OODP– Session 10

Session times

PT – Thursday 18:00-21:00

room: School of Pharmacy, John Hanbury Lecture Theatre

FT - Tuesday 13:30-17:00 room: Malet 404

Email: oded@dcs.bbk.ac.uk

Web Page: http://www.dcs.bbk.ac.uk/~oded

Visiting Hours: Tuesday 17:00 to 19:00

Anti-Patterns

What are “Anti-Patterns”?

A commonly occurring bad software engineering pattern. (name inspired by the GoF patterns)

Some of the important features of anti-patterns

1. In the first place they may seem like a good idea
2. There is a refactored solution to the problem

Sources:

1. Internet
2. ***AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*** **by William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, Thomas J. Mowbray**
3. Antipatterns: Identification, Refactoring and Management by Phillip A. Laplante, Colin J. Neill

Root Causes “Seven Deadly Sins”

1. Haste
2. Apathy
3. Narrow Mindedness
4. Sloth
5. Avarice
6. Ignorance
7. Pride

Haste

Compromising software quality in order to get things done ASAP.

Two problems that repeatedly occur

1. Testing is sacrificed
2. Refactoring is postponed
    - Not having mechanism (such as design patterns) in place when needed will force workarounds and these are very hard to fix at a later stage

Apathy

Not bothering to fix known problems before they have impact .

Example

- Not bothering to partition the code properly

Narrow-Mindedness

Sticking to old traditions with known problems despite the existence of known solutions

Example

- Not using meta-data

Sloth

Making “easy answer” decisions

OODP example:

- Not bothering to design an interface properly

Avarice

Making thing too detailed and too complex

In general

- An over complicated solution takes more time code, is harder to maintain and use

Ignorance

Not bothering to understand the problem

- Using code generation tools without bothering to understand the type of code they generate

Pride

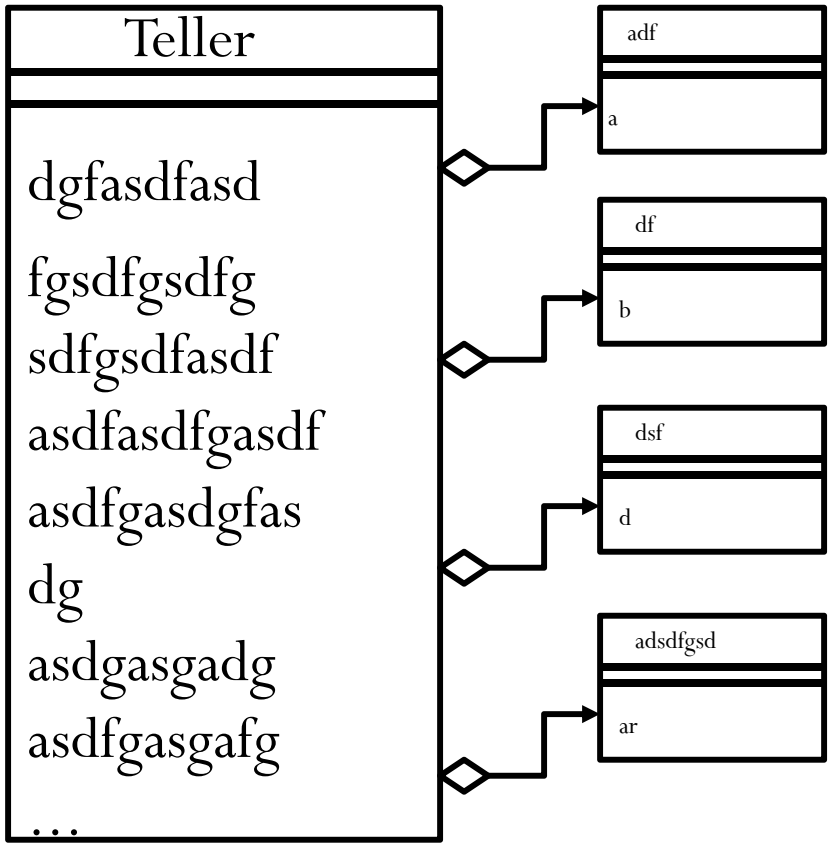
My solution is always the best

- Rewriting everything when there are known existing solutions

The Blob

A design in which there is one giant class that encapsulates all the processes while the rest of the classes are just used for encapsulating data:

Root causes: Sloth, Haste



- Maybe due to legacy code
- All advantages of OO lost
- Maybe the result of using just one controller

Functional Decomposition

Encapsulating every each method in one class.

Root Causes: Avarice, Greed, Sloth

Symptoms:

- Classes with method names
- Classes with single methods
- All attributes are private and used only inside the class

Damage:

- No use of OO features
- No chance for reuse
- Virtually impossible to understand the code

Data Base as IPC

A data base is used as means of communication instead of using direct messaging techniques.

