2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Miscellaneous Remarks
- Menus and Graphics in a Java GUI
- Command design pattern

Miscellaneous Remarks

- PDF of *Programming in the Large with Design Patterns*
- Review questions on author's book website
- YouTube movie of *The Power of Abstraction* by Barbara Liskov

  Turing Award 2008, Abstract Data Type
- Binary Puzzle Assistant final deadline: see peach^3

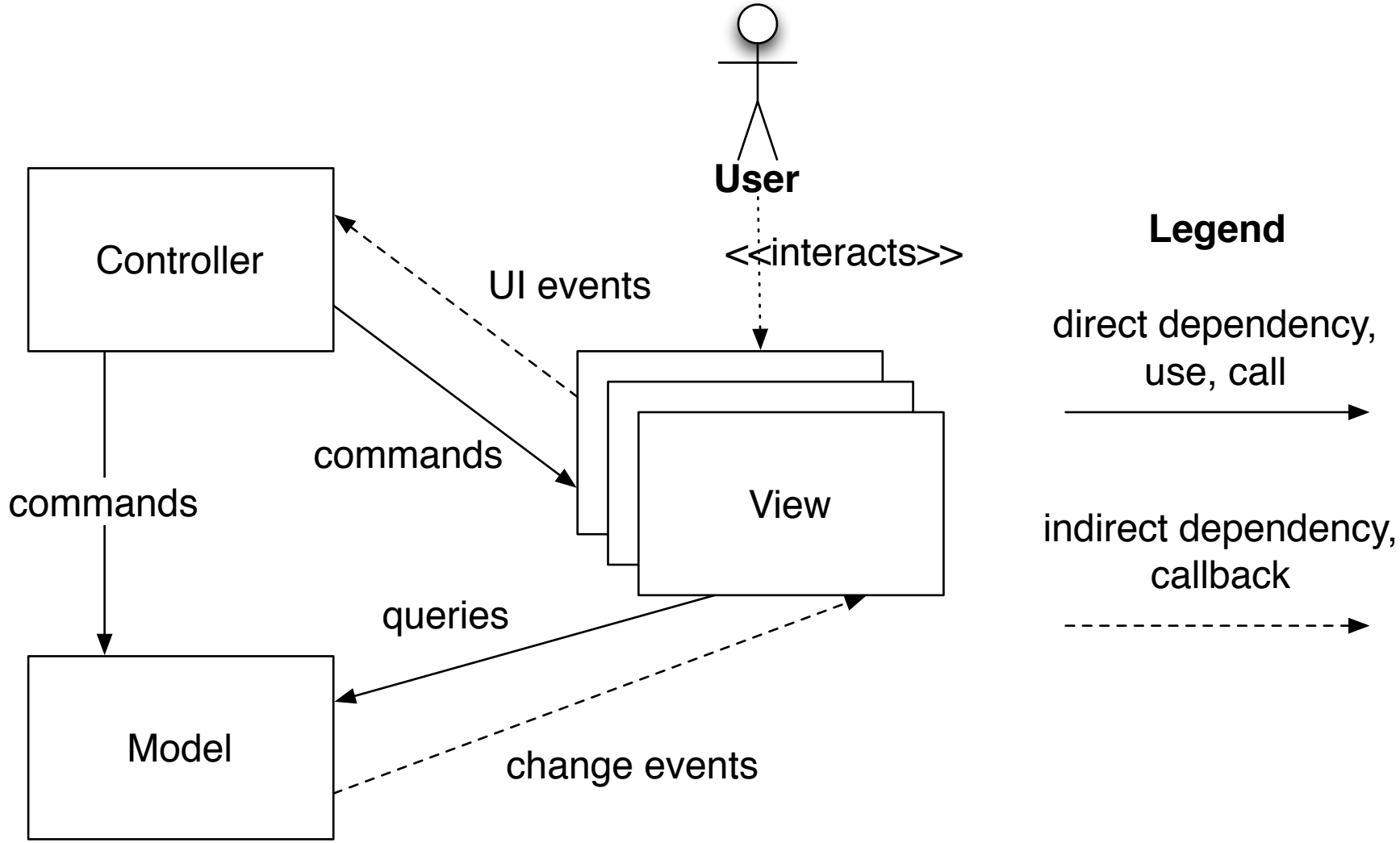
Binary Puzzle Assistant: Submit to Peach

