Inf2-SEPP: Lecture 13 Part 1: Design Patterns: Command, Singleton

Adriana Sejfia

School of Informatics
University of Edinburgh

Previous lecture

- Design patterns
  - Introduction, cautions
  - The MVC Pattern
  - The Observer Pattern (a behavioural pattern)

This lecture

Design patterns continued

- The Command Pattern (a behavioural pattern) and the Singleton Pattern (a creational pattern)
  - The problem
  - Details
  - Benefits
  - Drawbacks

Command pattern: the problem

Context: detailed design

Command pattern: the problem

Context: detailed design

Problems:

1. Parametrising an object (an invoquer) with a command to another (a receiver)

Command pattern: the problem

Context: detailed design

Problems:

1. Parametrising an object (an invoquer) with a command to another (a receiver)
2. (Optional) Objects from different classes being able to do the same command

Command pattern: the problem

Context: detailed design

Problems:

1. Parametrising an object (an invoquer) with a command to another (a receiver)
2. (Optional) Objects from different classes being able to do the same command

E.g. Universal remote control being programmable to turn on and off various items in your home like lights, stereo, AC etc. It should be easy to change button and dial controls, and to set buttons and dials to do the same thing.

Command pattern: the problem

Naïve solutions for problem 1):

- Adding to the invoquer's implementation long lists of if-else statements standing for all possible commands, what to do, and for what receiver.
  E.g. in a Button class: "if required to turn on AC, tell AC object . . . , if required to turn off lights tell Light object".

Command pattern: the problem

Naïve solutions for problem 1):

- Adding to the invoquer's implementation long lists of if-else statements standing for all possible commands, what to do, and for what receiver.
  E.g. in a Button class: "if required to turn on AC, tell AC object . . . , if required to turn off lights tell Light object".
- Subclassing the invoquer for its use for different commands (to different receivers), interchangeable at runtime.
  E.g. ButtonACOn, ButtonLightsOff inheriting from Button