Occurs many times since people do not know how to use available IPC (inter process communication) message queuing systems

Problems:

1. Extremely inefficient
2. Dealing with data-base is more complicated

Input Kludge

Not specifying what happens if the input is not valid.

Usually detected by end-user and not by programmers.

Problems:

1. End users may have unexplainable bugs
2. Security vulnerabilities

Interface Bloat

Designing an interface that is supposed to cover everything hence is way to big.

This is especially problematic when all the objects that implement the interface only implement a small portion of it.

Circular Dependancy

Circular dependency between different modules of a program:

Damages:

1. Reuse of single modules prevented because of tight dependency between modules created by the circular dependency
2. Small changes in once module have a higher chance of spreading to other modules.

This is a big enough issue to actually have dedicated tools for detecting such problems

Sequential coupling

A class that requires its methods to be called in a specific order.

Problem:

1. The knowledge of what exact order is required must be passed on to whom ever uses the class in the future, failing to pass this knowledge on may result in a tedious learning process.

Yo-yo problem

Using so much inheritance that the inheritance graph becomes to big accommodate on one screen.

This is one of the reasons why composition is preferable to inheritance.

Magic numbers

Including unexplained number in code.

Problem:

1. How should someone know if it is possible to change

Copy and paste programming

Very similar code in different places usually as a result of copy paste.

Problems:

1. If one does not fully understand the copied code changing it may result in unexpected bugs
2. Change of context may cause bugs
3. Code may become obfuscated in case it was not fully adapted to new purpose

When a potential copy paste situation arises one should think of other possible solutions.

Premature optimization

Optimizing before the need arises

Problems:

1. Unnecessary complication of code
2. Preferring performance issues to good architecture
3. Waste of time – was the optimization really necessary/

Back to Patterns

Memento

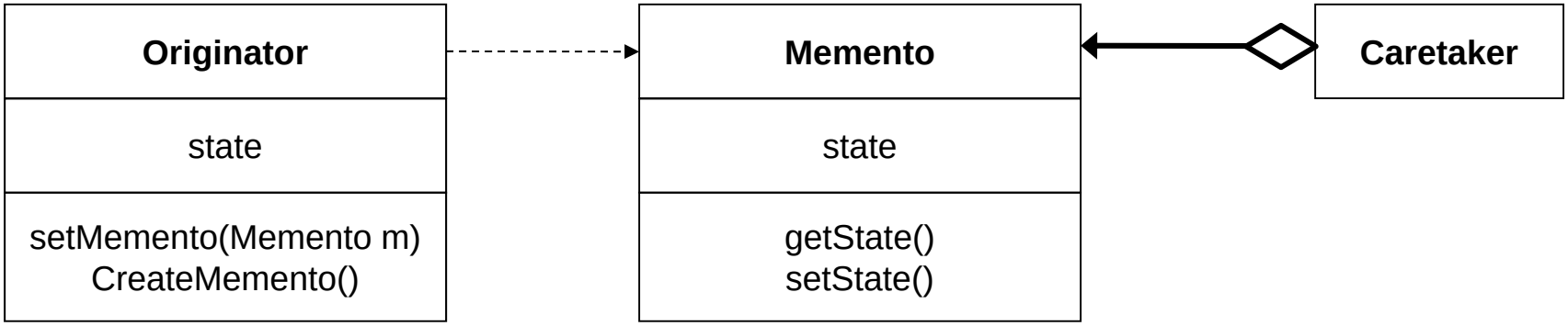
Memento

Problem: How to store the internal state of an object without violating encapsulation?

Motivation:

- Support undo

Memento



This doesn't end here!

Only the originator should be able to access and store information in the memeto

How?

Memento Code

```
public class Originator {
    private Integer state;

    public Originator(Integer initialValue) {
            this.state = initialValue;
    }

    public Integer whatNext(){
            state = state + 5;
            return state;
    }

    public Memento saveToMemento() {
            return new Memento(state);
    }

    public void restoreFromMemento(Memento memento) {
            state = memento.getSavedState();
    }

    public static class Memento {
        private final Integer state;

        private Memento(Integer state2) {
            state = state2;
        }

        private Integer getSavedState() {
            return state;
        }
    }
}
```

- Nested static class – can only access class its enclosing class and none of the enclosing class's members.
- Private memento methods – only enclosing class can access them and in this case create them
- Other classes can reference memento objects

Memento Code

```
public class OriginatorTest {

@Test
public void test() {
        Originator.Memento mem;

        Integer initialValue = 10;
        Integer firstExpectedResult = 20;
        Integer secondExpectedResult = 15;
        Integer firstActualResult,secondActualResult;

        Originator originator = new Originator(initialValue);
        mem = originator.saveToMemento();

        originator.whatNext();
        firstActualResult = originator.whatNext();
        assertEquals("Wronge answer! ",firstExpectedResult, firstActualResult);

        originator.restoreFromMemento(mem);

        secondActualResult = originator.whatNext();
        assertEquals("Wronge answer! ",secondExpectedResult, secondActualResult);
    }

}
```

Discussion

Why Use?

