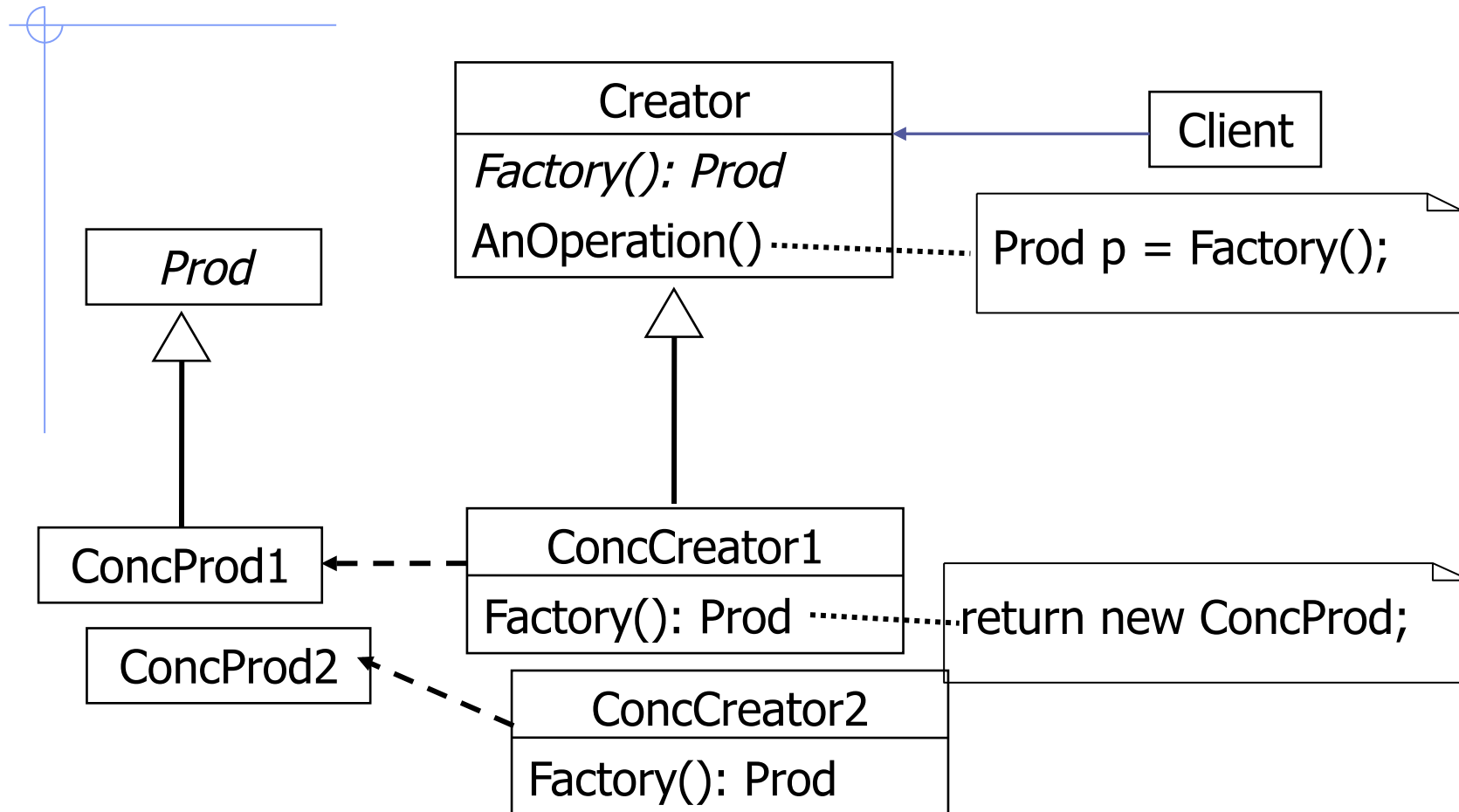
- ◆ Defines an interface for creating an object, but let subclasses decide which class to instantiate.
- ◆ Lets a class defer instantiation to subclasses.
- ◆ An increasingly popular definition of factory method is a static method of a class that returns an object of that class type.
- ◆ SW example: a framework for applications



## Factory Method Pattern - 2

- ◆ Unlike a constructor,
  - the actual object it returns might be an instance of a subclass.
  - an existing object might be reused, instead of a new object created.
  - factory methods can have different and more descriptive names

# Factory Method Structure



# Factory Method Applicability

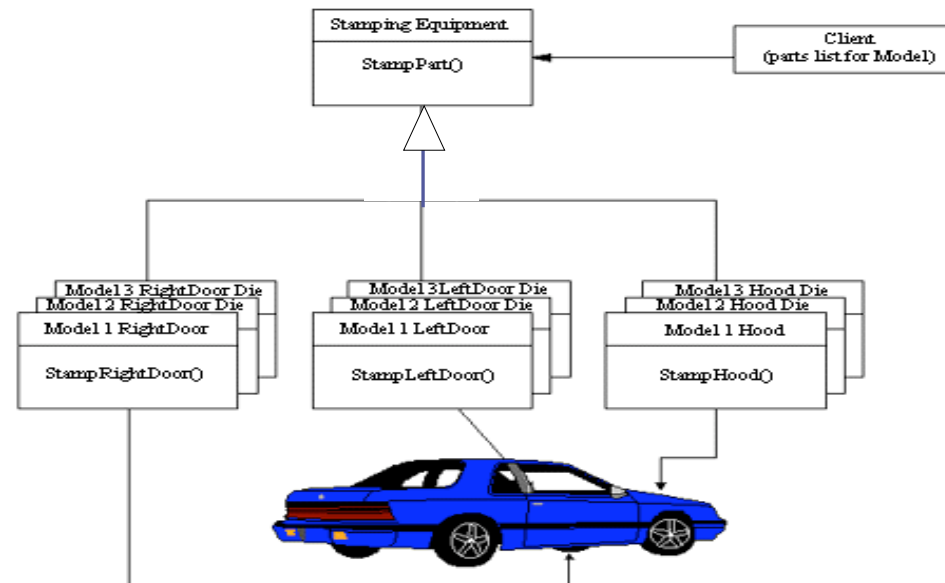
- ◆ Use a Factory Method when
  - a class cannot anticipate the class of objects it must create,
  - a class wants its subclasses to specify the objects it creates, or
  - classes delegate responsibility to one of several helper subclasses which localises the knowledge of which helper subclass is the delegate.

