

Design patterns part 2

Noël Plouzeau



Factory Method

- Motivation
 - An interface is all is needed to request service from an object
 - A concrete class is required to create an object
 - Very often we don't have and don't need this information



Factory Method (cont'd)

- Intent
 - To provide a means for object creation that does not require selection of a concrete class



Factory Method (cont'd)

4

- Participants
 - Interfaces
 - Creator, Product
- Implementation classes
 - ConcreteCreator, ConcreteProduct



Responsibilities

5

- Product
 - Abstracts the class by providing service operations that the concrete object will support
- Creator
 - Defines an interface for creating products



Responsibilities (cont'd)

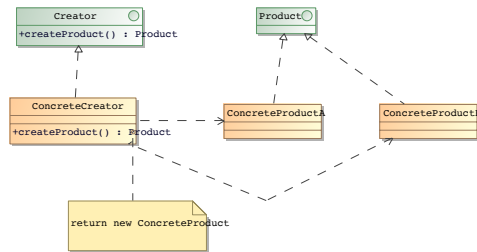
6

- ConcreteProduct
 - Implements the methods of Product
 - This is the concrete class that will be created
- ConcreteCreator
 - Implements the creation operation by doing the constructor call



Structure

7



Collaborations

8

- The collaboration here is trivial
- a client object (not represented in the pattern) calls a factory method
- the implementation chooses a concrete class
- calls the constructor using new
- return that object



Example

9

```
package fr.istic.nplouzeau.factoryMethod;
import java.util.List;
/**
 * An example of application of the Factory Method design pattern
 */
public interface ListCreator {
    /**
     * Allocates a list, with an implementation that switches
     * between ArrayList and LinkedList
     * @return a new list object
     */
    public List<String> createList();
}
```



```

public class ListCreatorImpl implements ListCreator {
    boolean useArray = false;

    /**
     * Allocates a list, with an implementation that switches
     * between ArrayList and LinkedList
     * @return a new list object
     */
    @Override
    public List<String> createList() {
        if (useArray) {
            useArray = !useArray; return new ArrayList<String>();
        } else {
            useArray = !useArray; return new LinkedList<String>();
        }
    }
}

```



The Memento design pattern

Motivation

- in many cases one needs to save the state of an object and then restore it later
- but at the same time encapsulation must be preserved



Solution: Memento

Intent

- to provide a mechanism that allow state storage
- will masking internal state of an object

Participants

- Memento, Originator, Caretaker, ConcreteMemento



Responsibilities

- Memento
- This is an empty interface (no operations)
- Serves as a type marker, not a service access (no services provided beyond assignment and reference R/W)



Responsibilities (cont'd)

- Originator
- has an internal state
- is able to create and read concrete memento objects upon request to save and restore its state



Responsibilities (cont'd)

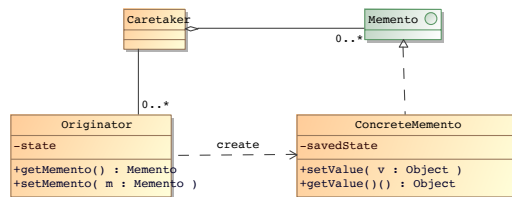
- Caretaker
- storage of mementos
- asks for mementos from originator and returns them when state restoration is needed



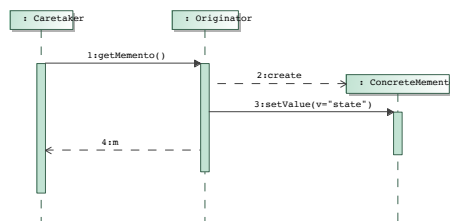
Responsibilities (cont'd)

- Concrete memento
 - contains all data necessary for restoring an originator state
 - provides accessors for the originator

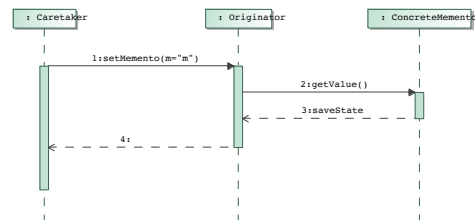
Structure



Collaboration: save



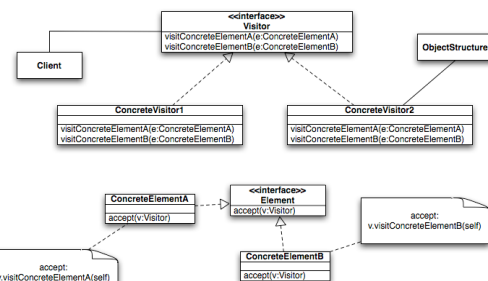
Collaboration: restore



The Visitor design pattern

- Intent
 - organize processing methods in a graph structure
 - separate types and processing
 - make addition of new algorithms easy