- Want to store internal state without
- Not just use other peoples code but override some of it

Issues:

- More classes
- Works in C++ what about other languages
- Cohesion

Composite

Composite Design-Pattern

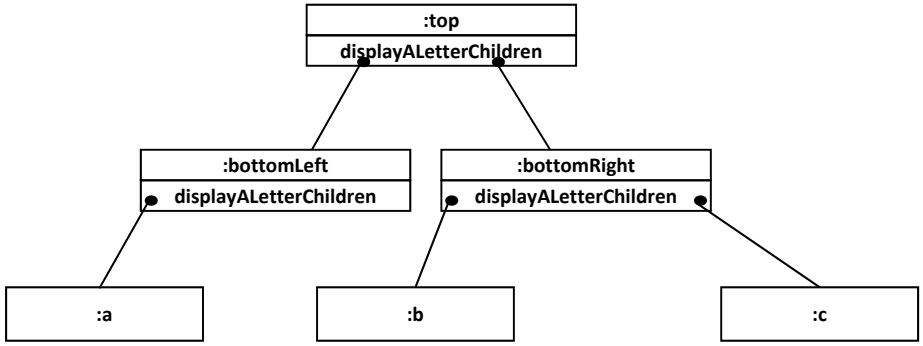
Problem: How to enable a group of objects to be treated like a single object

Motivation:

- Hide the fact that the client is actually interacting with a group of objects

Solution idea:

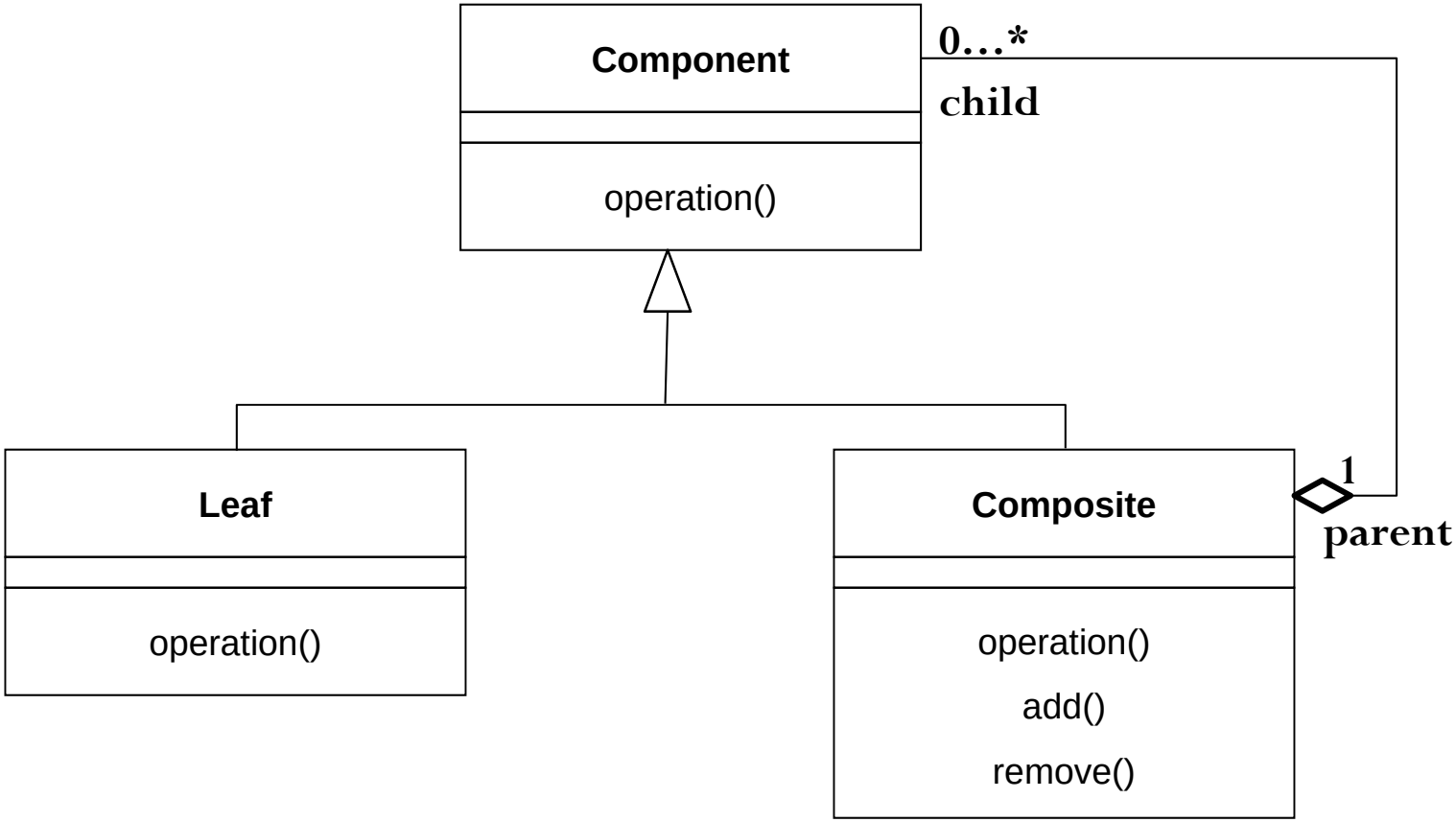
- compose the objects into a tree structure



