

Tecniche di Progettazione: Design Patterns

GoF: Builder, Chain Of Responsibility, Flyweight

1

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Builder

2

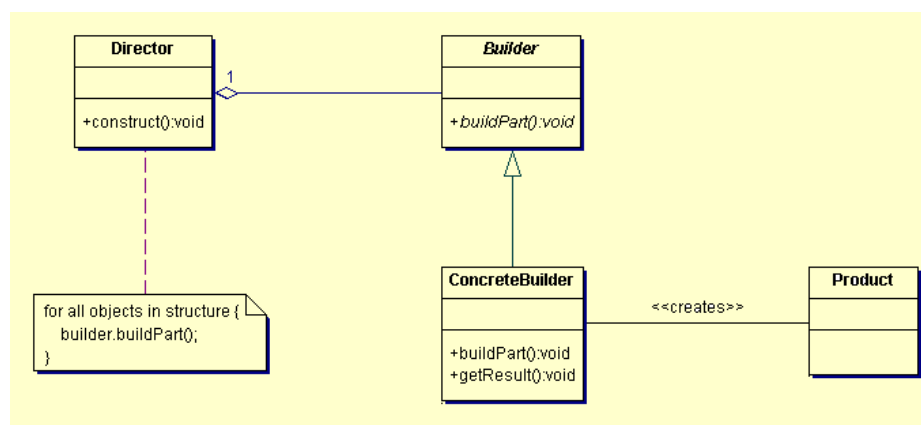
Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Builder: intent

- ▶ Separate the construction of a complex object from its representation so that the same construction process can create different representations

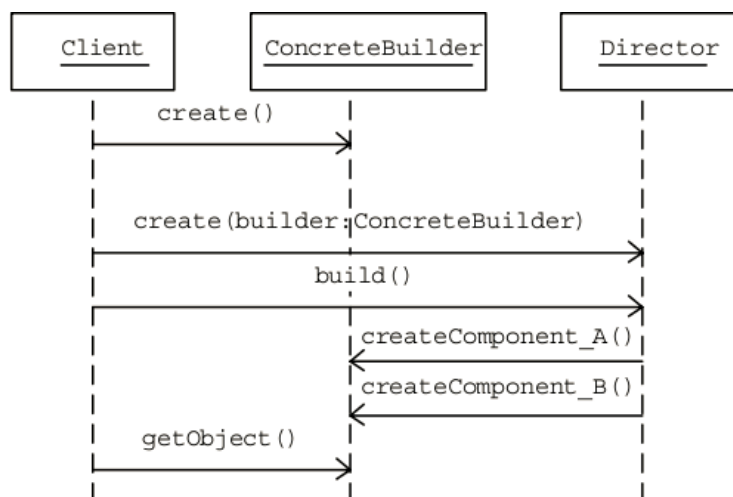


Builder: structure



Builder: participants

- ▶ **Builder**
 - ▶ Specifies an interface for creating/assembling parts of a product.
- ▶ **ConcreteBuilder**
 - ▶ Implements the Builder interface
- ▶ **Director**
 - ▶ Constructs an object of an unknown type using the Builder interface
- ▶ **Product**
 - ▶ Complete object returned by invoking getResult() on the ConcreteBuilder



Builder: consequences

- ▶ **Isolates code for construction and representation**
 - ▶ Construction logic is encapsulated within the director
 - ▶ Product structure is encapsulated within the concrete builder
 - ▶ => Lets you vary a product's internal representation
- ▶ **Supports fine control over the construction process**
 - ▶ Breaks the process into small steps



Builder: applicability

- ▶ **Use the Builder pattern when**
 - ▶ The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
 - ▶ The construction process must allow different representations for the object that's constructed