Indirection, decoupling, and polymorphic creation

# Abstract Factory Pattern

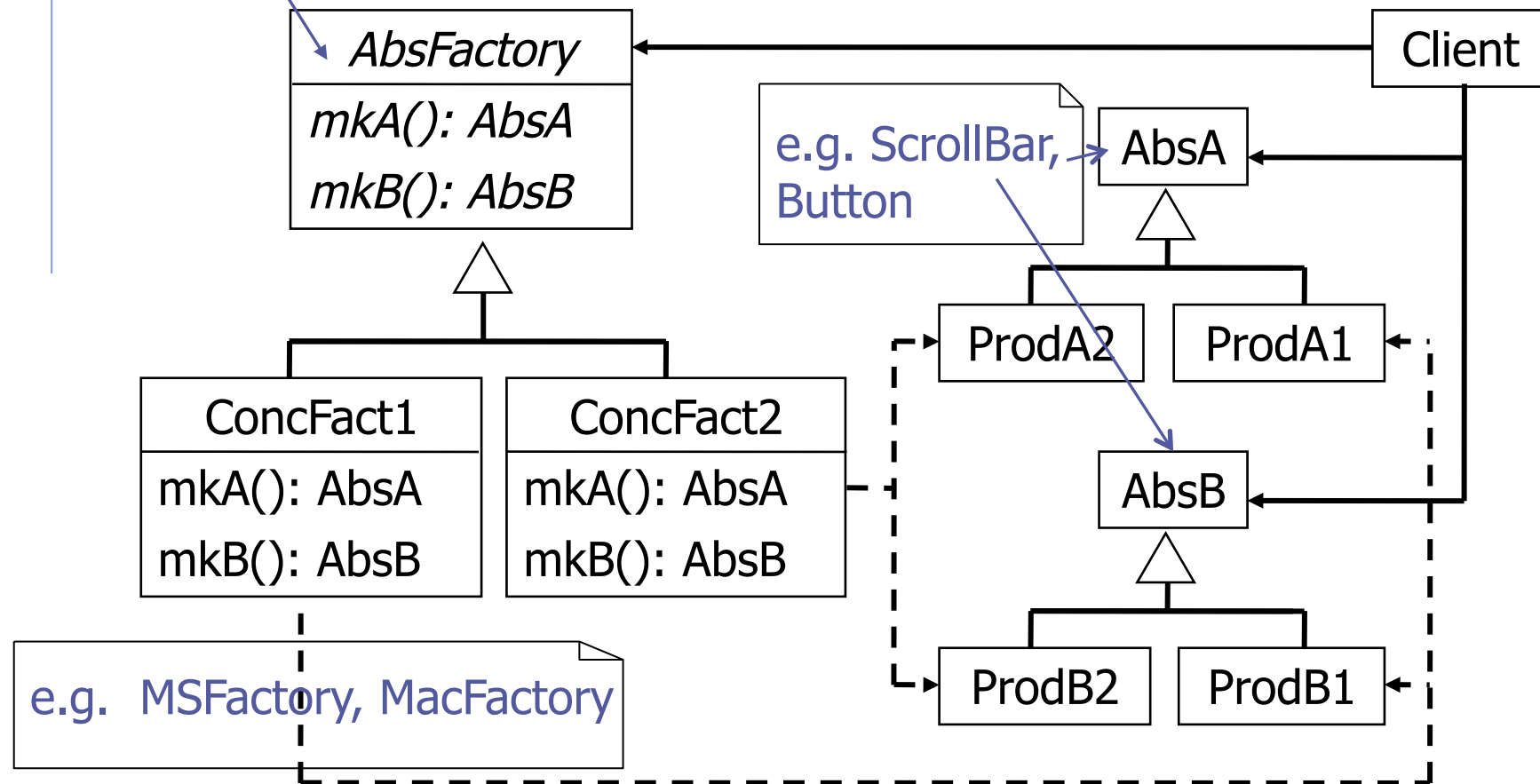
- ◆ Abstract Factory provides an additional level of abstraction compared to the Factory Method pattern.
- ◆ It provides an interface for creating families of related or dependent objects (e.g. buttons, menus etc.) without specifying their concrete classes.

- ◆ SW example: multiple look-and-feel user interfaces (win 9x, Macintosh and Motif)

