Tecniche di Progettazione: Design Patterns

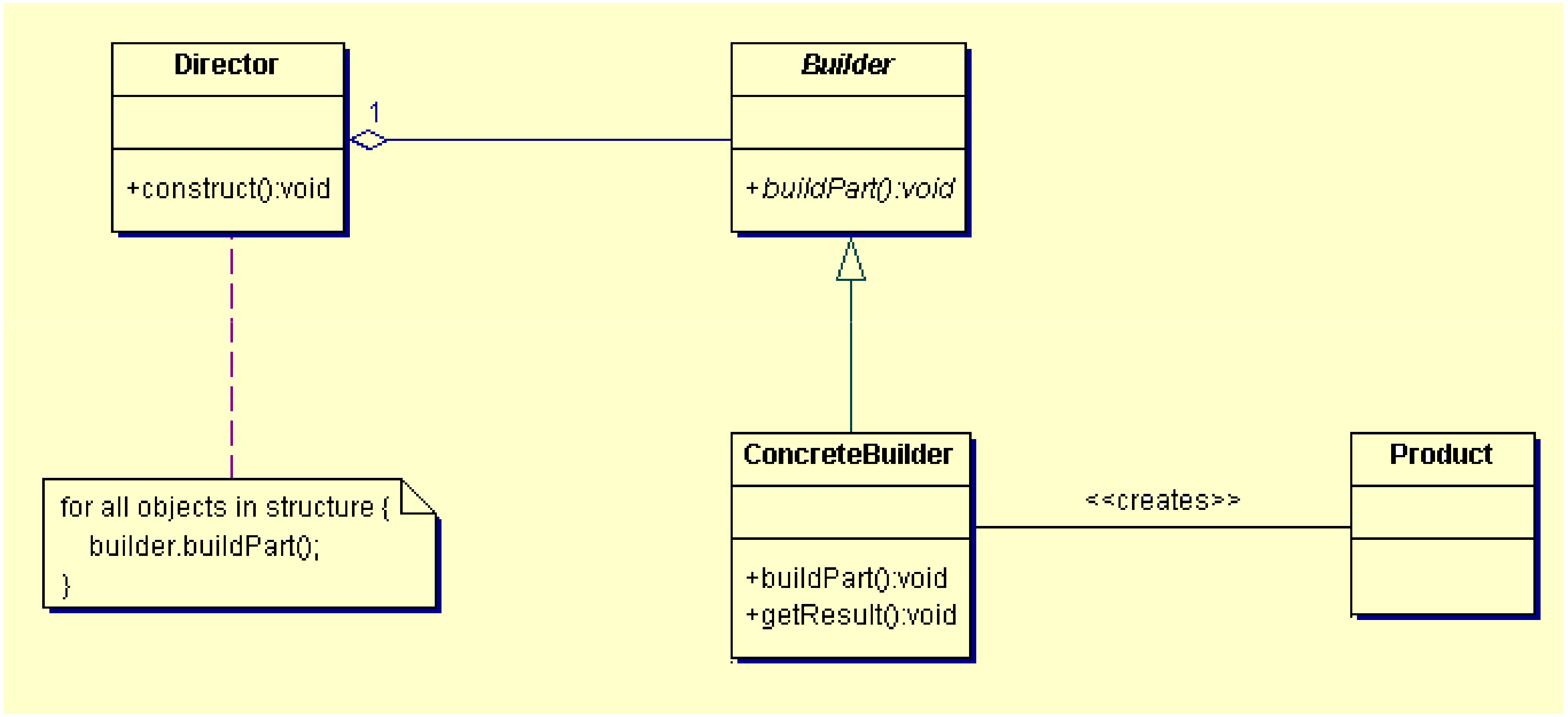
GoF: Builder, Chain Of Responsibility, Flyweight

Builder

Builder: intent

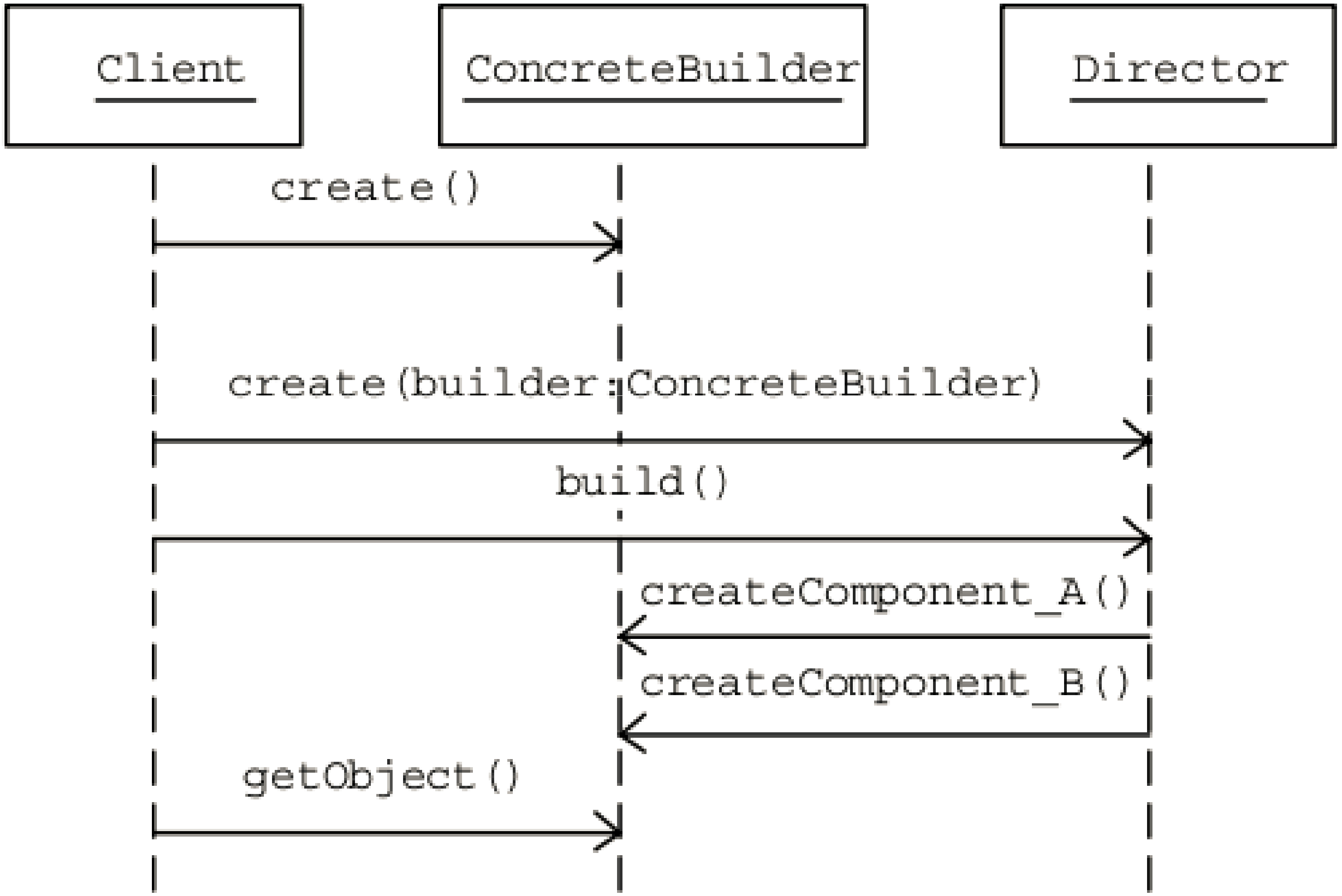
- Separate the construction of a complex object from its representation so that the same construction process can create different representations

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


Client
ConcreteBuilder
Director
create()