The code can be cloned from:

git://github.com/odedlac/MacroCommand.git

Composite Design-Pattern



Example: The classes that do the doing

displayALetter.java

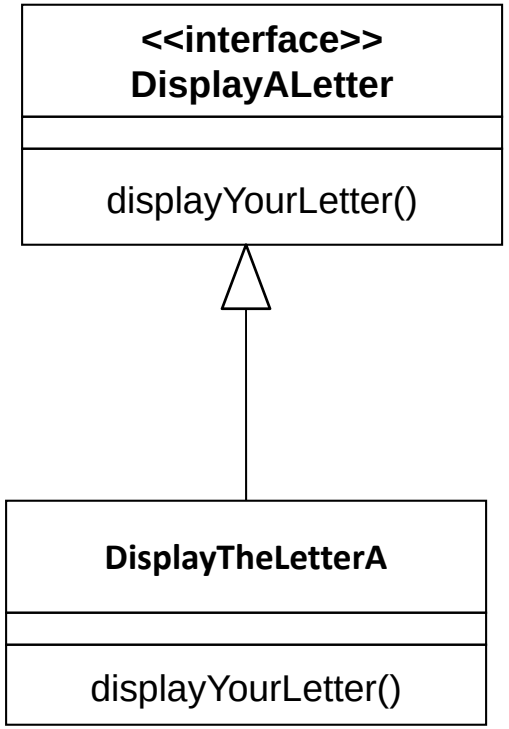
```
public interface DisplayALetter {
        void displayYourLetter();
}
```

DisplayTheLetterA.java

```
public class DisplayTheLetterA implements DisplayALetter {

        @Override
        public void displayYourLetter() {
                System.out.println("A");
        }

}
```



Example: The class that does the connecting

CompositeDisplayALetter.java

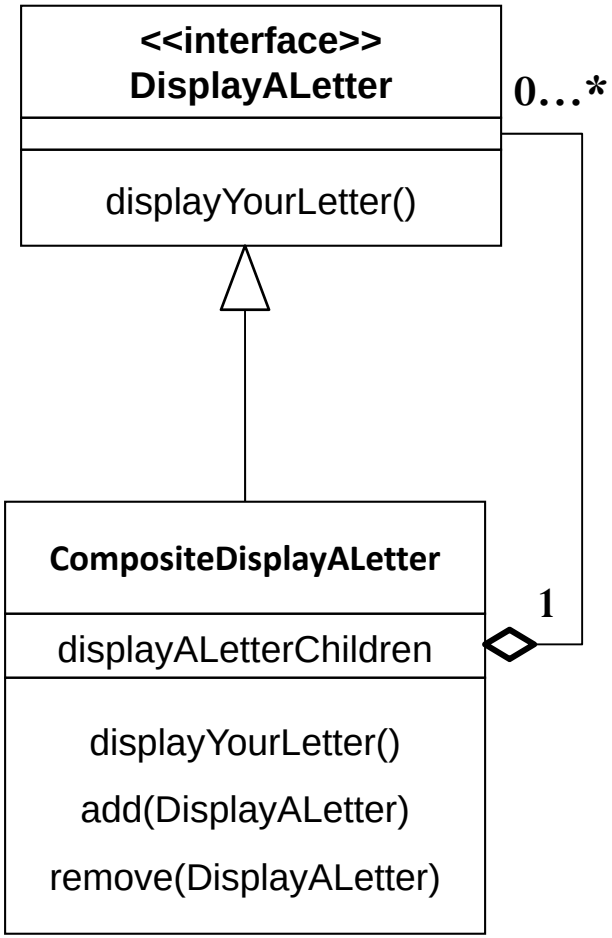
```
public class CompositeDisplayALetter implements DisplayALetter {

    //Collection of child graphics.
    private ArrayList<DisplayALetter> displayALetterChildren =
                        new ArrayList<DisplayALetter>();

    @Override
    public void displayYourLetter() {
        for (DisplayALetter displayALetter : displayALetterChildren) {
            displayALetter.displayYourLetter();
        }
    }

    //Adds the graphic to the composition.
    public void add(DisplayALetter displayALetter) {
            displayALetterChildren.add(displayALetter);
    }

    //Removes the graphic from the composition.
    public void remove(DisplayALetter displayALetter) {
            displayALetterChildren.remove(displayALetter);
    }
}
```