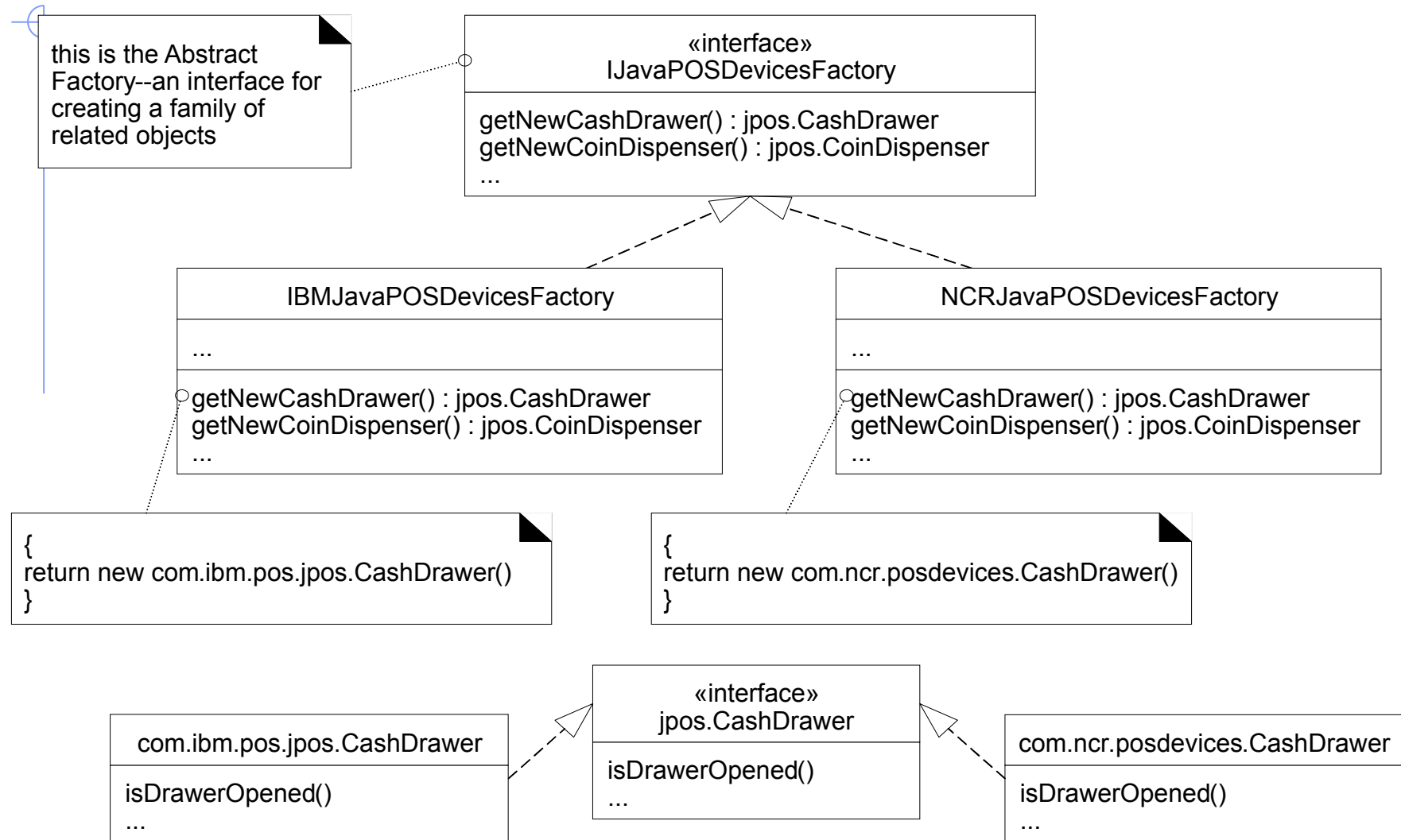


# Abstract Factory Structure

Declare an interface for operations that create abstract product objects, e.g. GUIFactory

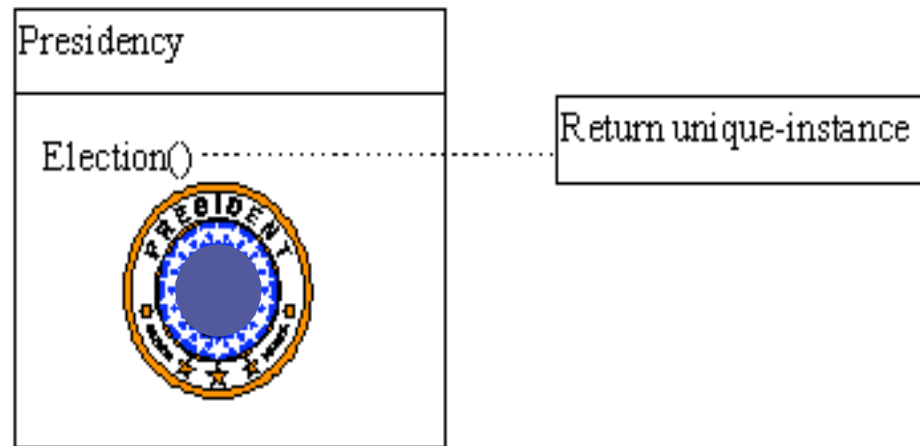


# Abstract Factory Example



# Singleton Pattern

- ◆ Ensures that a class has exactly one instance and provides a global point of access to it.



- ◆ In Java, using a private static variable and a class method are convenient implementation techniques.
- ◆ SW examples: one window manager, one print spooler, one file system.

# Singleton Implementation - 1

## ◆ Lazy initialization

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

public class Register {...
    accountingAdapter =
        ServicesFactory.getInstance().getAccountingAdapter();
...}
```

# Singleton Implementation - 2

## ◆ Eager initialization