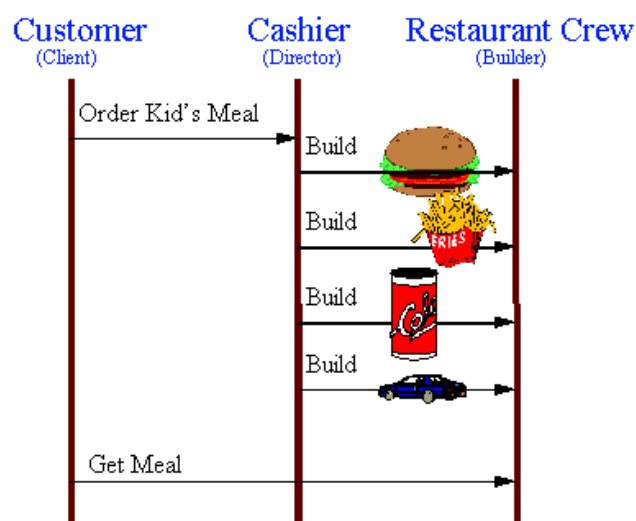


Builder: implementation

- ▶ The Builder interface
 - ▶ Must be general enough to allow construction of many products
- ▶ Abstract base class for all products?
 - ▶ Usually not feasible (products are highly different)
- ▶ Default implementation for methods of Builder?
 - ▶ "Yes": May decrease amount of code in ConcreteBuilders
 - ▶ "No:" May introduce silent bugs



Ex.



▶ 10

Design patterns, Laura Semini,
Università di Pisa, Dipartimento di

Ex: MazeBuilder (the builders)

```
public interface MazeBuilder {
    public void buildRoom(int r);
    public void buildDoor(int d);
    public Maze getMaze();
}
```

This interface can create three things: (1) the maze, (2) rooms with a particular room number, and (3) doors between numbered rooms. The GetMaze operation returns the maze to the client. Subclasses of MazeBuilder will override this operation to return the maze that they build.

All the maze-building operations of MazeBuilder do nothing by default. They're not declared pure virtual to let derived classes override only those methods in which they're interested.

```
public class StandardMazeGame implements MazeBuilder { }
public class ComplexMazeGame implements MazeBuilder { }
```

▶ 11

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Ex: MazeBuilder (director)

```
public class MazeGame {
    public Maze createComplexMaze(MazeBuilder builder) {
        builder.buildDoor(12);
        builder.buildDoor(22);
        builder.buildRoom(133);
        return builder.getMaze(); }
    public Maze createMaze(MazeBuilder builder) {
        builder.buildDoor(1);
        builder.buildDoor(3);
        builder.buildRoom(4);
        return builder.getMaze(); } }
```

▶ 12

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Ex: MazeBuilder (client)

```
public class Client {  
    public static void main(String[] args) {  
        Maze maze;  
        MazeGame game = new MazeGame();  
        MazeBuilder builder = new StandardMazeGame();  
        game.createMaze(builder);  
        maze = builder.GetMaze();  
    }  
}
```

▶ 13

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

-
- ▶ Compare this version of CreateMaze with the original. Notice how the builder hides the internal representation of the Maze that is, the classes that define rooms, doors, and walls and how these parts are assembled to complete the final maze. Someone might guess that there are classes for representing rooms and doors, but there is no hint of one for walls. This makes it easier to change the way a maze is represented, since none of the clients of MazeBuilder has to be changed.

▶ 14

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

What is the difference between Builder Design pattern and Factory Design pattern?

- ▶ The Factory pattern can almost be seen as a simplified version of the Builder pattern.
- ▶ In the **Factory** pattern, the factory is in charge of creating various subtypes of an object depending on the needs.
- ▶ The user of a factory method doesn't need to know the exact subtype of that object. An example of a factory method createCar might return a Ford or a Honda typed object.
- ▶ In the **Builder** pattern, different subtypes are also created by a builder method, but the composition of the objects might differ within the same subclass.
- ▶ To continue the car example you might have a createCar builder method which creates a Honda-typed object with a 4 cylinder engine, or a Honda-typed object with 6 cylinders. The builder pattern allows for this finer granularity.

▶ 15

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

What is the difference between Builder Design pattern and Factory Design pattern? Cont'd

- ▶ From Wikipedia:
- ▶ Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- ▶ Builder often builds a Composite.
- ▶ Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- ▶ Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.