Example: A Test

CompositeDisplayALetterTest.java

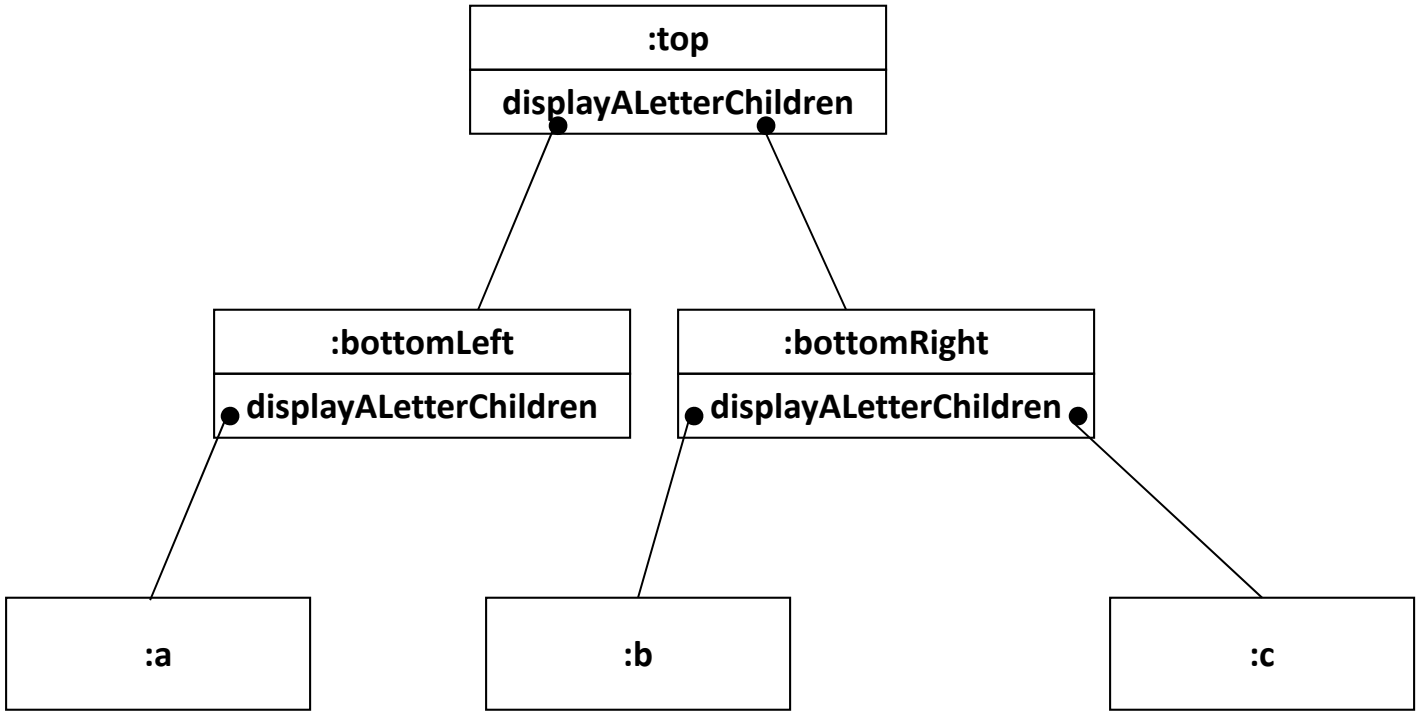
```
@Test
public void test() {
            DisplayALetter a = new DisplayTheLetterA();
            DisplayALetter b = new DisplayTheLetterB();
            DisplayALetter c = new DisplayTheLetterC();

            CompositeDisplayALetter top = new CompositeDisplayALetter();
            CompositeDisplayALetter bottomLeft = new CompositeDisplayALetter();
            CompositeDisplayALetter bottomRight = new CompositeDisplayALetter();

            top.add(bottomLeft);
            top.add(bottomRight);
            bottomLeft.add(a);
            bottomRight.add(b);
            bottomRight.add(c);

            top.displayYourLetter();
}
```

Example: The Structure we got



Discussion

Why Use?

