Comp-303 : Programming Techniques
Lecture 22

Alexandre Denault
Computer Science
McGill University
Winter 2004

Last lecture . . .

- Chain of Responsibility is a useful pattern when you want to decouple sender and receiver.
- It allows you to use objects to handle events dynamicly (i.e. dynamic switch statements).

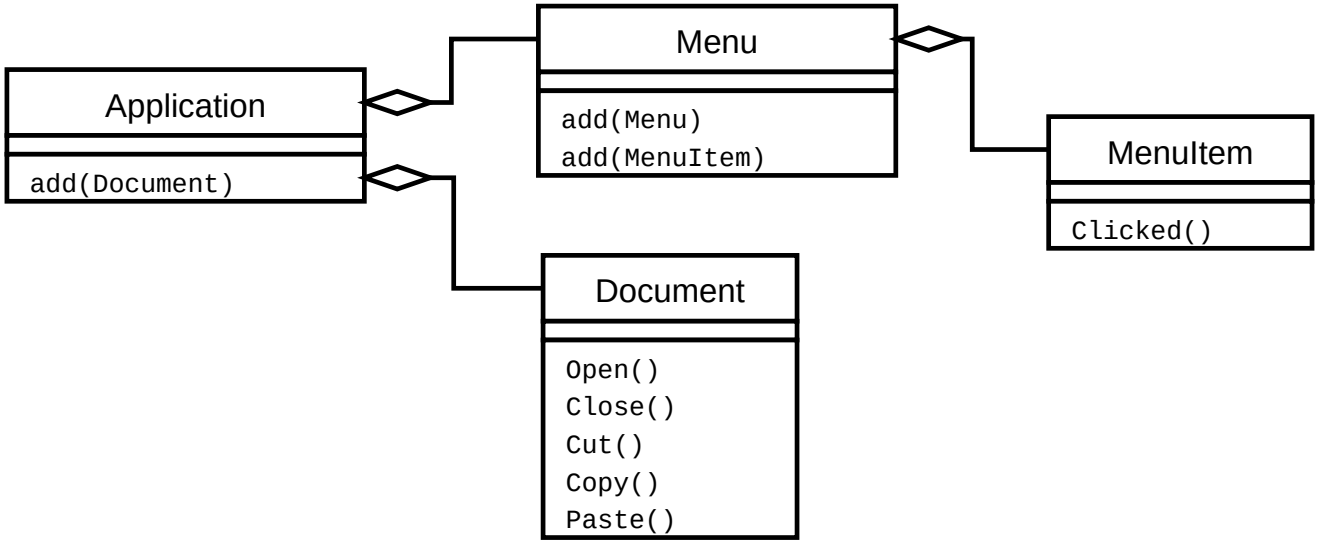
Typical Situation

- Today, our example starts a generic user interface (like a menu system).
- Designer can create the interface, but they can't know how people will use them?
  What happens when a user presses a button?
- We need a way to parameterize a function. In other words, we want to pass a function as an arguments.
- People who like C would smile and say *Function Pointers*! Ick!
- We want a more O.O. solution to this problem.
- We've already seen the solution with Threads, but we want to push this farther.