```
public class ServicesFactory {  
    private static ServicesFactory instance =  
        new ServicesFactory();  
    public static ServicesFactory getInstance() {  
        return instance;  
    }  
    ...  
}
```

## ◆ Lazy initialization is usually preferred

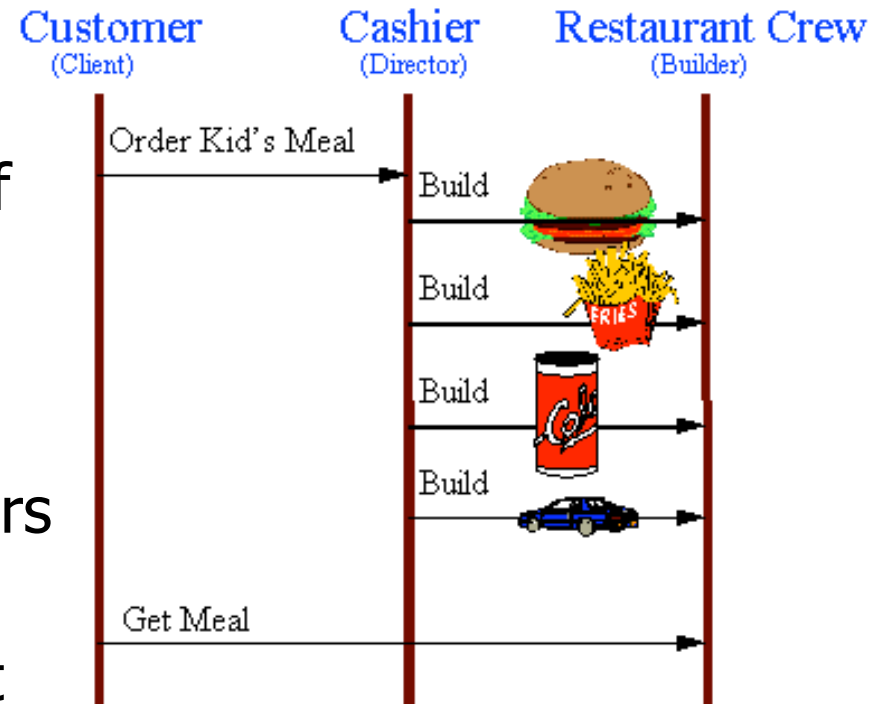
- Creation work is avoided if the instance is never actually accessed
- the *getInstance* lazy initialization sometimes contains complex and conditional creation logic

# Singleton Consequences

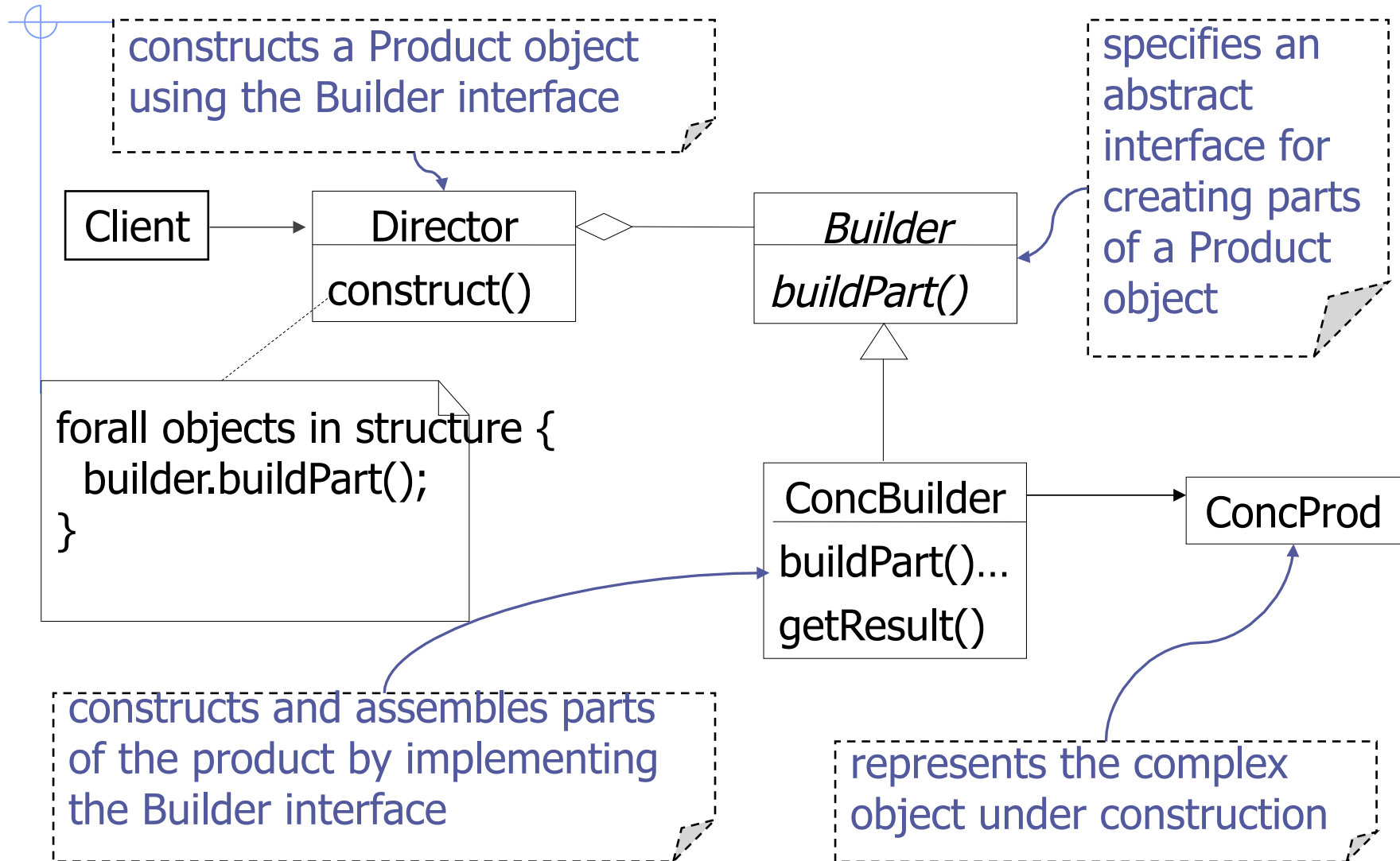
- ◆ In a multi-threaded environment, care is required to ensure that only **one** Singleton is created - use Java monitors (synchronized).
- ◆ The Singleton pattern avoids a main program creating an object and passing it to every other object that might need to access it.