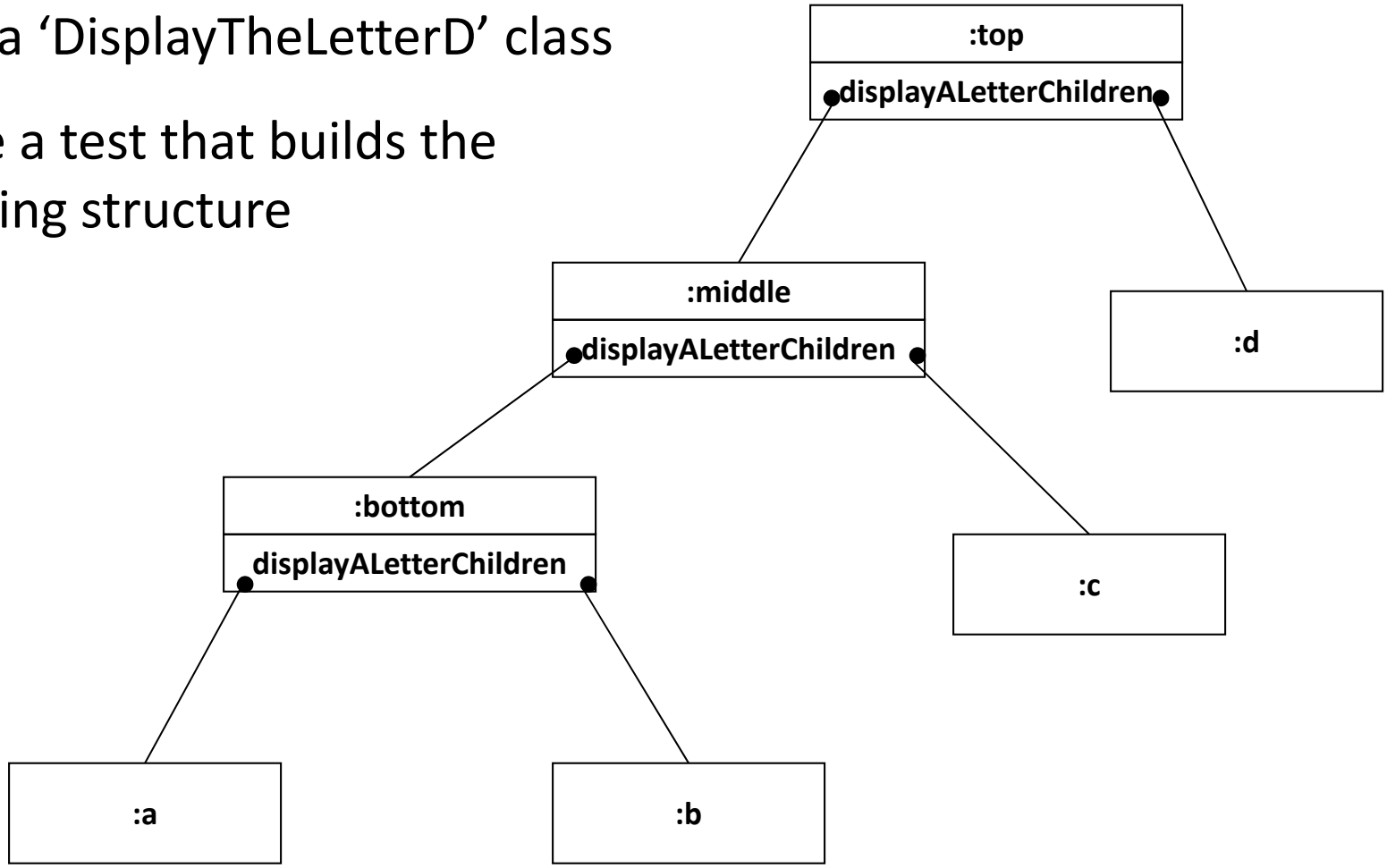
- Hide the fact that the client is actually interacting with a group of objects

Issues:

- More classes
- The structure can become very large and hence hard to manage

Todo:

1. Would you use a Factory method to create such a structure
2. Write a ‘DisplayTheLetterD’ class
3. Create a test that builds the following structure



Command

Command Design-Pattern

Problem: How to decouple the request for a command to performed and the timing of the performance