create(builder:ConcreteBuilder)
build()
createComponent_A()
createComponent_B()
getObject()

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

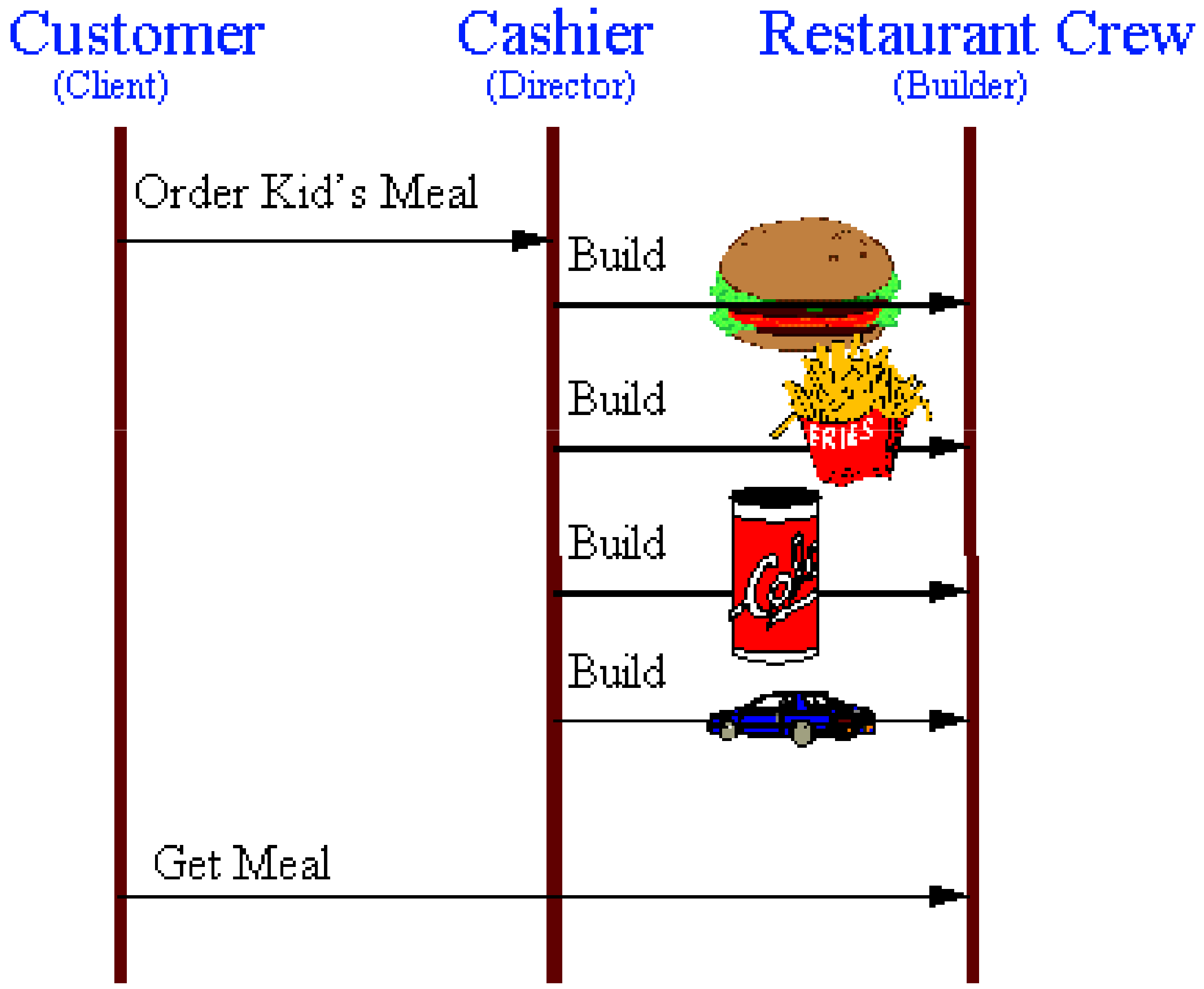
Builder: applicability

- 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. da riscrivere a fine lezione



Ex: MazeBuilder (the builders)

```
public class MazeBuilder {
    public void buildMaze(){ };
    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 abstarct to let derived classes override only those methods in which they're interested.

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

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(); } }
```

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();
   }
}
```

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.

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.

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.