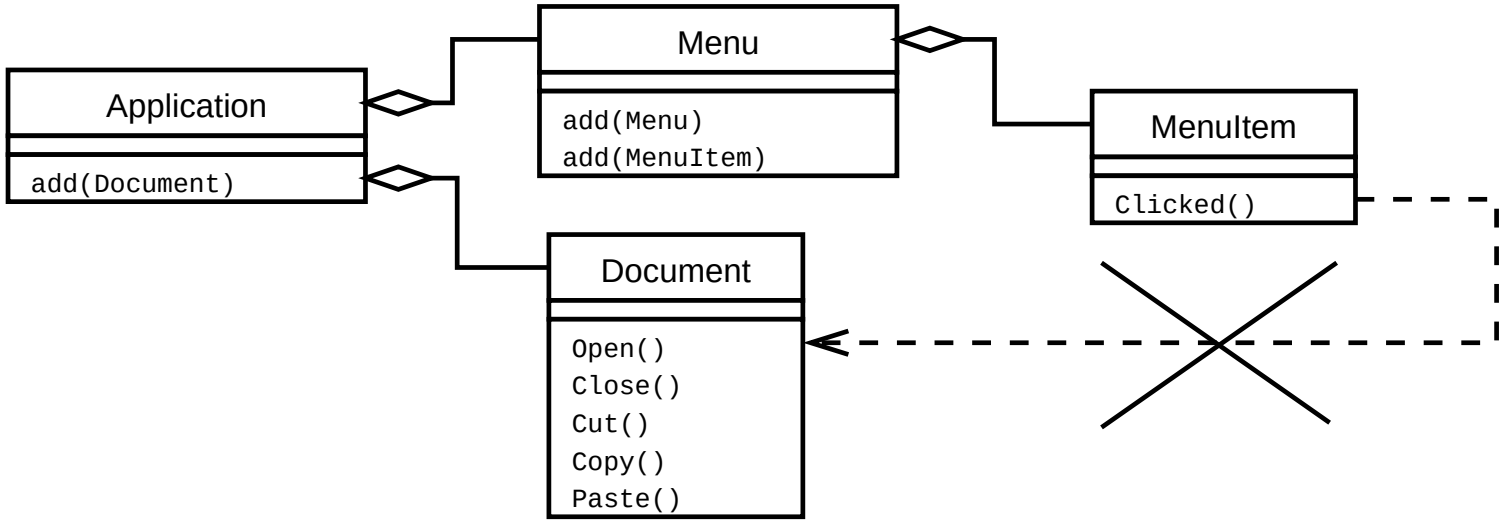
Menu system

- A menu system provides us a good example of the command pattern.
- Our menu system is composed of two classes : *Menu* and *MenuItem*.
- *Menu* is the container object, it contains either other *Menu* objects or *MenuItem* objects.
- *MenuItem* is the execution object, it executes an action when it is activated.
- We can think of our structure as a tree.
  - *Menu* objects are the nodes.
  - *MenuItem* objects are the leaves.

Architecture



Bad Scenario



Why is this a bad scenario?

- Such scenario would require multiple *MenuItem* (OpenMenuItem, PasteMenuItem, etc).
- However, *MenuItem* should not have any knowledge of Document.
  - *MenuItem* is a UI (user interface) object.
  - *Document* belongs in the application domain (specific to the program).
- In other words, the two classes belong in different domains (modules).