# Builder Pattern

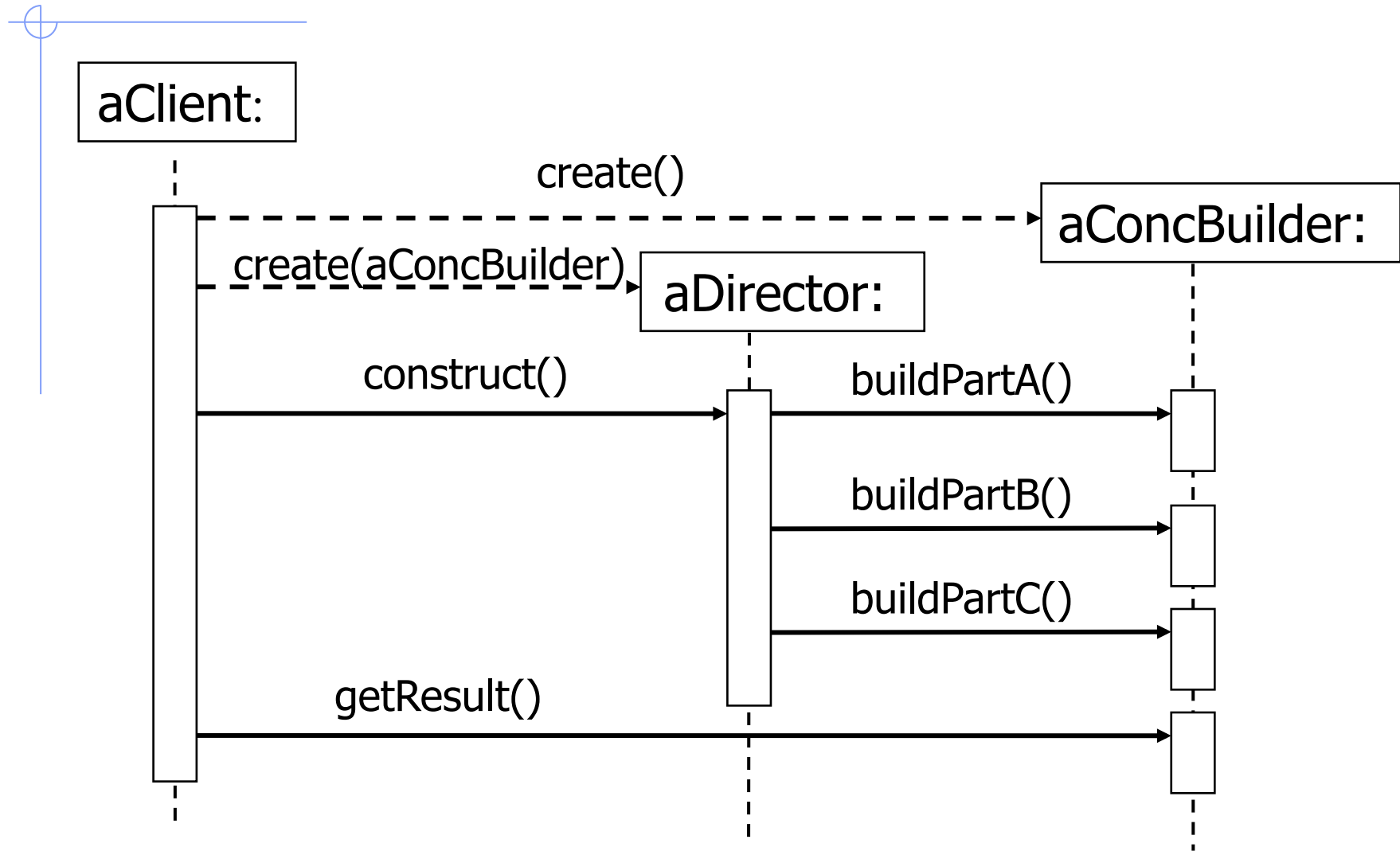
- ◆ Separates the construction of a complex object from its representation so the same construction process can create different representations.
- ◆ Useful to reduce size of complex class or to allow step-by-step construction of an object.
- ◆ The builder object can perform validity checks on the validity of the parameters supplied.
- ◆ SW example: a text convert



# Builder Pattern Structure



# Interactions



# Builder Pattern Consequences

- ◆ A Builder pattern lets you vary the internal representation of the product.
- ◆ Each specific builder is independent of any others.
- ◆ The builder pattern constructs the product step by step under the director's control, giving more control over the construction process.