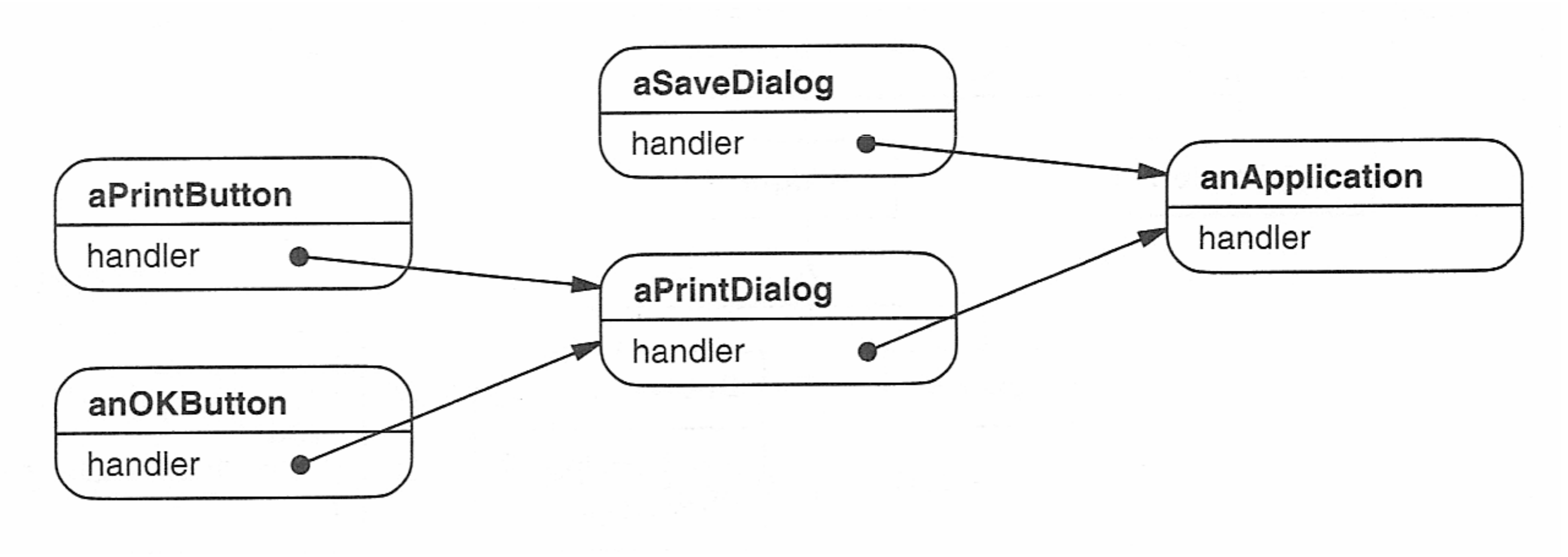
Chain Of responsibility

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 ruchiesta)

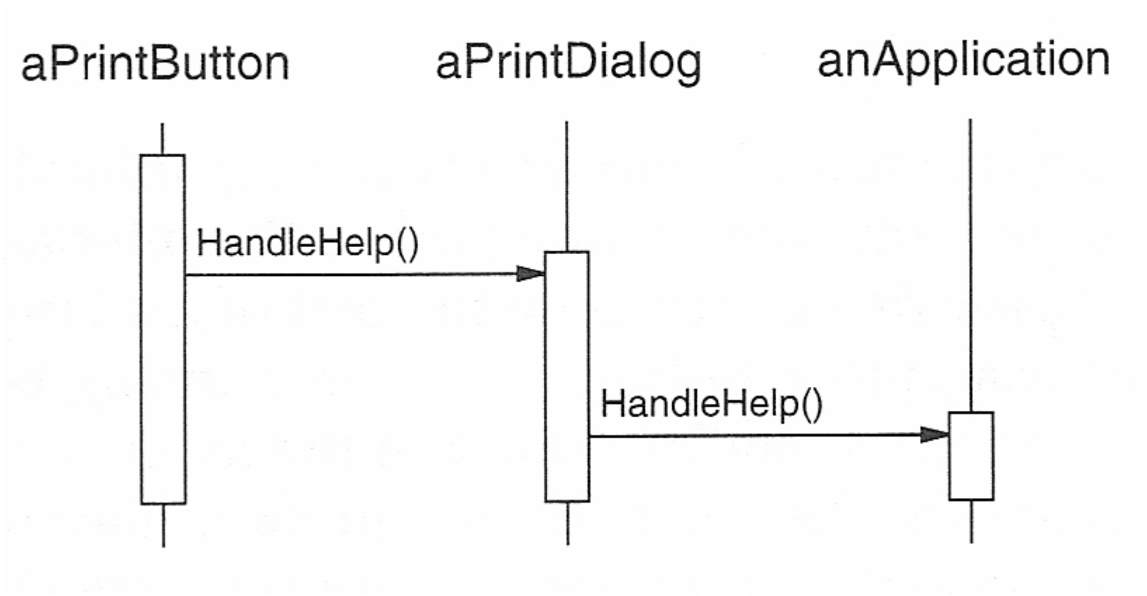
Flow

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



Flow

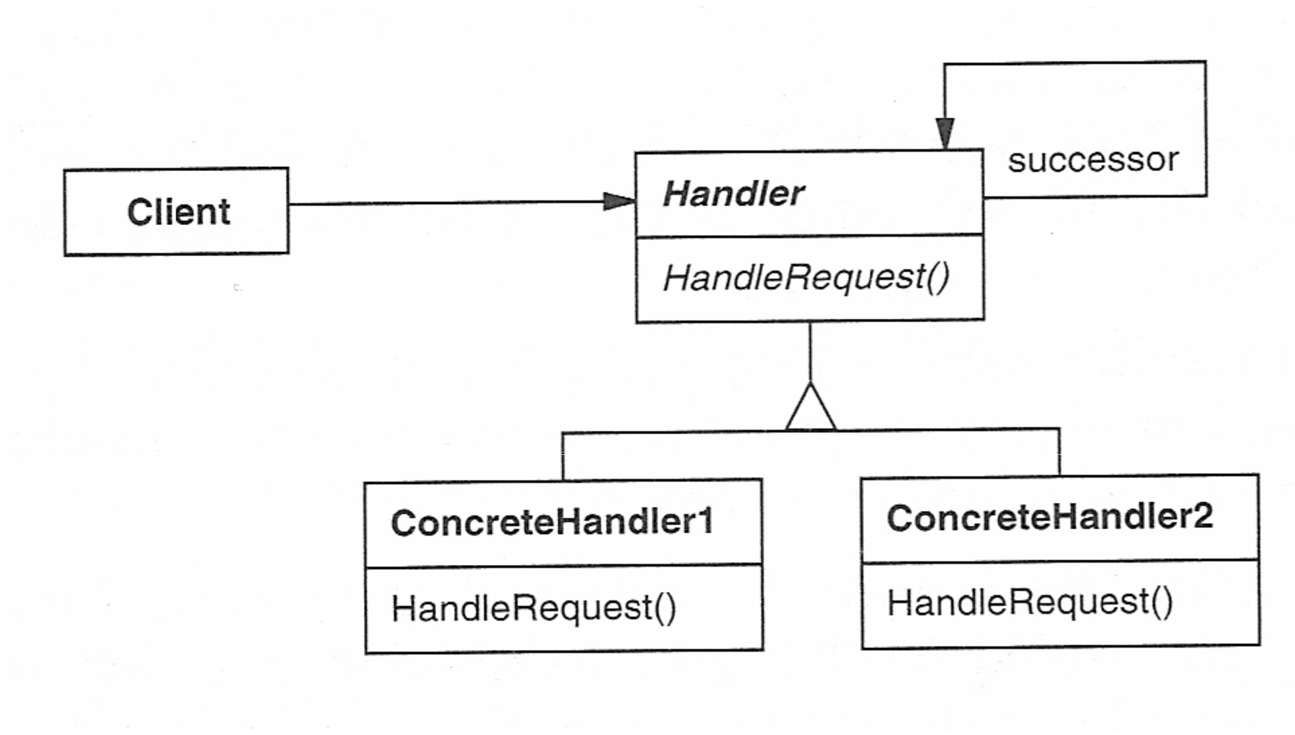
- First Object receives the request and either handles it or forwards it to the next candidate
- Example
  - User clicks help on a PrintButton within a PrintDialog

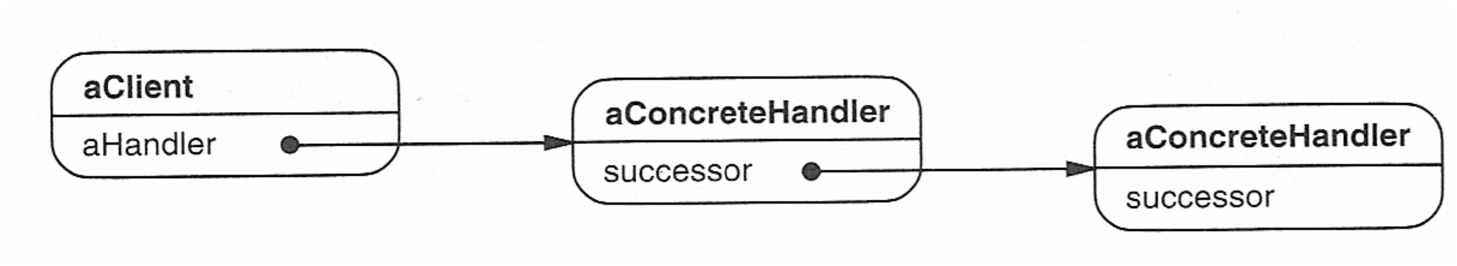


Applicability

- 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 than can handle a request should be specified dynamically

Structure


Client
Handler
HandleRequest()
successor
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()


aClient
aHandler
aConcreteHandler
successor
aConcreteHandler
successor

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
- Decorator
  - See next slide

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

- **CoR: you can break the chain at any point**
  - This is not true of **Decorator**.
