Example calculator design

Dr. Onno van Roosmalen
LaTeX version: Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

September 25, 2017

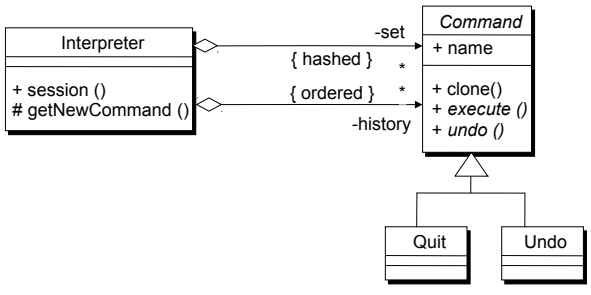
Content

Example Design

Part 1

- Implement a command interpreter. Introduce a set of commands that are accepted during a command-interpreter session. The effect of each command can vary with the application.
- Two special command names are Quit and Undo. These must be accepted. Quit stops the session. Undo removes commands from the history of executed commands in reverse chronological order. It notifies the user of the name of the undone command. The number of executed commands is thus decremented.

Interpreter Design



Interpreter Implementation

```
public class Interpreter {
    protected Stack history = new Stack();
    protected HashTable set = new HashTable();

    public void session() {
        while(true) {
            Command command = getNewCommand();
            if (command instanceof Quit) {
                return;
            } else {
                command.execute();
            }
        }
    }

    protected Command getNewCommand() {
        String s = input.getString();
        while (!set.containsKey(s)) {
            display.printError("Not a valid command; try again!");
            s = input.getString();
        }
        return ((Command)set.get(s)).clone();
    }
}
```


Undo Implementation

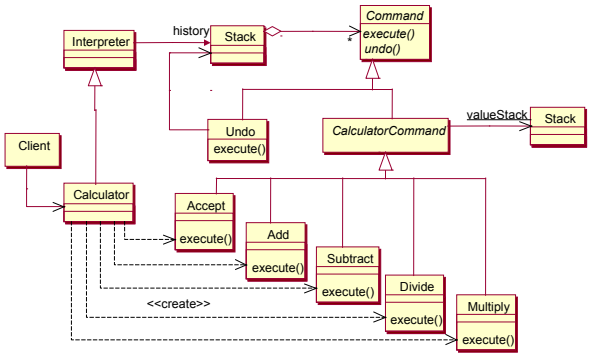
```
abstract class Command implements Cloneable {
    public abstract void execute();
    public abstract void undo();
}

class Undo extends Command {
    protected Stack history;
    public Undo(Stack h) { history = h; }
    public void execute() { ((Command) history.top()).undo(); }
    public void undo() {}
}
```

Calculator

Part 2

- Using the interpreter, build a calculator with Reverse Polish Notation (RPN). It uses commands like
  - accept "n"
  - *
  - /
  - +
  - -
  - quit
  - undo
- The reverse polish notation uses the stack concept for prescribing the order of executing the commands.
- The calculator commands must be "un-doable".

Calculator Design



Calculator Implementation

```
class Calculator extends Interpreter {

    public Calculator(){
        set.put("quit",   new Quit());
        set.put("undo",   new Undo(history));
        set.put(   "+",   new Plus(history));
        set.put(   "-",   new Min(histroy) );
        set.put(   "*",   new Mul(history) );
        set.put(   "/",   new Div(history) );
        set.put("accept",new Accept(history));
    }
}
```


Design Rationales

- Use specialization of abstract classes to avoid unnecessary coupling between classes
- Obtain extendibility with regard the type of requested commands
- Obtain reusability of certain complex or carefully optimized algorithms
- Obtain a readable and communicable design through proper separation of concerns

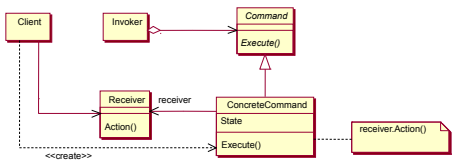
The essence of this can be captured in a design pattern

Command

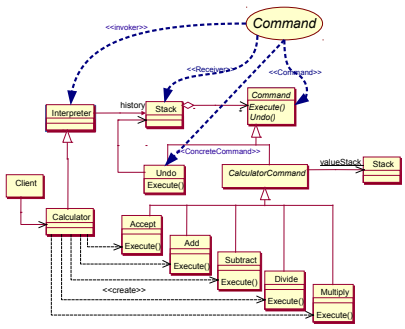
- **Classification**: Behavioral
- **Intent:** Encapsulate a request as an object, thereby letting you parametrise clients with different requests, queue or log requests, and support un-doable operations.
- **Motivation:** Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
  - E.g. user interface toolkits include objects that carry out a user request like
    - buttons
    - menus in response to user input.
  - The toolkit can't implement the request explicitly because only applications know what should be done on.