- ◆ Example: a GUI interface that uses either a multi-choice list box or a set of checkboxes depending on the number of items from which to choose.

# Discussion of Creational Patterns

- ◆ Factory Method, Abstract Factory, Builder, and Singleton define a **factory object** that's responsible for **knowing** and **creating** the class of product objects, and make it a parameter of the system.
- ◆ Abstract Factory has the factory object producing objects of **several classes**.
- ◆ Builder has the factory object building a **complex product incrementally** using a correspondingly complex protocol.
- ◆ Abstract Factory and Builder can use **Singleton** in their implementation.

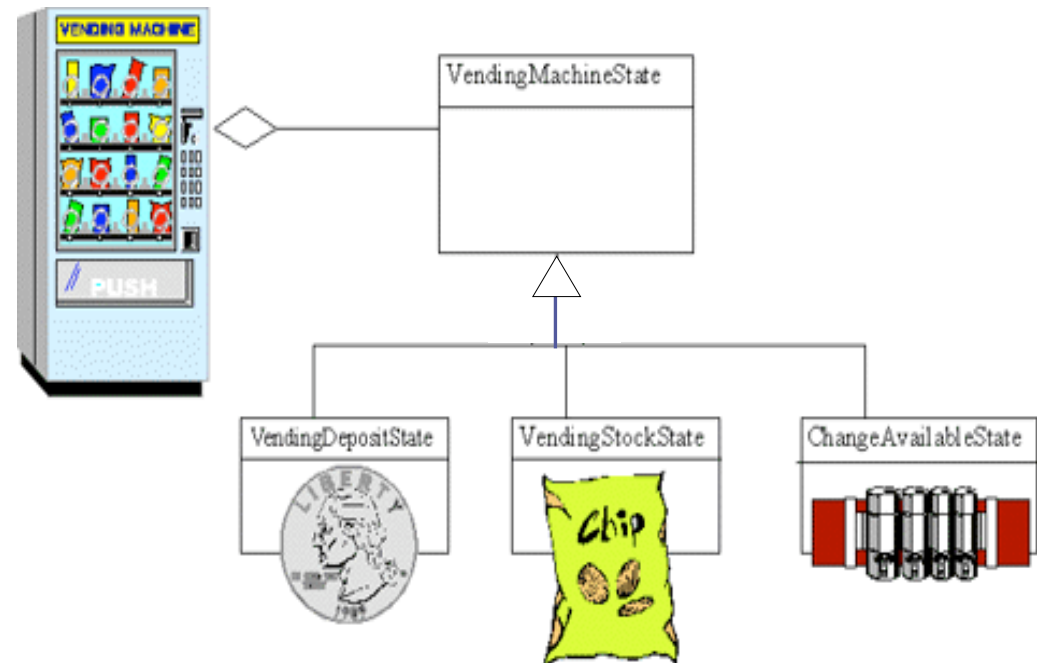
# GoF: Operation Patterns

- ◆ **State** - distribute an **operation** so each class represents a different state.