  - Decorators can be thought of as executing all at once without any interaction with the other decorators.

- 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.

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

- 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, a 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. (dependency injection and Inversion of Control.)

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.

Chain-of-responsibility vs. lists of handlers

- Problem: refactor a huge(1000 lines) method full of "if" statements.
- Solution 1: chain-of-responsibility pattern: base "Handler" class. Then, "Handler1", "Handler2", etc.
- Solution 2: base "Handler" class as well, with "Handler1", "Handler2", just like the previous method mentioned.
  However, there would be no "getSuccessor" method. Instead, a Collection class with a list of handlers(a Vector, an ArrayList, …).
  The handleRequest function would still exist, but it wouldn't propagate the call to the next handlers. It would just process the request or return null.
  To handle a request, one would use
-
```
for(Handler handle : handlers){
    result = handle.handleRequest(request);
    if(result!=null) return result;
}
throw new CouldNotParseRequestException();
```

Chain-of-responsibility vs. lists of handlers (cont'd)

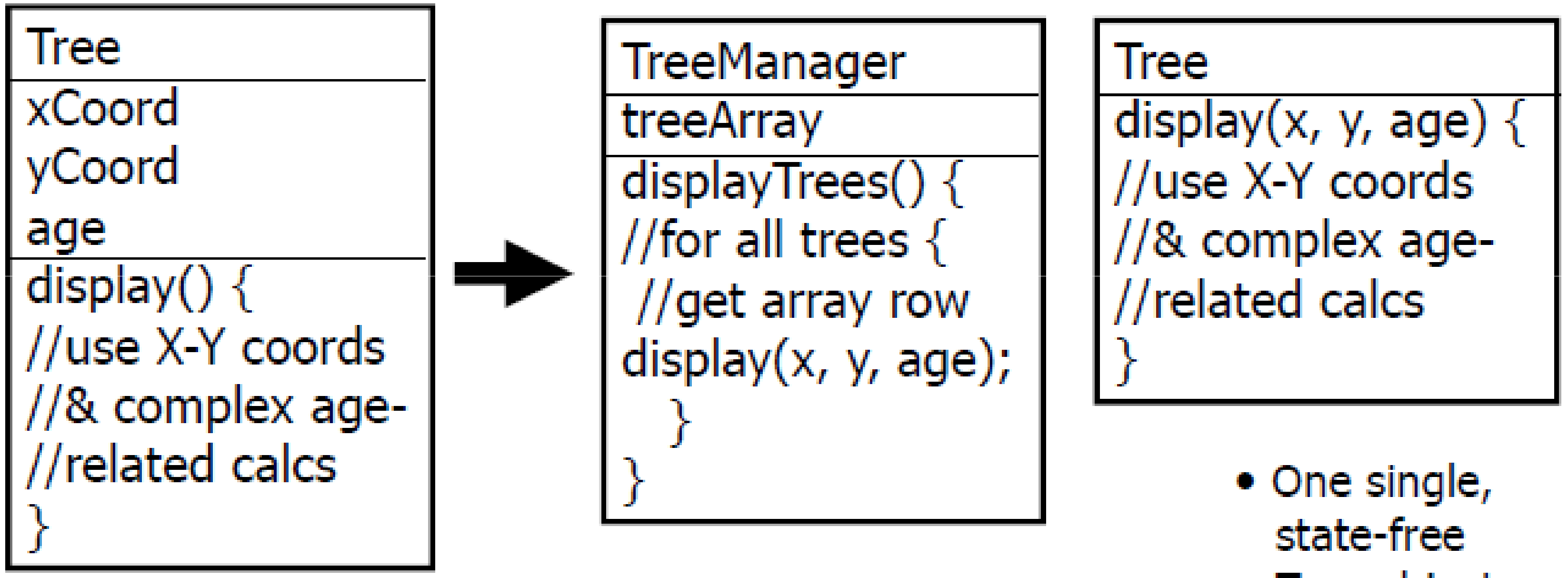
- Solution 2 makes it easy and clear to manipulate this set of handlers: the collections interface is well known and everybody understands how to iterate over a List or what not.