Command

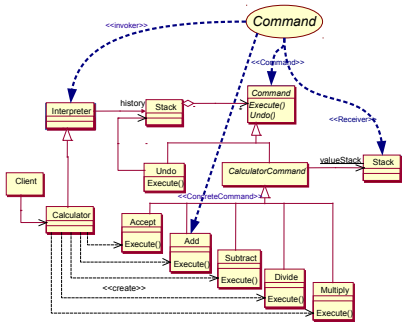
- **Structure**



Two command instances in calculator



Two command instances in calculator



Reflections

- The class names in the pattern reflect roles of classes played. The pattern is a meta-description of the design.
  - A mapping must be found:
    - Concrete command ← Mul, Min,
    - Client ← Calculator
    - Invoke ← Interpreter
    - Receive ← Stack
- The command interpreter implementation contains several design patterns (as we will see later).

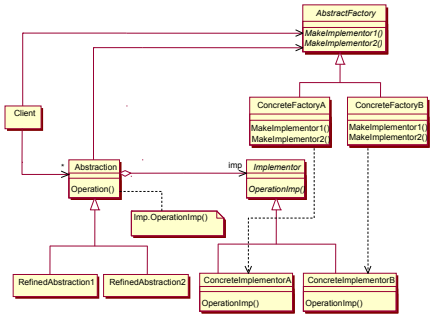
Command

- **Participants:**
  - *Command*
    - declares an interface for executing an operation
  - *ConcreteCommand*
    - defines a binding between a receiver object and an action
    - implements Execute() by invoking operation(s) on Receiver
  - Client
    - creates a ConcreteCommand object and sets its receiver
  - Invoker
    - asks the command to carry out a request.
  - Receiver
    - knows how to perform the operation associated with carrying out a request.

Command

- **Consequences**
  - Command de-couples the object that invokes the operation from the one that knows how to perform it.
  - Commands are first class objects. They can be manipulated and extended like any other object.
  - You can assemble commands into a composite command to obtain a form of macro's. (This can be achieved wit the composite pattern).
  - It is easy to add new commands. Only the creating client needs to be adapted.

Combining Patterns



Prototype

- **Classification:** Creational
- **Intent:** Specify the kind of objects to create using a prototypical instance, and creating new objects by copying this prototype.
- **Motivation:**
  - In the command-interpreter example, a prototype of each command was stored in a table.
  - The table provides a link between the strings that can be typed by the user and the actual command objects.
  - Because a history is kept, with each command issued, even of the same type, state information has to be retained.
  - Hence a new command-object has to be created at each command issued by the user.
  - The command retrieved from the table is copied, to keep the interpreter independent from specific command classes.

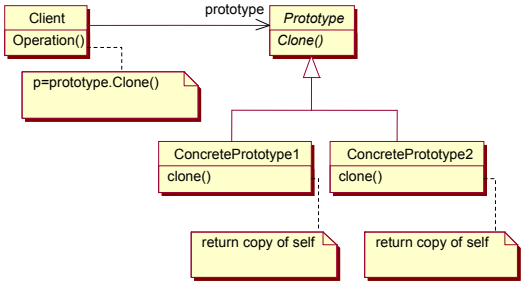
Interpreter Implementation

The `Interpreter.getcommand()` method returns a clone of one of the members of the command set.

```
protected Command getNewCommand() {
    String s = input.getString();
    while (!set.containsKey(s)) {
        display.printError("Not a valid command; try again!");
        s = input.getString();
    }
    return ((Command)set.get(s)).clone();
}
```

Prototype

- Prototype Pattern structure



Prototype

- **Participants:**
  - *Prototype:*
    - declares an interface for cloning itself :
  - *Concrete prototype:*
    - implements an operation by cloning itself.
  - *Client:*
    - Only couples to abstract Prototype to obtain clones of concrete versions.