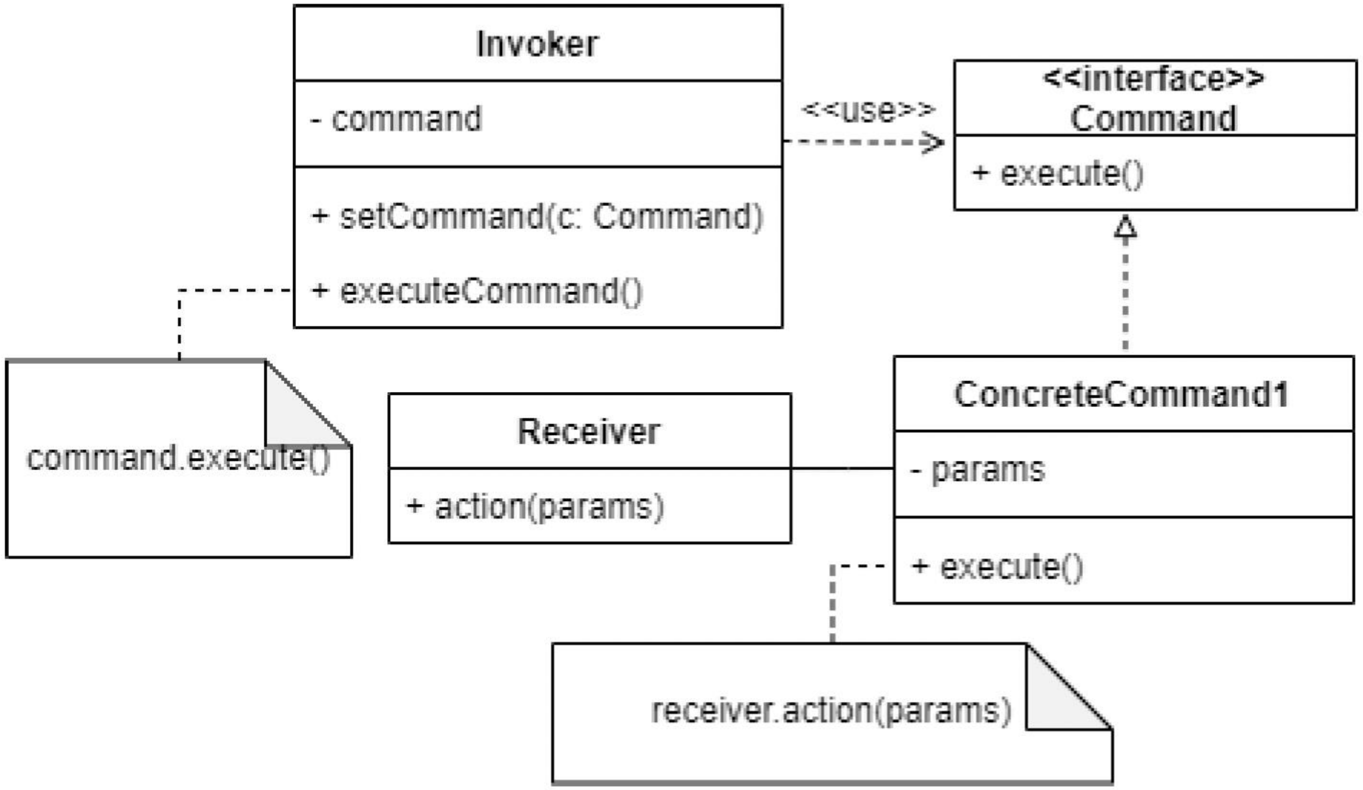
Command pattern: the problem

Naïve solutions for problem 1):

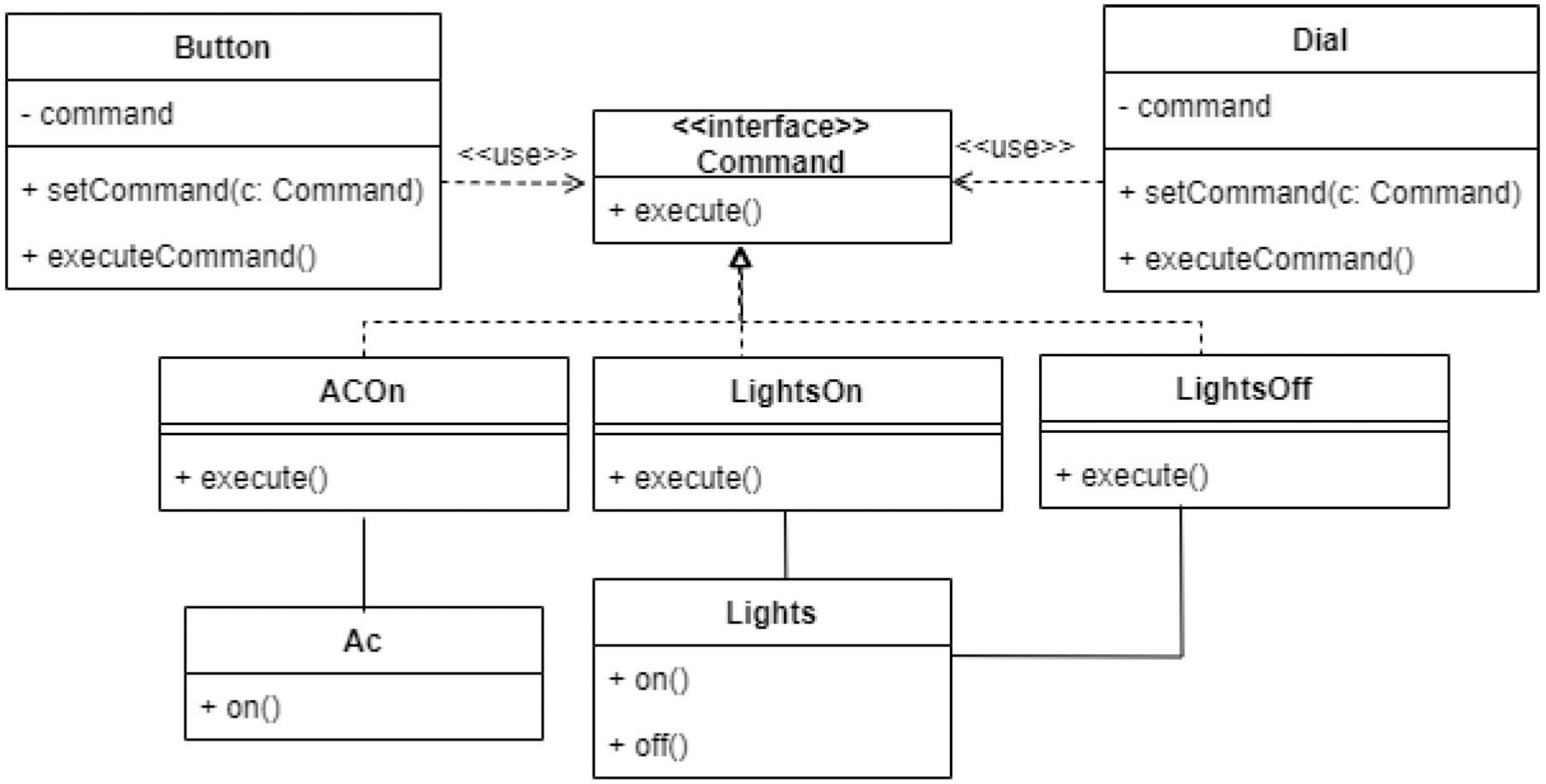
- Adding to the invoquer's implementation long lists of if-else statements standing for all possible commands, what to do, and for what receiver.
  E.g. in a Button class: "if required to turn on AC, tell AC object . . . , if required to turn off lights tell Light object".
- Subclassing the invoquer for its use for different commands (to different receivers), interchangeable at runtime.
  E.g. ButtonACOn, ButtonLightsOff inheriting from Button