- If the handlers can completely handle a request on their own, Solution 2 is fine. The handlers do not have a reference to other handlers, which makes the handler interface simple. You can add or remove handlers from the middle of the chain.
- One problem with Solution 2 is that a handler cannot do pre-processing or post-processing on the request. If this functionality is required, then Chain of Responsibility is better. In CoR, the handler is the one responsible for delegating to the next handler on the chain, so the handler can do pre-processing and/or post-processing, including modifying or replacing the response from the next handler on the chain. In this way, CoR is very similar to Decorator; it's just the intent that's different.

Flyweight

Flyweight Pattern

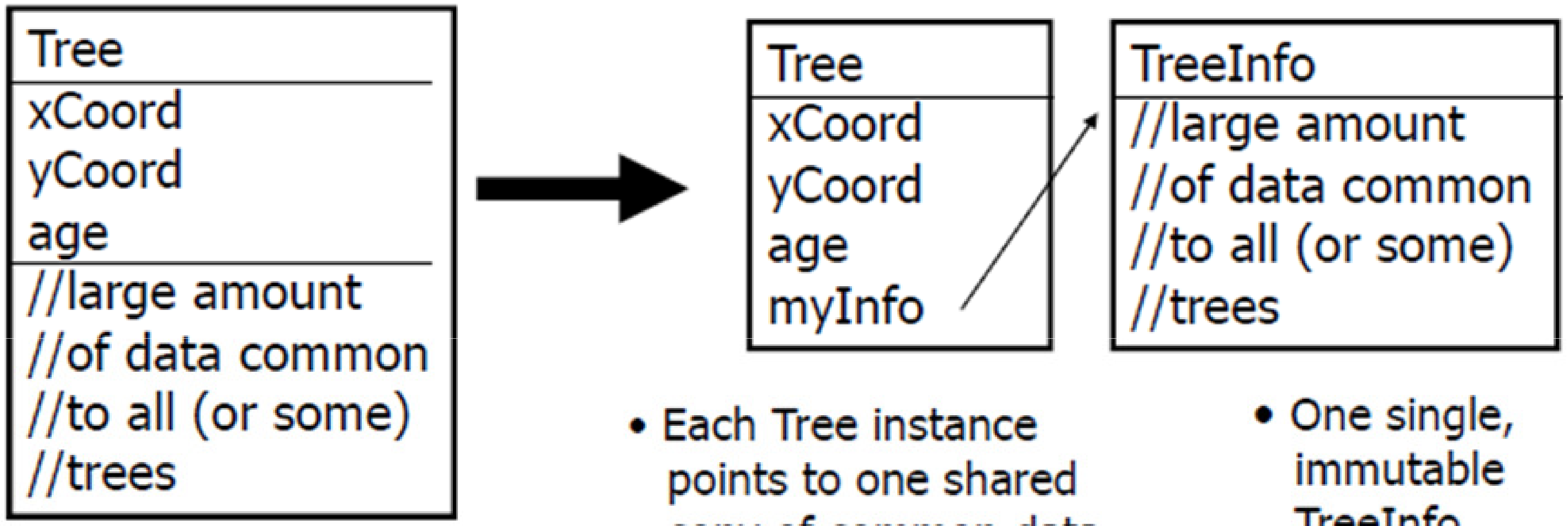
- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently
- Motivation
  - Can be used when an application could benefit from using objects throughout their design, but a naïve implementation would be prohibitively expensive
    - Objects for each character in a document editor
      - Cost is too great!