▶ 16

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Chain Of responsibility

Chain Of Responsibility

▶ Intent

- ▶ Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it

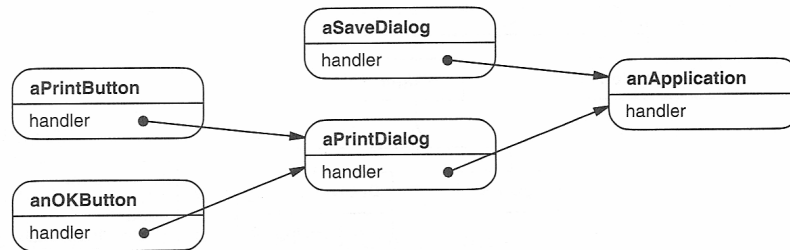
▶ Example

- ▶ Help information on an interface
- ▶ Organize from most specific to general (da chi sa fare meno cose a chi ne sa fare di più)
- ▶ Pattern decouples object that initiates request from the object the ultimately provides the help (non è il cliente che gira tra gli sportelli, ma la sua richiesta)



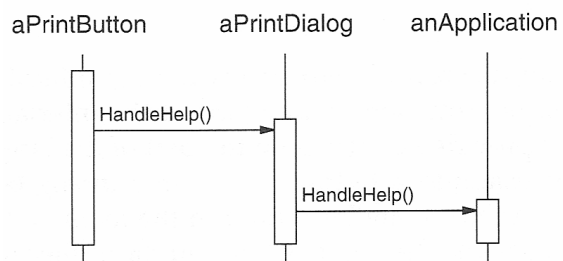
Flow

- ▶ Request is passed a long chain of objects until one handles it



Flow

- ▶ First Object receives the request and either handles it or forwards it to the next candidate
- ▶ Example
 - ▶ User clicks help on a PrintButton within a PrintDialog

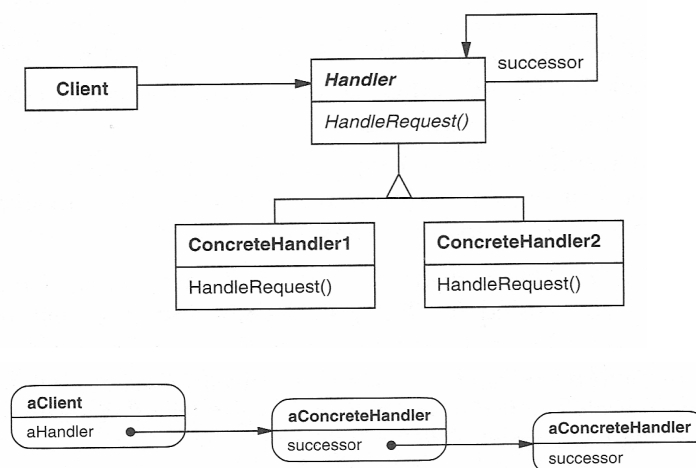


Applicability

- ▶ Use Chain of Responsibility when
 - ▶ More than one object may handle a request and the handler isn't known a priori.
 - ▶ You want to issue a request to one of several objects without specifying the receiver explicitly
 - ▶ The Set of objects that can handle a request should be specified dynamically



Structure



Participants

- ▶ **Handler**
 - ▶ Defines an interface for handling request
 - ▶ Optional implements the successor link
- ▶ **ConcreteHandler**
 - ▶ Handles requests it is responsible for
 - ▶ Can access its successor
 - ▶ Forwards requests it does not handle
- ▶ **Client**
 - ▶ Initiates the request to a (usually the first) ConcreteHandler object on the chain



Consequences

- ▶ **Reduced Coupling**
 - ▶ Objects are free from knowing what object handles the request
- ▶ **Added Flexibility in assigning responsibilities to objects**
 - ▶ Can change chain at runtime
 - ▶ Can subclass for special handlers
- ▶ **Receipt is guaranteed**
 - ▶ Request could fall off the chain
 - ▶ Request could be dropped with bad chain