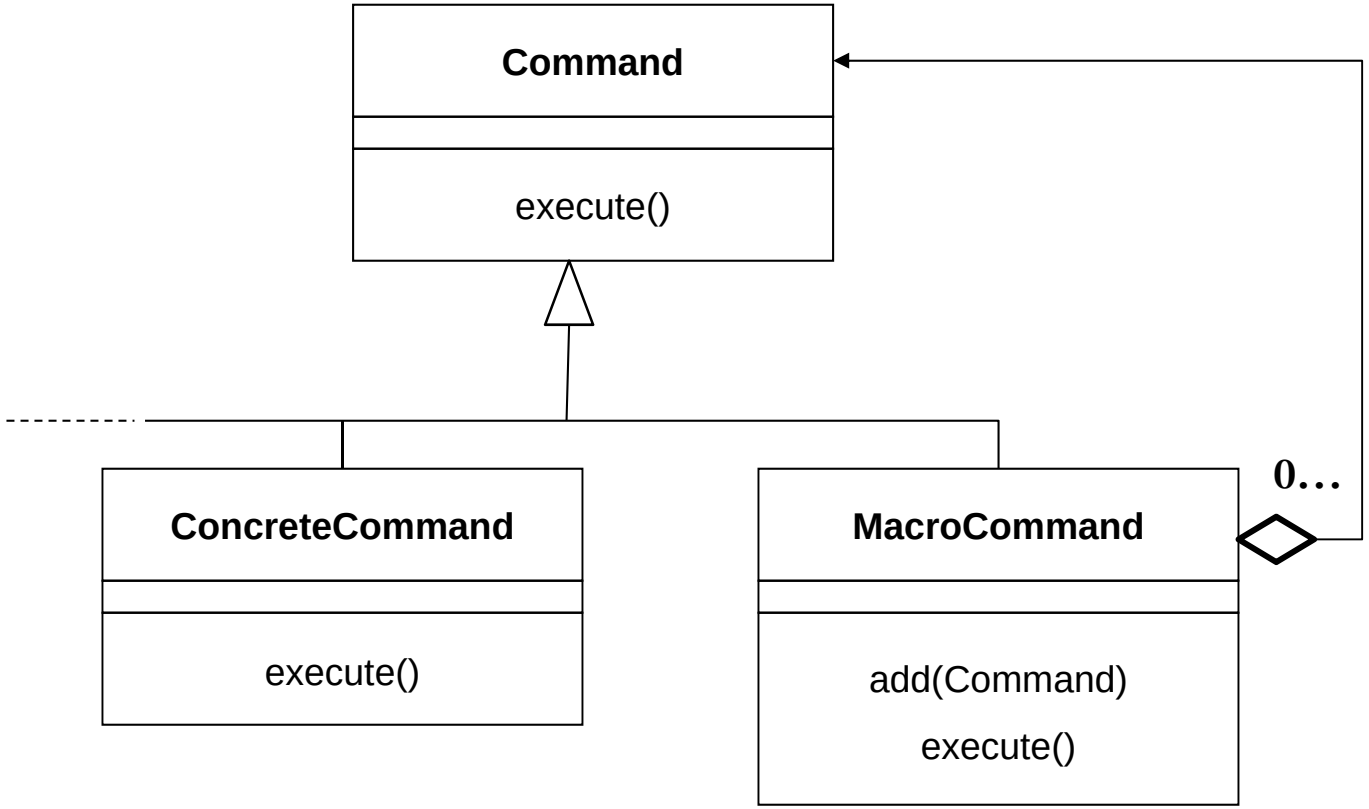
Motivation:

- Support undo redo
- Build macro commands

The code can be cloned from:

git://github.com/odedlac/MacroCommand.git

Command Design-Pattern

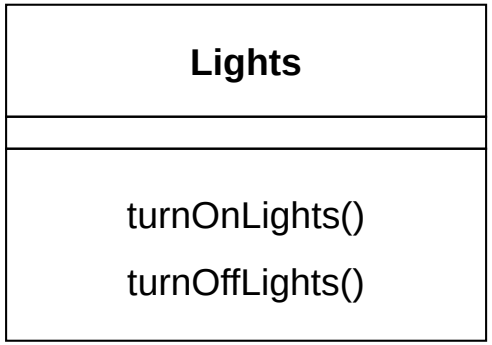


- Each composite has at 1 parent.
- A parent may have many children
- A leaf does not have children

Example: The classes that do the doing

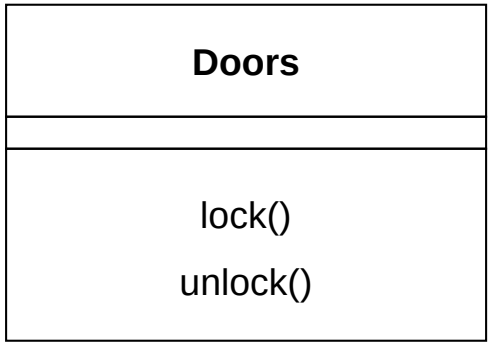
Lights.java

```
public class Lights {
        public void turnOnLights() {
                System.out.println("Lights Turned on");
        }
        public void turnOffLights() {
                System.out.println("Lights Turned off");
        }
}
```



Doors.java

```
public class Doors {
        public void lock() {
                System.out.println("Doors Locked");
        }
        public void unlock() {
                System.out.println("DoorsUnlocked");
        }
}
```