    - Can use flyweight to share characters

Head first example



- Each Tree instance maintains its own state
- State for all virtual Tree objects stored in treeArray
- One single, state-free Tree object

Another ex.

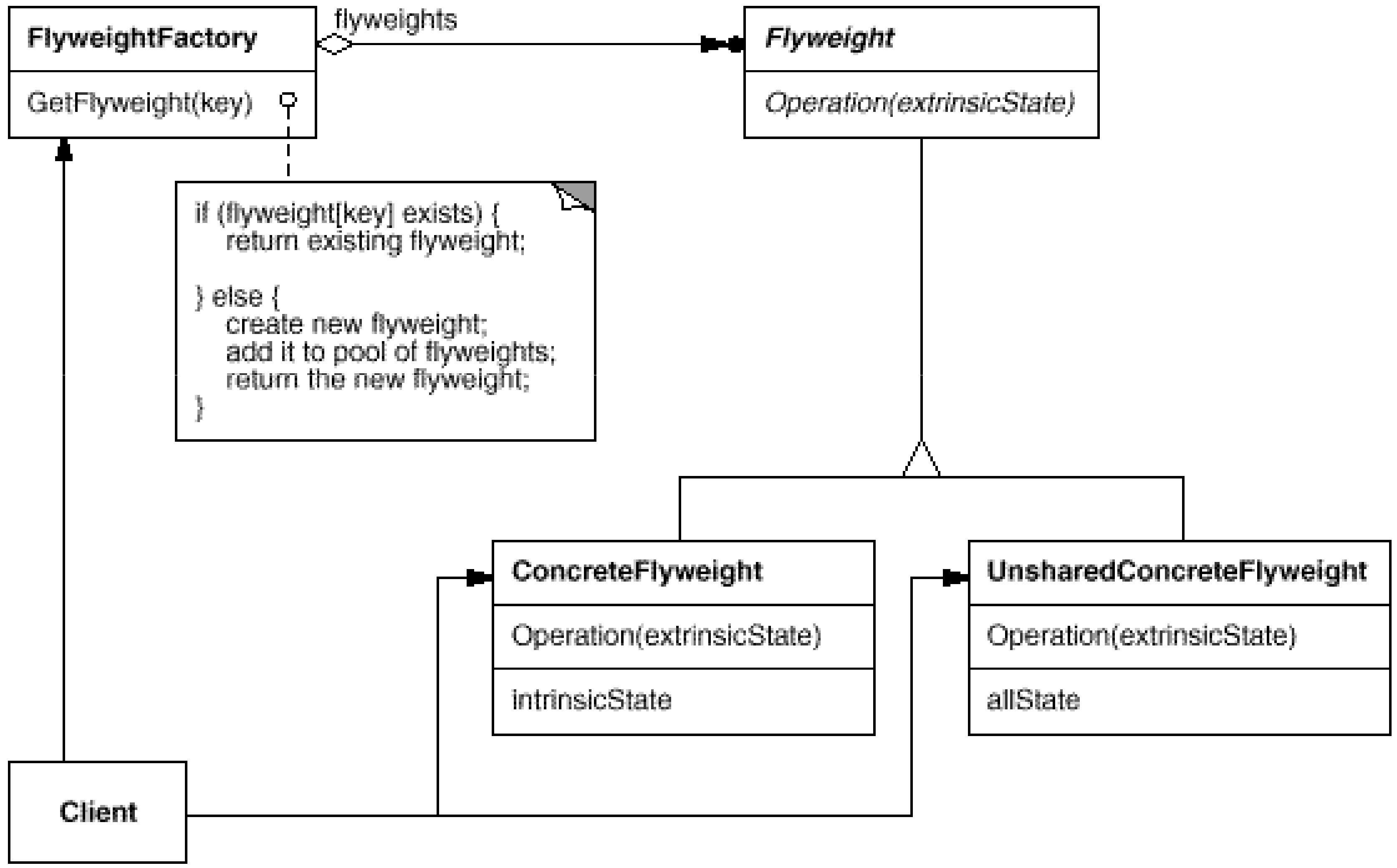


- Each Tree instance points to one shared copy of common data
- One single, immutable TreeInfo object
- Each Tree instance maintains its own copy of common data

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


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

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).