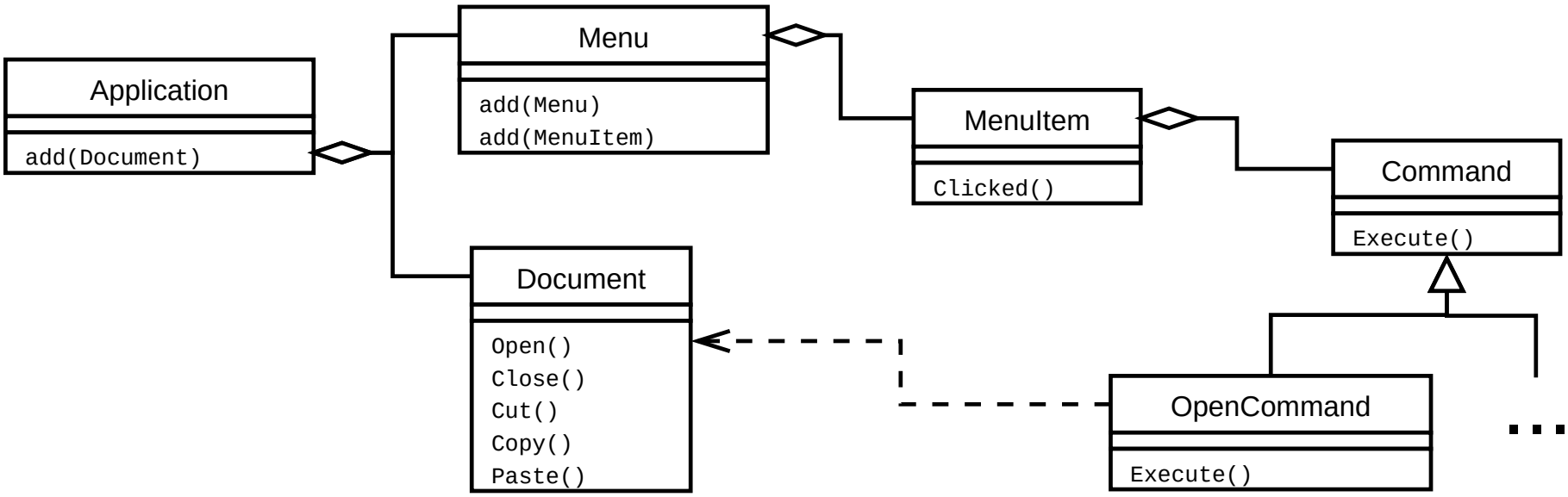
Command

- *Pattern Name* : Command
- *Classification* : Behavioral
- *Also Known As* : Action, Transaction
- *Intent* : Encapsulate a request as an object, thereby letting you parameterize clients with differents requests, queue or log requests, and support undoable operations.