Implementation

- ▶ **Implementing the successor chain**
 - ▶ Define new links
 - ▶ Can be handled at the base class level
 - ▶ Use existing links
 - ▶ In case like Composite, can use parent link
 - ▶ Sometimes redundant links are needed, if relationship structure differs



Implementation

- ▶ **Representing Requests**
 - ▶ Hard coded operations
 - ▶ Limited in handling requests
 - ▶ Argument with handler
 - ▶ Requires conditionals
 - ▶ Requires packing/unpacking arguments
 - ▶ Separate Request Objects
 - ▶ Must be able to determine type in handler
 - ▶ Can subclass handlers



Related Patterns

- ▶ **Composite**
 - ▶ Used with Chain of Responsibility so parent can act as a successor
-



Why would I ever use a Chain of Responsibility over a Decorator?

- ▶ **The fact that you can break the chain at any point differentiates the Chain of Responsibility pattern from the Decorator pattern.** Decorators can be thought of as executing all at once without any interaction with the other decorators. Links in a chain can be thought of as executing one at a time, because they each depend on the previous link.
- ▶ **Use the Chain of Responsibility pattern when you can conceptualize your program as a chain made up of links, where each link can either handle a request or pass it up the chain.**

Experience with CoR

- ▶ Ok, just recently I've got some experience with this pattern in real enterprise development.
- ▶ In our system we had different levels of storages (in order of precedence): base, product-specific and user-specific. Each storage held a set of XML files which described in metalanguage the elements to be shown on a webpage. Each XML file described the whole page to be shown.
- ▶ When user tried to navigate to a webpage our system performed a lookup in storages of necessary XML file with regard to page ID.
- ▶ Lookup ended when necessary file found, no merging algorithm was expected (entirely no merging between layers).
- ▶ So for this particular situation we've introduced an interface with 1 method (in pseudocode):
- ▶ ViewMetadata GetViewMetadata(string viewId); We've simply constructed a chain of storages - implementors of this interface. The HttpHandler (for Asp.Net usecase) which was responsible for building the actual HTML representation from XML view metadata had **only one single entry point** to lookup mechanism: an instance of mentioned interface implementer.

▶ 29

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Experience with CoR cont'd

- ▶ All seemed to be OK (for several months actually). **But**
- ▶ New requirement came. This requirement was about "promotion": XML file under certain circumstances ("promotion workflow") must be transferred from user storage to product storage.
- ▶ Actually due to our too abstract implementation of lookup mechanism we couldn't handle this requirement - there was only one abstract entry point for handler to the whole chain of storages.
- ▶ So as a result we rejected CoR pattern for this requirement. Now actual handler has explicit references to all 3 storages and when "Promote view with view Id = X" request comes it simply obtains the metadata directly from user storage and saves the metadata to product storage.
- ▶ So, as a bottom line, I could say that CoR pattern is not a brilliant solution when you have some requirements about explicit interactions with chain elements. Too abstract interface of CoR "HandleRequest()" gives you only an opportunity to interact with an entry point of the chain.
- ▶ BTW it was an opportunity to save CoR-based implementation, but the question actually was really about efforts and maintainability :)

▶ 30

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

What's the difference between “Chain of responsibility” and “Strategy” patterns?

- ▶ They're very different.
- ▶ **Strategy** is about having a generic interface which you can use to provide different implementations of an algorithm, or several algorithms or pieces of logic which have some common dependencies.
- ▶ For instance, your `CollectionSorter` could support a `SortingStrategy` (merge sort, quick sort, bubble sort). They all have the same interface and purpose, but can do different things.
- ▶ In some cases you may decide to determine strategy *inside*. Maybe the sorter has some heuristics based on collection size etc. Most of the time it indeed is *injected from outside*. This is when the pattern really shines: It provides users the ability to override (or provide) behavior.
- ▶ This pattern is base of the now-omnipresent **Inversion of Control**. Study that next once you're done with the classic patterns.

▶ 31

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

What's the difference between “Chain of responsibility” and “Strategy” patterns?