- Every week, submit your work to peach
- Submit your *entire* project in zip archive, including test cases
- Submit what you have done; it must compile (incl. tests)
- Include status information in top-level `package-info.java`

  Use `TODO:` and `FIXME:` flags in comments

  To view: Window > Action Items

Model–View–Controller Architecture



Menus and File Choosers

- See tutorial under Downloads
- For common aspects of buttons and menu items, see Eck Ch.13.3

  `Action` and `AbstractAction`

Graphics in Java using Swing

- A component paints itself via its `paintComponent(Graphics g)`.

  This method is called automatically whenever "needed".
  (It is called by `paint()`, which itself is a callback method.)
  Via `repaint()`, the application can request a `paint()` call.
  There is no guarantee when this is request will be honored.
  Multiple `repaint` requests may get merged into one update.

- Override `paintComponent()` to replace the default paint behavior.

  The new definition must first call **`super`**`.paintComponent(g)`.
  `Graphics g` is used for drawing: `g.drawXXX(...)`.

- `JFrame` uses `paint()` instead of `paintComponent()`.

  However, it is not recommended to override its `paint()` method.
  Instead, draw on a component (like a `JPanel`) inside the frame.

Graphics in Java using Swing

There are various ways to show application-specific graphics.

1. `paintComponent()` redraws the graphics "from scratch".

   con Everything must be redrawn, in every call. Can be slow.
   Needs to store data to redraw the graphics.

   pro No need to "undraw": easy to delete something.

   Terminology: Graphics shows a view of an abstract model.

2. Draw in a buffer and show it by `paintComponent()`.

   con Must set up buffer; must "undraw" to delete something.

   pro No need to redraw everything; only draw the changes.

Example: `GraphicsExample`, takes approach 1

Simple Graphics on a `JPanel` in NetBeans

1. In Design mode, right-click on the `JPanel`, select Customize Code

2. Under **Initialization code** change

default code

```
jPanel1 = new javax.swing.JPanel();
```

custom creation

```
jPanel1 = new javax.swing.JPanel() {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        paintPanel1(this, g);
    }
}
;
```

3. In Source mode, add the following code to the frame:

```
private void paintPanel1(javax.swing.JPanel p, Graphics g) { ... }
```

Complex Graphics on a `JPanel` in NetBeans

1. Create a new `JPanel` descendant via

   File > New File... > Swing GUI Forms > JPanel Form

   name it `XxxPanel` and put it in the `gui` package.

   This adds a `.java` and related `.form` file to the project.
   NetBeans controls the `Generated Code`.

2. In Design mode *of the main frame*, drag-and-drop `XxxPanel` from the NetBeans Navigator onto your `JFrame`.

3. In Source mode *of the panel*, define the data needed to render the view, and override **`void`** `paintComponent()`.

Undo-Redo Facility

- User invokes commands via GUI, changing data in models: Do
- Various degrees of Undo:
  - Undo last (one level of undo)
  - Limited undo (fixed number of levels)
  - Arbitrary undo (limited by memory only)
- Various degrees of Redo:
  - Not available
  - Redo last (one level of redo)
  - Limited redo (fixed number of levels)
  - Arbitrary redo (limited by memory only)

Implement Undo-Redo by Storing Full Model States

- Store full model state before executing an undoable command
- Organize stored states as a stack
- Undo: Pop state from undo stack; restore model to popped state
- Redo: Use a second stack for undone model states
- Model needs ability to clone, and to restore a given state
- Pro: Simple
- Con: Performance loss when state is 'large'