Example: The Command Interface and a concrete command

Command.java

```
public interface Command {
          void execute();
}
```

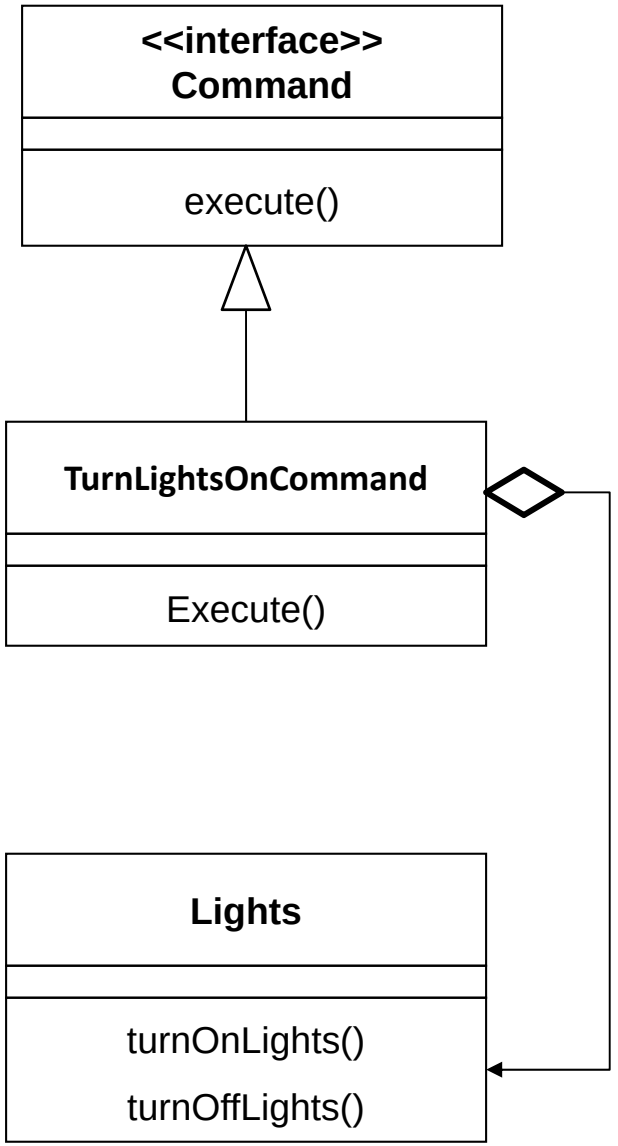
TurnLightsOnCommand.java

```
public class TurnLightsOnCommand implements Command {

          private Lights lights;

          public TurnLightsOnCommand(Lights lights) {
                    this.lights = lights;
          }

          @Override
          public void execute() {
                    lights.turnOnLights();
          }
}
```



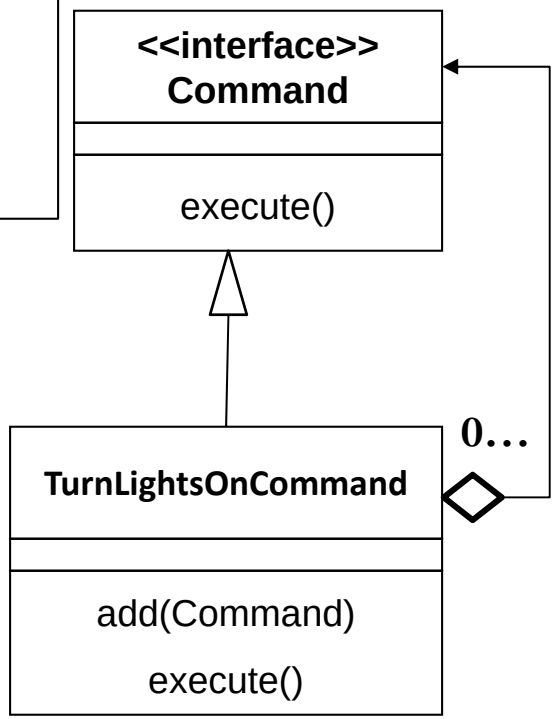
Example: The MacroCommand

MacroCommand.java

```
public class MacroCommand implements Command {
        private ArrayList<Command> commands = new ArrayList<Command>();

        public void add(Command cmd) {
                this.commands.add(cmd); // optional
        }

        @Override
        public void execute() {
                for(Command cmd : commands){
                        cmd.execute();
                }
        }
}
```



Example: The MacroCommand Test

MacroCommandTest.java

```
@Test
public void test() {
            Lights lights = new Lights();
            Doors doors = new Doors();
            Siren siren = new Siren();

            Command turnLightsOnCommand = new TurnLightsOnCommand(lights);
            Command turnSirenOnCommand = new TurnSirenOnCommand(siren);
            Command turnSirenOffCommand = new TurnSirenOffCommand(siren);
            Command turnLightsOffCommand = new TurnLightsOffCommand(lights);
            Command lockDoorsCommand = new LockDoorsCommand(doors);
            Command unlocklDoorsCommand = new UnlockDoorsCommand(doors);

            MacroCommand alarmDrill = new MacroCommand();

            alarmDrill.add(turnLightsOnCommand);
            alarmDrill.add(turnSirenOnCommand);
            alarmDrill.add(lockDoorsCommand);
            alarmDrill.add(unlocklDoorsCommand);
            alarmDrill.add(turnSirenOffCommand);
            alarmDrill.add(turnLightsOffCommand);

            System.out.println("========== MacroCommand Test ========");
            alarmDrill.execute();
}
```

Discussion

Why Use?

- Create macro command
- Decouple requests from their execution
- Support undo and redo

Issues:

- More classes
- Undo Redo mechanisms are harder to debug

Todo:

1. Write a new command and incorporate it in into the current example
2. Add a remove command option to the ‘macro command object’ and write a test that demonstrates it.
3. Use the factory method design-patten in order to create the command objects