Difficult to understand (1), maintain (1, 2 if many subclasses), error prone (1, 2 especially if updating superclass); Invoquers are tightly coupled with receivers; problem 2 would only be solvable with code duplication or adding class dependencies.

Command pattern: general solution



Command pattern: solution to the example



Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver

Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver
- Commands are objects so can be manipulated and extended like any object

Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver
- Commands are objects so can be manipulated and extended like any object
- Composite commands can be set up

Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver
- Commands are objects so can be manipulated and extended like any object
- Composite commands can be set up
- Commands can be queued

Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver
- Commands are objects so can be manipulated and extended like any object
- Composite commands can be set up
- Commands can be queued
- Undo/redo and logging can be set up

Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver
- Commands are objects so can be manipulated and extended like any object
- Composite commands can be set up
- Commands can be queued
- Undo/redo and logging can be set up
- Changing command in invoquer is easy

Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver
- Commands are objects so can be manipulated and extended like any object
- Composite commands can be set up
- Commands can be queued
- Undo/redo and logging can be set up
- Changing command in invoquer is easy
- Several invoquers can use the same commands without code duplication

Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver
- Commands are objects so can be manipulated and extended like any object
- Composite commands can be set up
- Commands can be queued
- Undo/redo and logging can be set up
- Changing command in invoquer is easy
- Several invoquers can use the same commands without code duplication
- Extensible as adding new commands is easy

Command pattern: advantages and disadvantages

Advantages:

- Reduced coupling between invoquer and receiver
- Commands are objects so can be manipulated and extended like any object
- Composite commands can be set up
- Commands can be queued
- Undo/redo and logging can be set up
- Changing command in invoquer is easy
- Several invoquers can use the same commands without code duplication
- Extensible as adding new commands is easy

Disadvantage: code may become more complex due to extra layer between invoquer and receiver

Singleton Pattern: The problem

Context: detailed design

Singleton Pattern: The problem

Context: detailed design

There are situations in which we want for a class to:

1. Have a single instance

Singleton Pattern: The problem

Context: detailed design

There are situations in which we want for a class to:

1. Have a single instance
2. Offer global access to it

Singleton Pattern: The problem

Context: detailed design

There are situations in which we want for a class to:

1. Have a single instance
2. Offer global access to it
3. Protect it from being overwritten

E.g. A single log of all all actions taken by all the entities in the system.

Singleton Pattern: The problem

Naïve partial solution to problem 2 in other programming languages than Java: global variables can make objects accessible

Singleton Pattern: The problem

Naïve partial solution to problem 2 in other programming languages than Java: global variables can make objects accessible

BUT:

- You could still instantiate several such objects (breaks problem 1)

Singleton Pattern: The problem

Naïve partial solution to problem 2 in other programming languages than Java: global variables can make objects accessible

BUT:

- You could still instantiate several such objects (breaks problem 1)
- They can be overwritten, so very unsafe (breaks problem 3)

Singleton Pattern: The problem

Naïve partial solution to problem 2 in other programming languages than Java: global variables can make objects accessible

BUT:

- You could still instantiate several such objects (breaks problem 1)
- They can be overwritten, so very unsafe (breaks problem 3)

The Singleton Pattern is used to address this problem. It is often used for logging, driver objects, caching, and in many other patterns.

Singleton Pattern: Details

There are many versions of Singleton, but they all share the following main ideas:

Singleton Pattern: Details

There are many versions of Singleton, but they all share the following main ideas:

- Make the class itself responsible to keep track of its sole instance, by hiding its constructor (using private in Java)