Implement Undo-Redo by Storing State-changing Commands

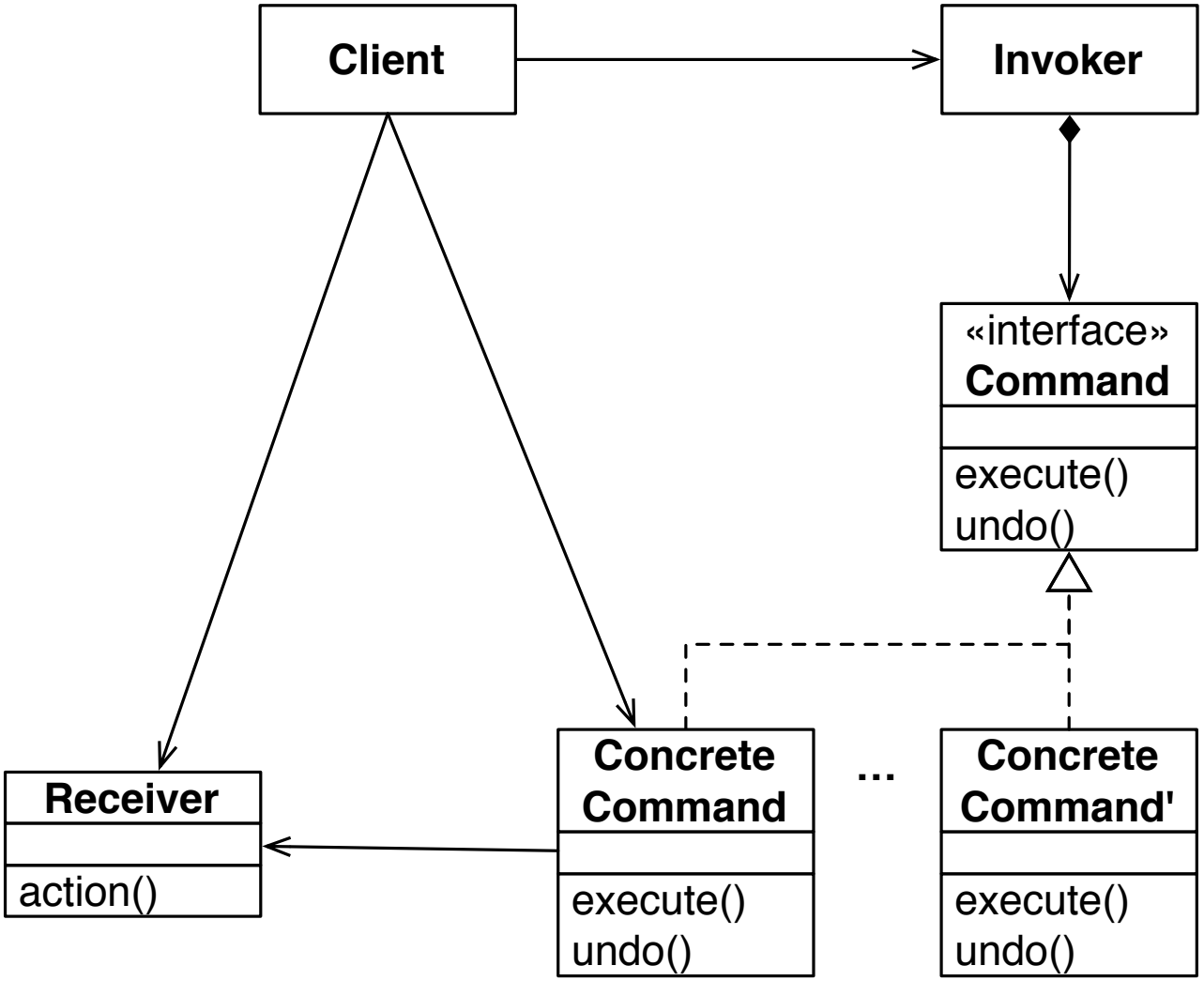
- User invokes commands via GUI, changing data in models: Do.
- These commands can be executed right away, by calling methods.
- It can be useful to have the ability to call these methods later: command queuing, transaction recovery, redo after undo.
- Undo: use a stack of done commands.
- Redo: use a second stack of undone-but-redoable commands.
- Pro: Stores only information to change state ('deltas')
- Con: Needs objects to store commands (Command pattern)