- ▶ **Chain of responsibility** is about having a chain of objects which usually go from more detailed to more generic. Each of the pieces in chain can provide the answer, but they have different levels of detail.
- ▶ Popular GOF example is a context help system. When you click on a component in your desktop app, which help to display? First item in chain could look for help for the very component you clicked. Next in chain could try and display help for the whole containing dialog. Next for the application module... and so on.

▶ 32

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Flyweight

33

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Flyweight Pattern

▶ Intent

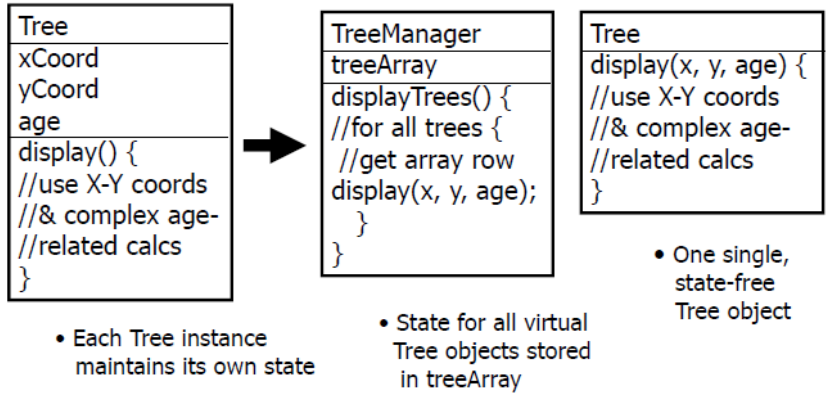
- ▶ Use sharing to support large numbers of fine-grained objects efficiently

▶ Motivation

- ▶ Can be used when an application could benefit from using objects throughout their design, but a naïve implementation would be prohibitively expensive
 - ▶ Objects for each character in a document editor
 - Cost is too great!
 - ▶ Can use flyweight to share characters

▶

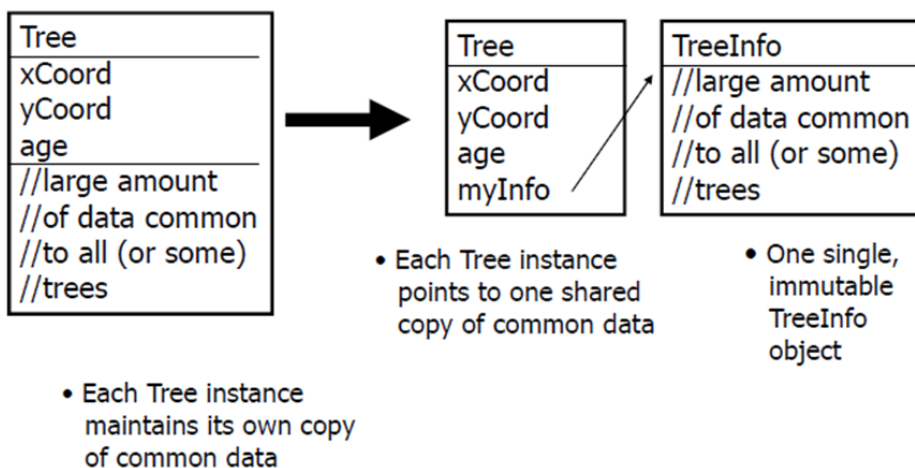
Head first example



▶ 35

Design patterns, Laura Semini, Università di Pisa, Dipartimento di

Another ex.