Singleton Pattern: Details

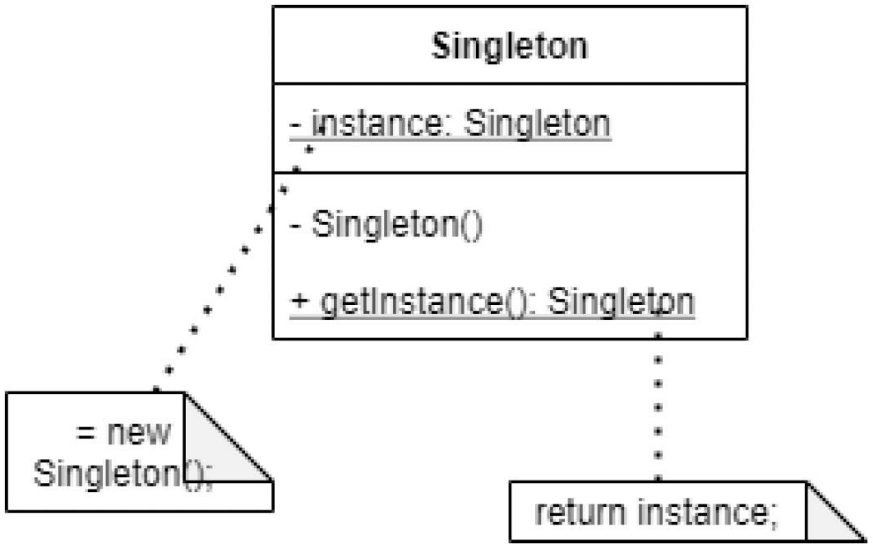
There are many versions of Singleton, but they all share the following main ideas:

- Make the class itself responsible to keep track of its sole instance, by hiding its constructor (using private in Java)

- The class offers a way to access the instance, through a static operation (getInstance()) which returns the sole instance of the class

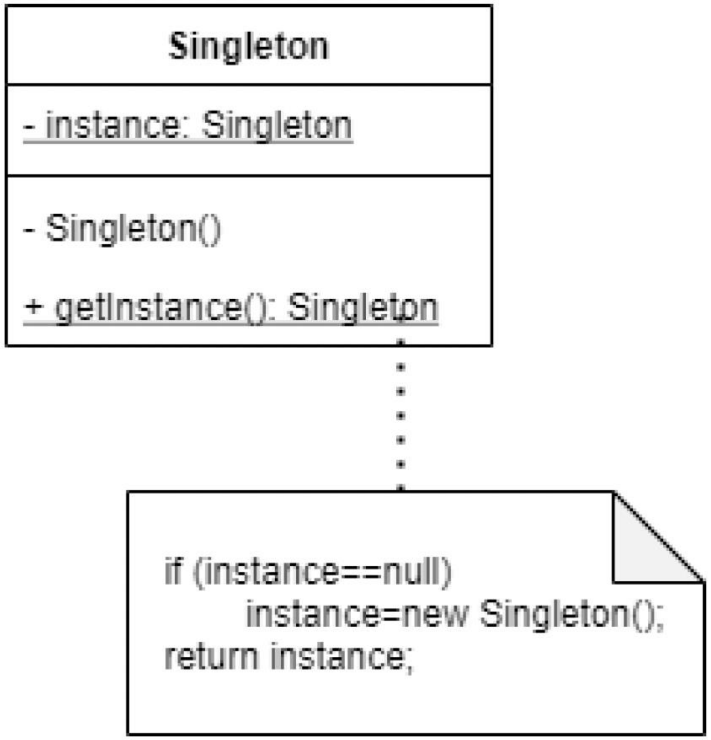
Singleton Pattern: Eager initialisation

The instance is created at first loading of the class (even if not needed).



Singleton Pattern: Lazy initialisation (for single threaded systems)

The instance is created the first time the global creation operation is called.



```
if (instance==null)
        instance=new Singleton();
return instance;
```

Singleton Pattern: Advantages and disadvantages

Advantages:

- Offers controlled global access to a sole instance of a class
- The object is initialised only once
- Preferred over global variables: avoids polluting the name space, permits lazy allocation and initialisation
- Can be easily changed to allow more instances of the class, by editing the getInstance() operation

Singleton Pattern: Advantages and disadvantages

Advantages:

- Offers controlled global access to a sole instance of a class
- The object is initialised only once
- Preferred over global variables: avoids polluting the name space, permits lazy allocation and initialisation
- Can be easily changed to allow more instances of the class, by editing the getInstance() operation

Disadvantages (leading some to frame it as an anti-pattern):

- It is frequently misused, adding unnecessary restrictions
- Introduces global state, potentially unsafe
- Leads to tight coupling between classes in your application
- Multiple threads may create multiple objects
- Its private constructor and static operation make it difficult to produce mock objects needed for unit testing

Resources

- **Essential: Read about the Command and Singleton patterns:**
- If you can get a copy of Gamma, E., 1995. Design patterns: elements of reusable object-oriented software. Pearson Education India: p. 263-268 "Command", p. 144-146 "Singleton"
- On the Command pattern: from Refactoring Guru and YouTube
- On the Singleton pattern: from Refactoring Guru and Wikipedia