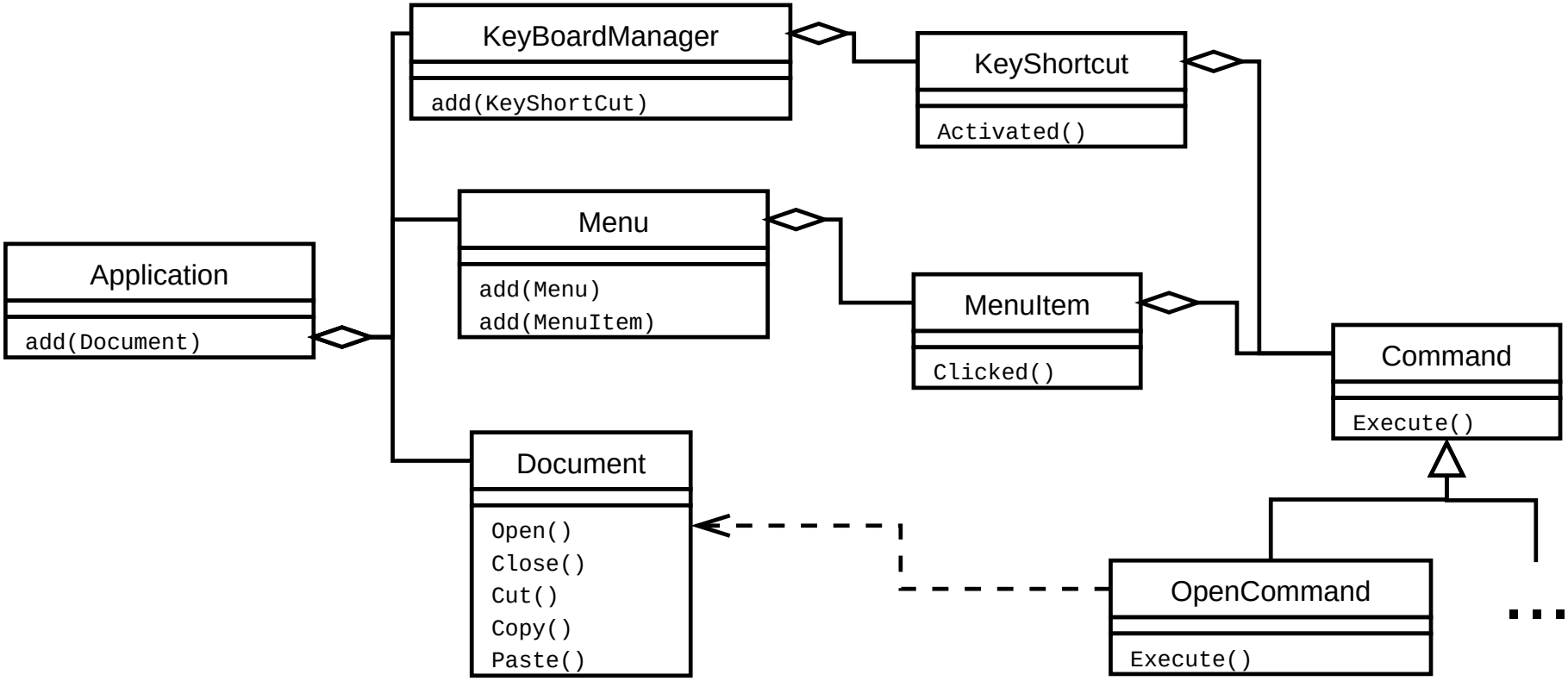
Motivation

- *MenuItem* must work in any application context. In other words, it shouldn't have any specific behavior hardcoded inside.
- We need to find a way to parameterize the behavior of *MenuItem*.
- Our solution is very similar to threads and *Runnable* objects.
- When we create a new *MenuItem*, we pass along a *Command* object.
- This *Command* object describes the behavior of that instance of the *MenuItem*.

Motivation (cont.)



Motivation (cont.)

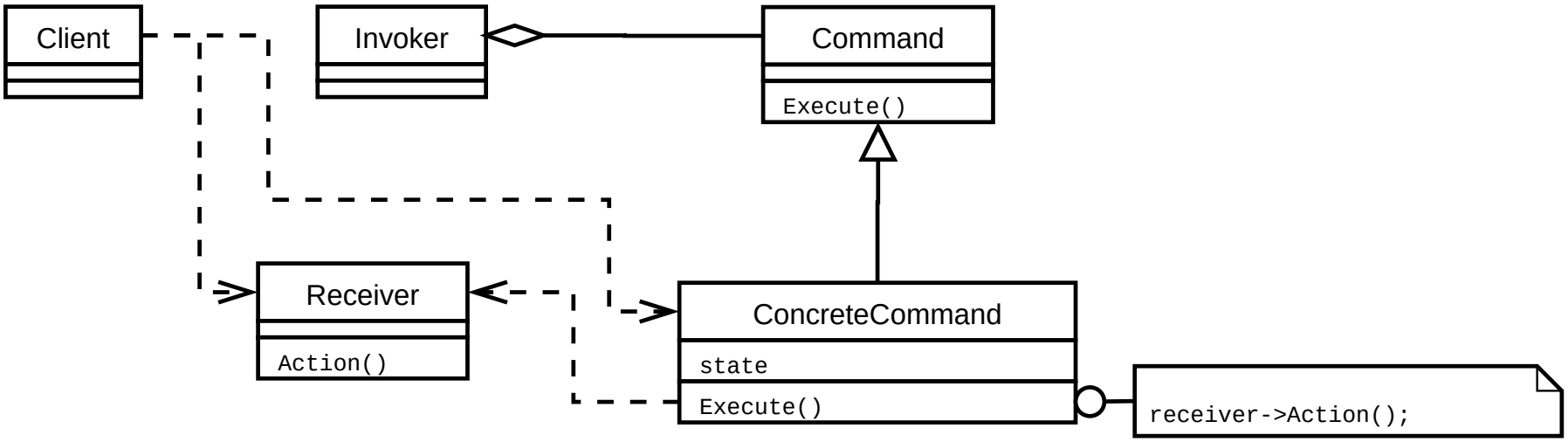


Applicability

Use Command when you want to ...

- ... parameterize objects by an action to perform. In procedural languages, this would be called a callback function.
- ... specify, queue and execute requests at different times.
- ... support undo/redo.
- ... support logging of changes so that they can be reapplied in case of a system crash (transaction).
- ... structure a system around high-level operations built on primitives operations.