- ◆ **Strategy** - encapsulate an **operation**, making implementations interchangeable.
- ◆ **Template** - implement an **algorithm** in a method, deferring the definition of some steps so subclasses can redefine them.
- ◆ **Command** - encapsulate a **method call** in an object.

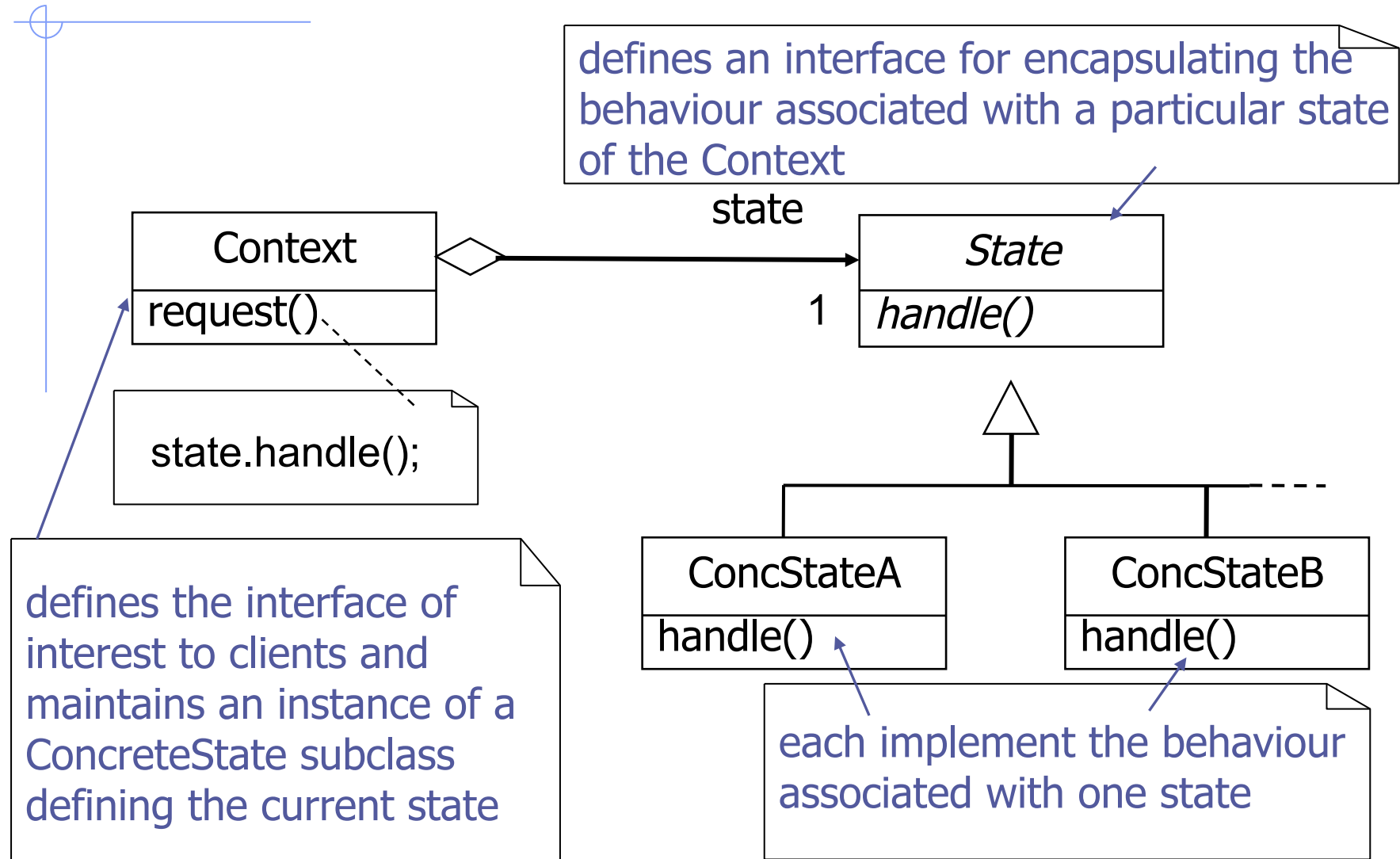
# State Pattern

- ◆ Allows an object to alter its behaviour when its internal state changes.
- ◆ Captures the different behaviours as different states with a class for each. All state-specific requests are delegated to the current state object.

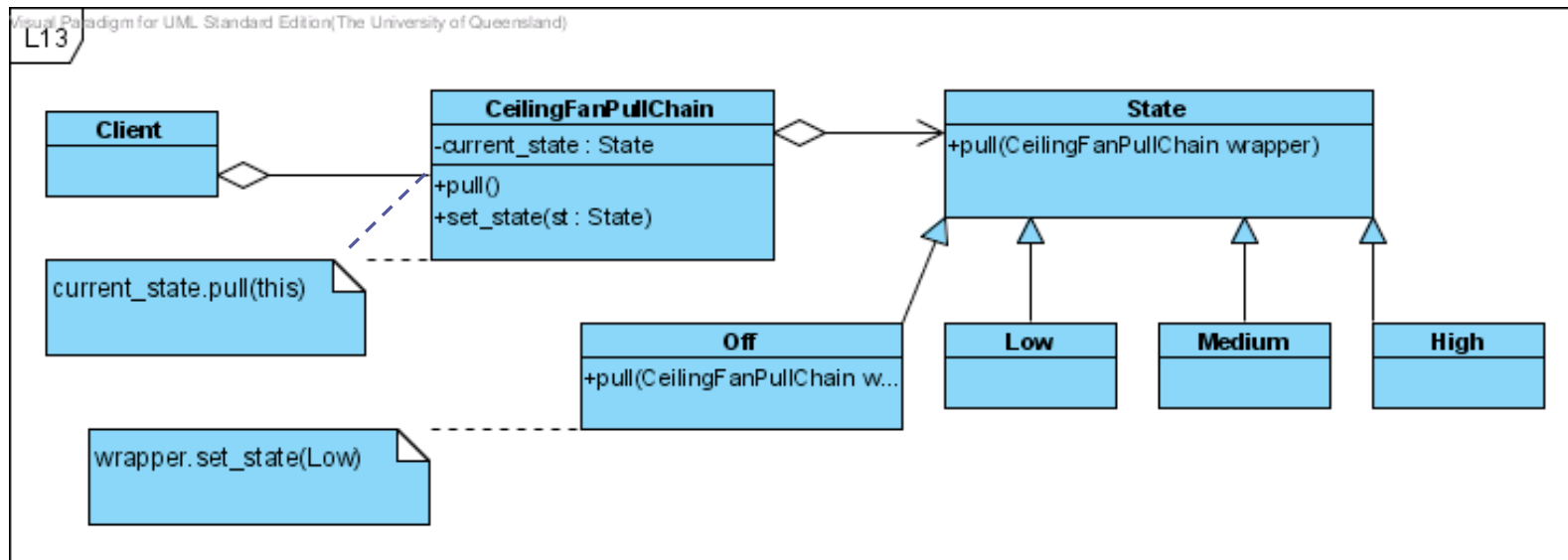
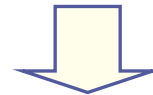
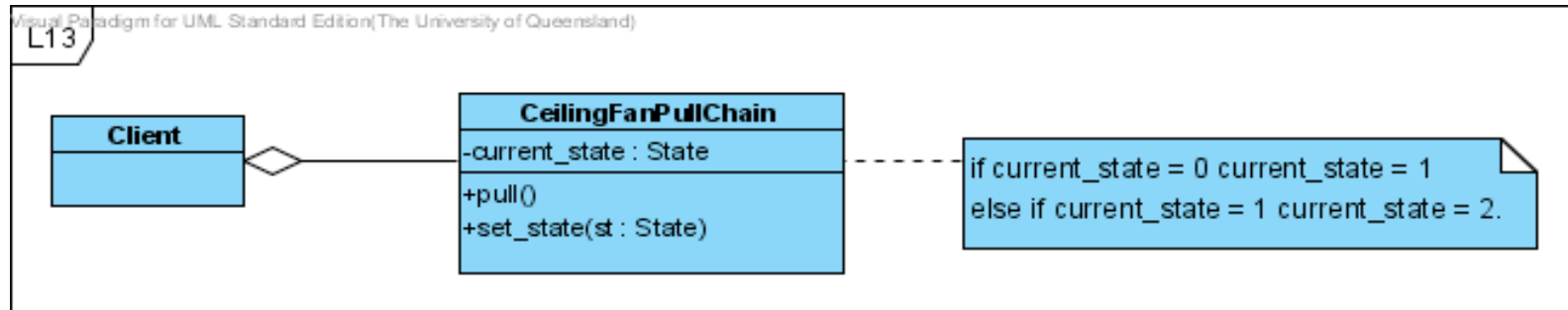
- ◆ SW example:  
a drawing editor where the behaviour depends on the currently selected tool (pen, brush, eraser, etc.)