Prototype

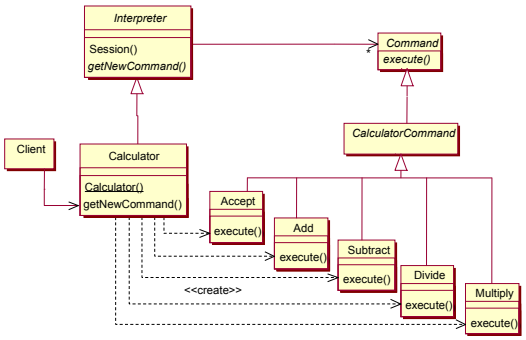
- **Instance**
  - Prototype ← Command
    - Clone() ← clone()
  - Concrete prototype ← Plus, Minus, . . .
  - Client ← Interpreter
    - Operation() ← getNewCommand
  - Prevents coupling
    - The Concrete Prototype class is invisible to the client

Factory Method

- **Classification:** Creational
- **Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
- **Motivation:** In the command interpreter example we have a class (Interpreter) that deals with objects that are subclass-instances of a single interface (Command).
  - We do not want to have the interpreter know all these subclasses. (We want to decouple the class from these concrete command classes for reuse of the Interpreter class).
  - We want to defer creation of the concrete commands to subclasses of the interpreter for two reasons
    1. Only this subclass knows the ConcreteCommand classes
    2. This subclass must determine the algorithm by which the concrete commands are created

Factory Method example

- Example



Interpreter with Factory Method

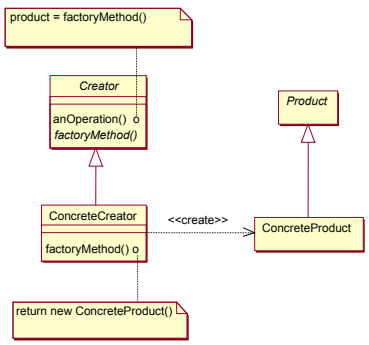
```
abstract class Interpreter {
    protected Stack history = new Stack();
    protected HashTable set = new HashTable();

    public void session() {
        while(true) {
            Command command = getNewCommand();
            if (command instanceof Quit) {
                return;
            } else {
                command.execute();
            }
        }
    }

    abstract protected Command getNewCommand();
}
```


Calculator with Factory Method

```
class Calculator extends Interpreter {
    Calculator() {
        set.put("quit", new Quit());
        set.put("undo", new Undo(history));
        set.put("plus", new Plus(history));
        set.put("min", new Min(histroy) );
        set.put("mul", new Mul(history) );
        set.put("div",
                new Div(history) );
        set.put("@accept",new Accept(history));
    }
    protected Command getNewCommand() {
        String s = input.getString();
        while (! set.containsKey(s)) {
            display.printError("Not a valid command; try again!");
            s = input.getString();
        }
        return ((Command)set.get(s)).clone();
    }
}
```


Factory Method

- Structure



Factory Method

- **Participants:**
  - *Product:*
    - defines the interface of objects the factory method creates.
  - *ConcreteProduct:*
    - implements the product interface.
  - *Creator:*
    - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
    - may call the factory method to create Product objects.
  - *ConcreteCreator:*
    - Overrides the factory method to return an instance of a ConcreteProduct
- **Applicability:**
  - Use when a class can't anticipate the class of objects it must create.
  - A class wants its subclasses to specify the objects it wants to create.
  - Classes delegate responsibility to one of several subclasses, and you want to localize the knowledge of such helper subclass is the delegate.

Factory Method

- **Prevents coupling:**
  - Factory method avoids having to specify the ConcreteProduct creations
  - The factory method defers decisions on the creation algorithm
  - Avoids coupling between Creator and the concrete product classes
- **Consequences:**
  - Provides hooks for subclasses.The factory principle (hiding creation in factory methods (either using abstract factory or factory method) adds flexibility to a design.
    (Note that the AbstractFactory class uses factory methods. It is a class with a factory method for each type of abstract product it is supposed to deliver.)
  - Connects parallel class hierarchies.

Factory Method

Note on the example

In the given interpreter/calculator example Prototype and Factory Method are combined. Prototype is used to dynamically decide which object must be created (using dynamic binding rather than explicit selection based on the string entered by the user). Factory Method is used to vary the algorithm by which Command objects are created with the subtype of Interpreter(e.g. a using graphical interface for the calculator may change this algorithm).

A factory method is not *required* to create a new instance. It may decide to use an existing, if the situation warrants it. See for instance `java.lang.Integer.valueOf(int)`.