Structure


Client
Invoker
Command
Execute()
Receiver
Action()
ConcreteCommand
state
Execute()
receiver->Action();

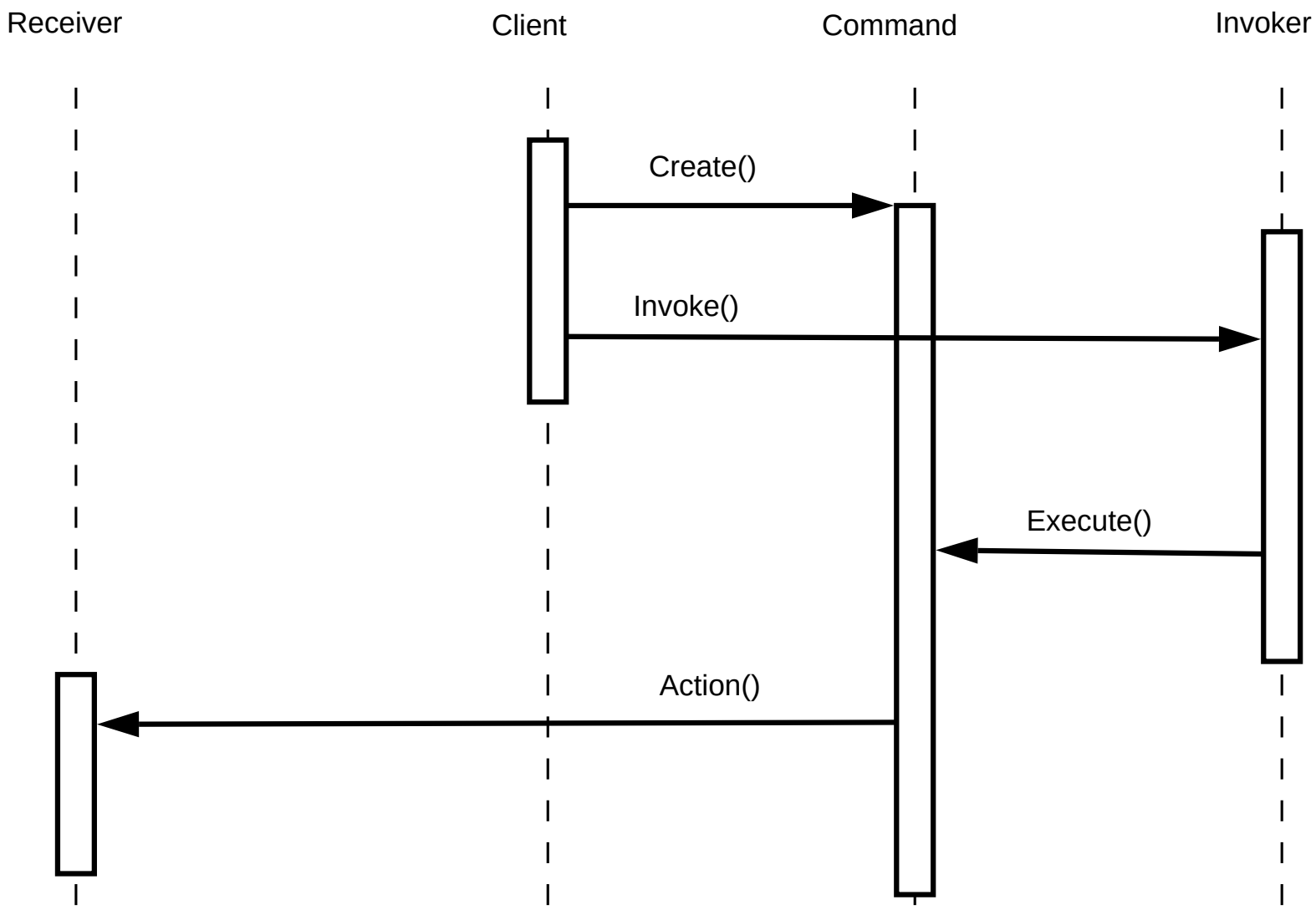
Participants

- Command : Declares the interface for executing a command.
- ConcreteCommand : Implements *Execute()* ny invoking the corresponding operation(s) on *Receiver*.
- Client : Creates a ConcreteCommand object and sets its receiver.
- Invoker : Asks the command to carry out the action.
- Receiver : Knows how to perform the operations associated with carrying out a request.