Structure



Responsibilities

- Visitor
 - Define a processing operation for each type of graph element
 - This operation will be called by the elements themselves
- Element
 - Define an operation that will be called by the visitor



Responsibilities (cont'd)

- ConcreteElement
 - Implementation of Element::accept()
 - To call the processing operation that is appropriate for the concrete element's type
- ConcreteVisitor
 - Implementation of a process with type specific methods



Responsibilities (cont'd)

- ObjectStructure
 - stores the elements of the graph to be visited
 - this role may be assigned to the elements themselves (use of attributes to reference neighbor nodes)
 - or it can be assigned to a specific class (e.g. Tree)



Separation of concerns

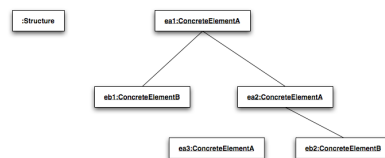
- Each concrete visitor knows what to do for each concrete element
- Each concrete element knows what to say to the visitor to trigger processing
- We have a separation between types (concrete elements) and processing (concrete visitor)

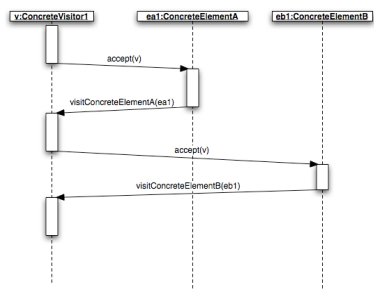
Code matrix

To compute length of a keyword

	Print	Length	Condenser
Section	1A	2A	3A
Title	1B	2B	3B
Keyword	1C	2C	3C
Document	1D	2D	3D

Object diagram





The Builder design pattern

Intent

- to separate a construction algorithm from the constructed structure

Motivation

- a construction algorithm for a composite structure can stay the same in spite of implementation changes of the structure representation

Participants of Builder

Builder

- Defines the operations for build parts and assembling them

ConcreteBuilder

- implements Build's operations

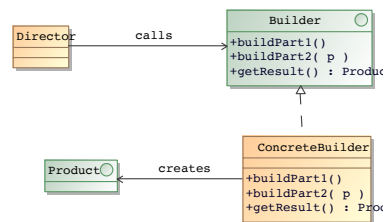
Product

- The structure under construction

Participants of Builder (cont'd)

- Director
- The algorithm to construct the product by calling operations of creation and assembly

Structure

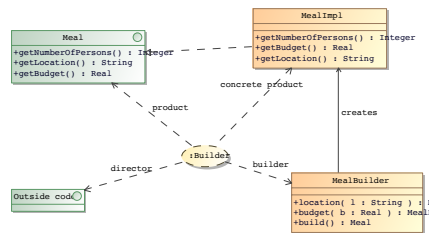


Example

- How to build a complex object with required and optional data
- Put the required data in the constructor
- Apply the Builder pattern to manage optional parts
- Protect construction inside the concrete builder

Example structure

34



Meal.java

35

```
package fr.istic.nplouzeau.builder;

/**
 * Author: PLOUZEAU, Noël
 * ...
 */

public interface Meal {

    public int getNumberOfPersons();

    public String getLocation();

    public double getBudget();

}
```



MealImpl.java

36

- Declares a public inner class as the Meal builder
- Mandatory data is passed in the builder constructor to make sure it is given
- Optional data is set by calling a data specific operation
- Final product is returned with a specific builder operation



Builder definition

```

public static class MealBuilder {
    private int numberOfPersons; // Mandatory
    private String location = "<none defined>"; // Optional
    private double budget = 0; // Optional
    public MealBuilder(int numberOfPersons) {
        this.numberOfPersons = numberOfPersons;
    }
}

```



Builder definition (cont'd)

```

// Store the optional data in the builder
public MealBuilder location(String l) {
    this.location = l;
    return this; // Allows for chaining calls
}

```



Builder definition (cont'd)

```

// Final construction of product
// Builds a meal from stored parameters
public Meal build() {
    return new MealImpl(this);
}

```



MealImpl constructor

```

c private MealImpl(MealBuilder builder) {
c     numberOfPersons =
builder.numberOfPersons;
c     location = builder.location;
c     budget = builder.budget;
c }

```



Example of use

```

c @org.junit.Before
c public void setUp() throws Exception {
c     meal = new
MealImpl.MealBuilder(nbPersons)
.location("ISTIC").build();
c }

```



Example of use (cont'd)

```

c @org.junit.Test
c public void testGetLocation() throws Exception
c {
c     Assert.assertEquals("ISTIC",
meal.getLocation());
c }

```