▶ 36

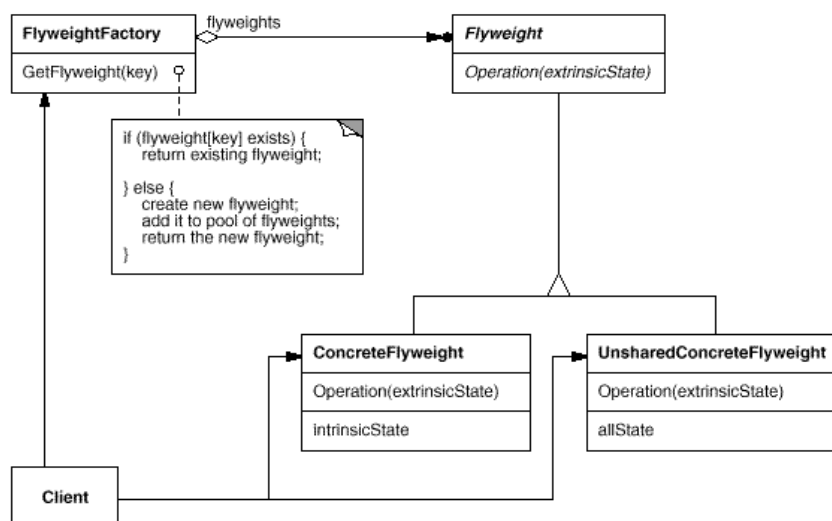
Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Intrinsic vs. Extrinsic

- ▶ most objects would share a set of stateless information, this could be extracted from the main objects to be held in flyweight objects
- ▶ Intrinsic
 - ▶ The intrinsic data is held in the properties of the flyweight objects that are shared. This information is stateless and generally remains unchanged, as any changes would be effectively replicated amongst all of the objects that reference the flyweight
- ▶ Extrinsic
 - ▶ Extrinsic data can be stateful as it is held outside of a flyweight object. It can be passed to methods of a flyweight when needed but should never be stored within a shared flyweight object.



Flyweight: Structure



Flyweight: Participants

- ▶ **Flyweight**
 - ▶ Declares an interface through which flyweights can receive and act on extrinsic state
- ▶ **ConcreteFlyweight**
 - ▶ Implements the Flyweight interface and adds storage for intrinsic state, if any
 - ▶ Must be shareable
- ▶ **UnsharedConcreteFlyweight**
 - ▶ Although the flyweight design pattern enables sharing of information, it is possible to create instances of concrete flyweight classes that are not shared. In these cases, the objects may be stateful.



Flyweight: Participants

- ▶ **FlyweightFactory**
 - ▶ Creates and manages flyweight objects
 - ▶ Ensures that flyweights are shared properly
- ▶ **Client**
 - ▶ Maintains reference to flyweights
 - ▶ Computes or stores the extrinsic state of flyweights



Flyweight: Applicability

- ▶ Use the Flyweight pattern when ALL of the following are true
 - ▶ An application uses a large number of objects
 - ▶ Storage costs are high because of the sheer quantity of objects
 - ▶ Most object state can be made extrinsic
 - ▶ Many Groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
 - ▶ The application doesn't depend on object identity
-



Flyweight: Consequences

- ▶ May introduce run-time costs associated with transferring, finding, and/or computing extrinsic state
 - ▶ Costs are offset by space savings
 - ▶ Storage savings are a function of the following factors:
 - ▶ The reduction in the total number of instances that comes from sharing
 - ▶ The amount of intrinsic state per object
 - ▶ Whether extrinsic state is computed or stored
 - ▶ Ideal situation
 - ▶ High number of shared flyweights
 - ▶ Objects use substantial quantities of both intrinsic and extrinsic state
 - ▶ Extrinsic state is computed
-



Implementation

- ▶ **Removing extrinsic state**
 - ▶ Success of pattern depends on ability to remove extrinsic state from shared objects
 - ▶ No help if there are many different kinds of extrinsic state
 - ▶ Ideally, state is computed separately
- ▶ **Managing shared objects**
 - ▶ Objects are shared so clients should not instantiate
 - ▶ FlyweightFactory is used to create and share objects
 - ▶ Garbage collection may not be necessary



Flyweight: Related Patterns

- ▶ **Composite**
 - ▶ Often combined with flyweight
 - ▶ Provides a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes
- ▶ **State and Strategy**
 - ▶ Best implemented as flyweights



-
- ▶ Flyweight is a boxing category, for light weight people.
 - ▶ Flyweight pattern is for "light weight" objects (though many of them).