Collaborations

- *Client* creates a *ConcreteCommand*.
- *Invoker* stores the *ConcreteCommand*.
- *Invoker* issues a request by calling the *Execute* method on the *ConcreteCommand*.
- *ConcreteCommand* invokes the appropriate operations on the *Receiver*.

Collaborations (Action)



Consequences

- Command decouples the object that invoke the operation from the one that knows how to perform it.
- Commands are objects. They can be manipulated and extended like any other objects.
- You can assemble commands into a composite command (Macro).
- It's easy to add new commands.

Implementation

- How intelligent should a command be?
  - It may merely define a binding between a receiver and an action.
  - It may implement everything itself without delegating to a receiver at all.

Supporting Undo/Redo

- Commands can support undo and redo capabilities if they provide a way to reserve their execution.
- The *ConcreteCommand* object might need to store additionnal information such as
  - the receiver object.
  - the arguments to the operations.
  - the original values in the receiver that can cahnge as a result of handling the request.
- For multiple levels of undo/redo, the application needs a history list (of past commands).
- Be careful when implementing multiple levels of undo/redo. It's easy to lose coherency when executing multiple undo/redo.

Sample Code

```
public abstract class Command {

  State info;

  public void execute() {

  }

  public void undo()
    throw UndoUnsupportedException {

  }

  public void redo()
    throw RedoUnsupportedException {

  }
}
```

Command

```
public abstract class MenuItem {

  Command doWhenClicked;

  public MenuItem(Command behavior) {
    this.doWhenClicked = behavior;
  }

  public void whenClicked() {
    this.doWhenClicked.execute();
  }

}
```

Macro commands

```
public class MacroCommand extends Command {

Linklist commandList;

  public MacroCommand() {
    this.commandList = new LinkList();
  }

  public void add(Command behavior) {
     this.commandList.add(behavior);
  }

  public void execute(){
   // for each command in the linklist
   // command.execute();
  }
```

Macro commands

```
  public void undo(){
   // for each command in the linklist
   // command.undo();
   // Is this always a good idea?
  }

  public void redo(){
   // for each command in the linklist
   // command.redo();
   // Is this always a good idea?
  }

}
```

Known Uses

- Transaction Systems
- Menu system Systems
- Many systems that support redo/undo operations

Related Patterns

- Composite
- Memento
- Prototype

Summary

- The Command design pattern allows you to seperate the Invoker from the Receiver, thus allowing you to create parametrizable frameworks.
- The Command design pattern also allows you to do nifty things like queing commands, undo, redo, transactions, etc.

Tool of the day: Shell HomeGenie

- Shell HomeGenie lets you use virtually any web-ready computer, cell phone or PDA to control your home's lights, small appliances, and thermostat.
- You can even use the included wireless camera to make sure things are okay when you're not there.
- Current accessories include:
  - Power Switch
  - Programmable Thermostat
  - Wireless Camera
  - *Contact Sensor
  - *Motion Sensor
  - *Temperature Sensor
  - *Water Sensor

References

- These slides are inspired (i.e. copied) from these three books.
  - Design Patterns, Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; Addison Wesley; 1995
  - Java Design Patterns, a Tutorial; James W. Cooper Addison Wesly; 2000
  - Design Patterns Explained, A new Perspective on Object Oriented Design; Alan Shalloway, James R. Trott; Addison Wesley; 2002