# State Pattern Structure

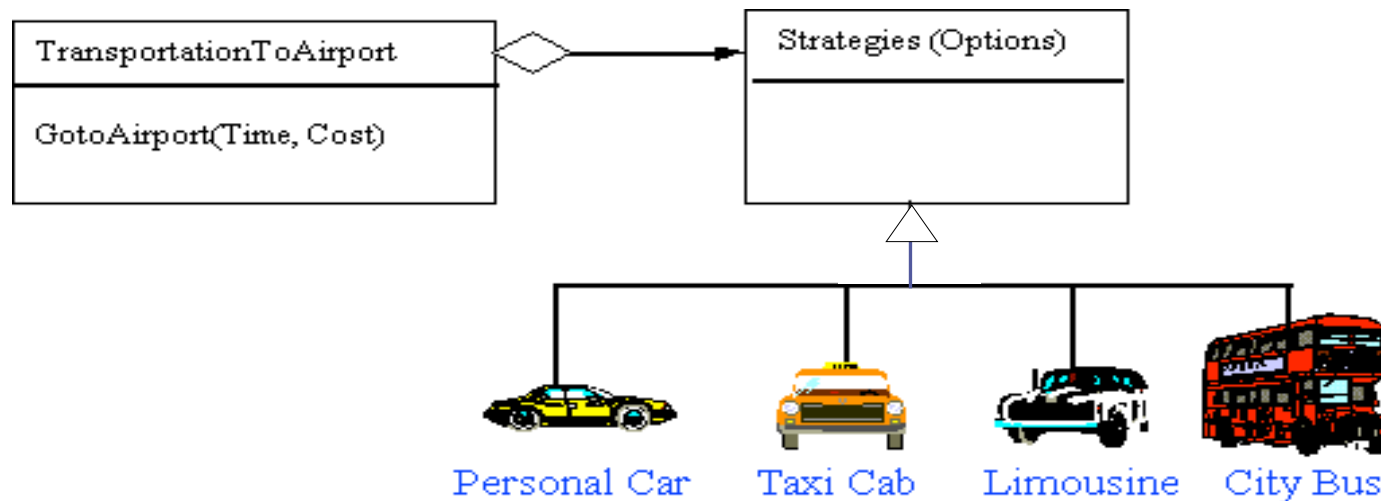


# State Example



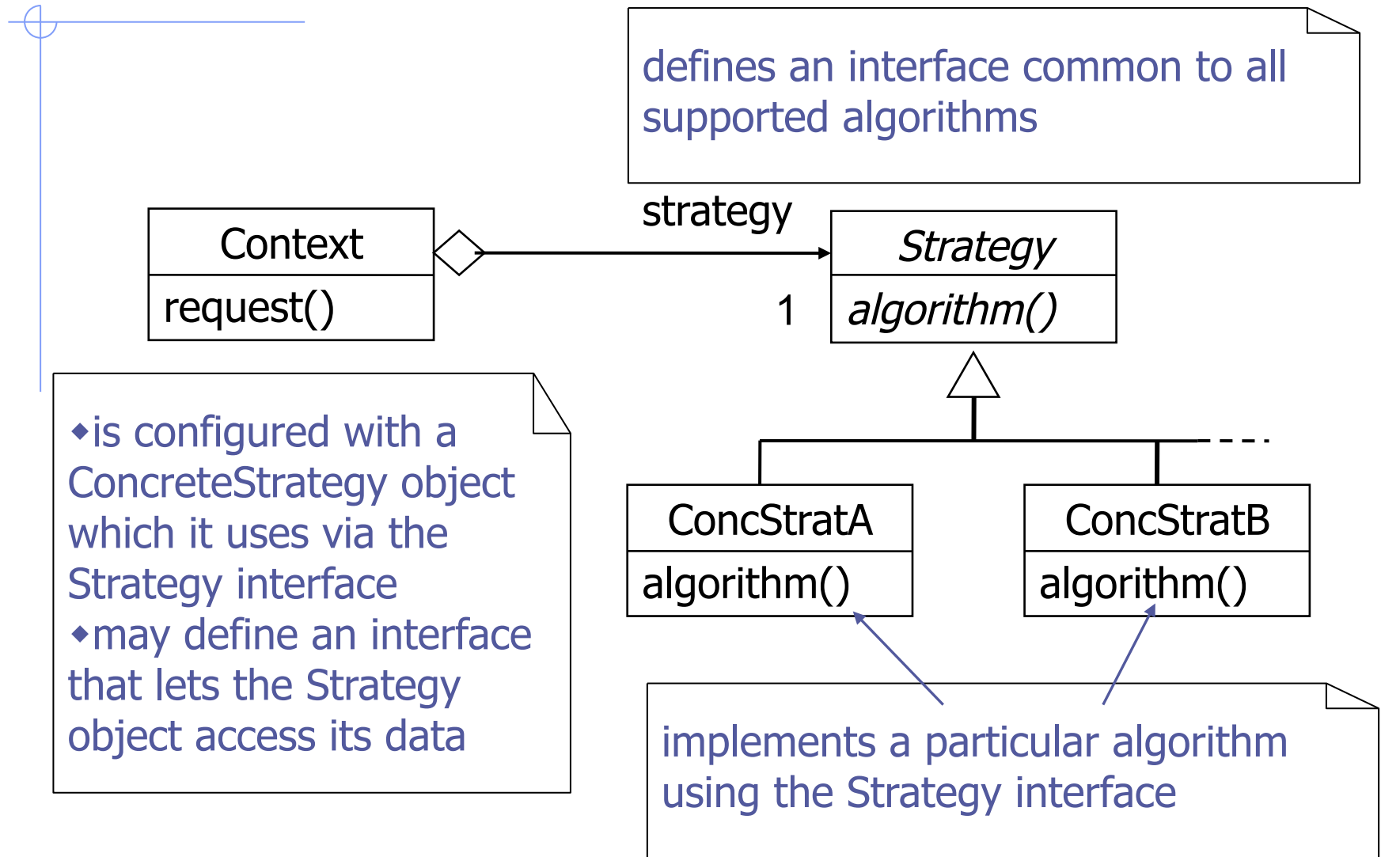
# Strategy Pattern

- ◆ Encapsulates a family of algorithms in a set of subclasses making them interchangeable.

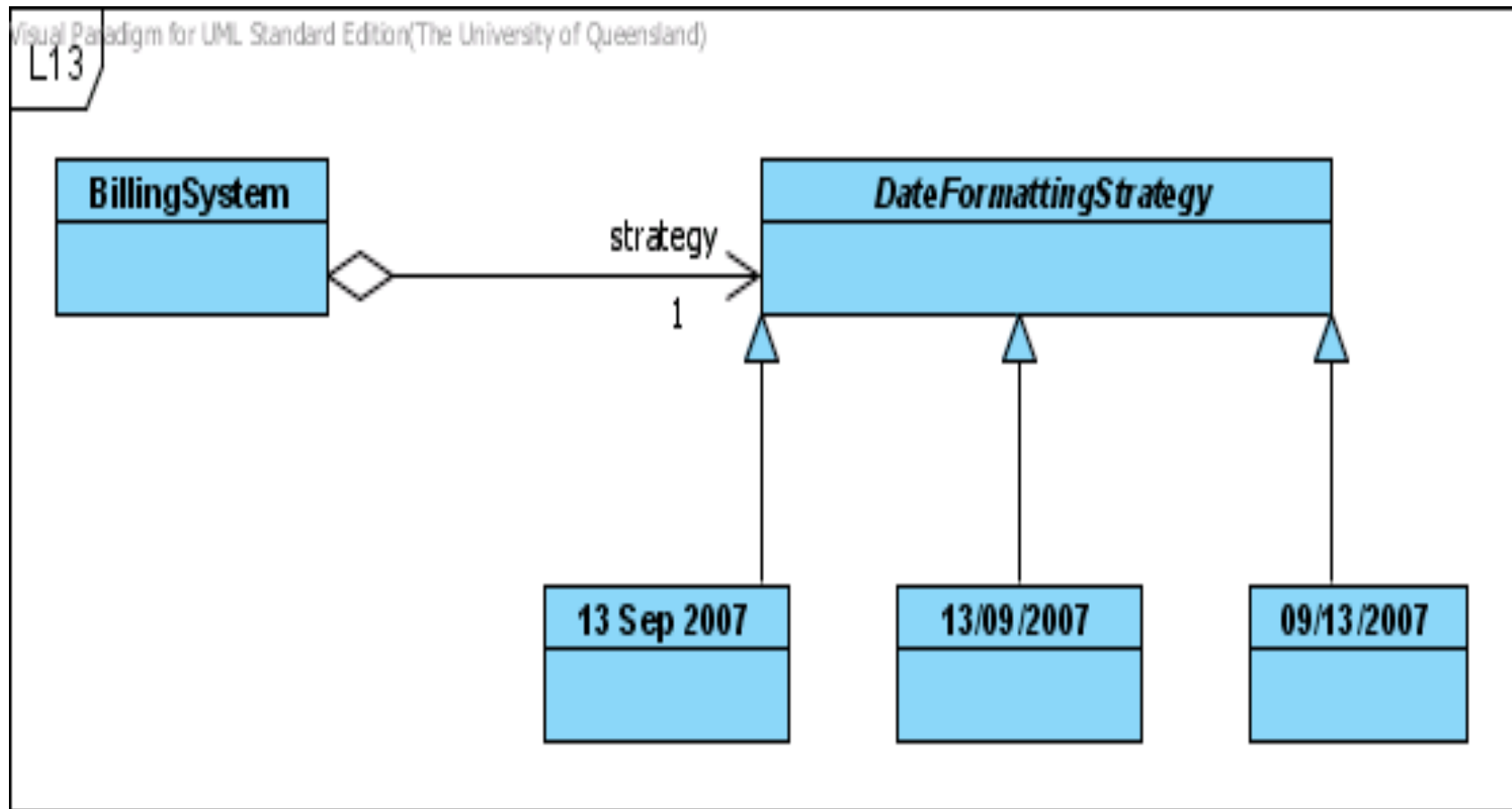


- ◆ SW examples: save files in different formats, compress files using different algorithms, present dates in different formats ...

# Strategy Pattern Structure



# Strategy Example



# Follow-up Reading

- ◆ Larman: Chapter 26 and 36
- ◆ Software Pattern References:
  - E. Gamma et al., Design patterns: elements of reusable object-oriented software, Addison-Wesley, 1995
  - F. Buschmann et al., A system of patterns, Wiley, 1996
  - Steven Metsker, Design Patterns Java Workbook, Addison-Wesley, 2002.
- ◆ Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p52-57