Command Design Pattern

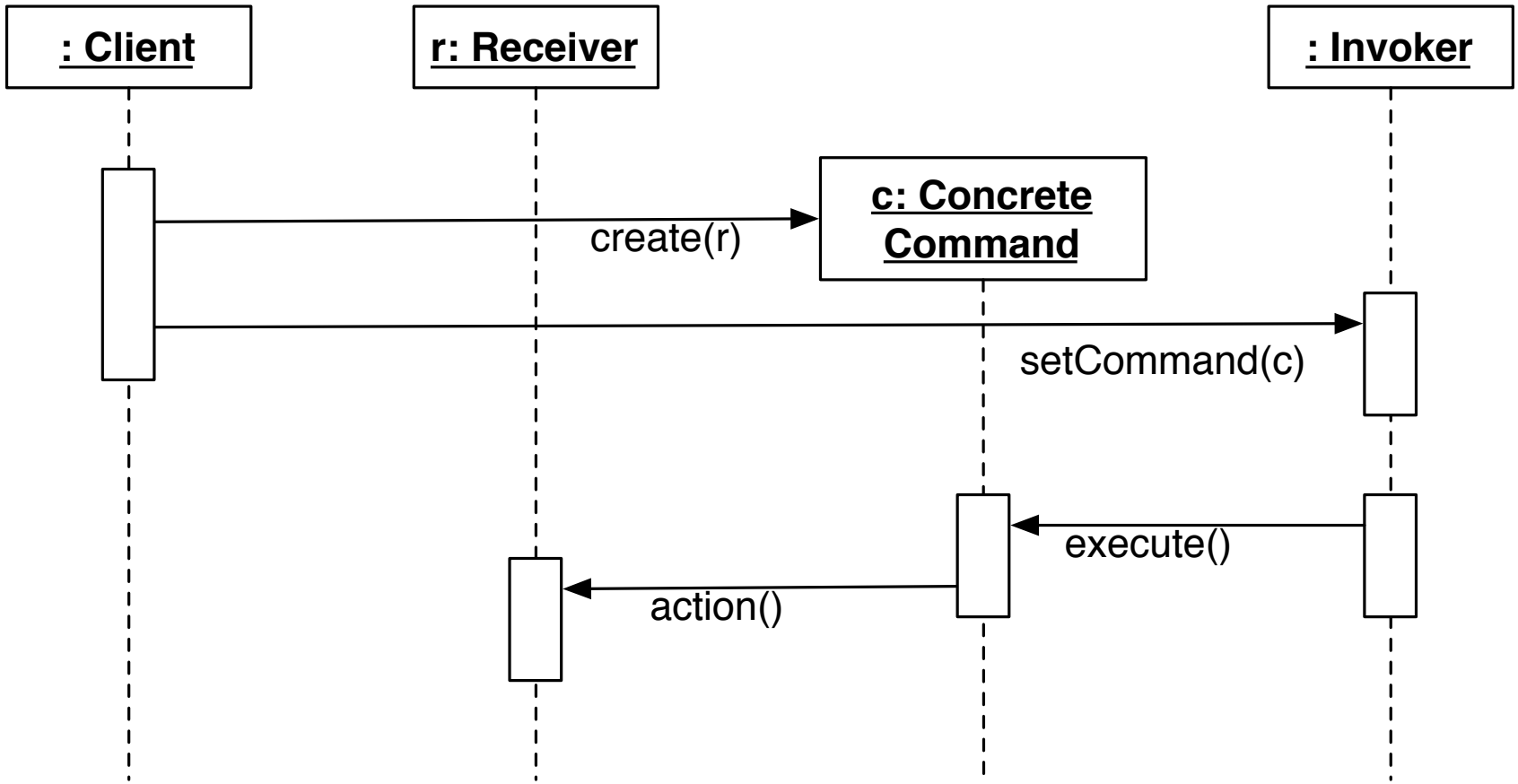
- Command class encapsulates all data to call a method elsewhere:
  - which method to call, including in which object: Receiver
  - which actual parameters to supply as arguments
  - what to do with the return value

  Offers method to execute the call, optionally to undo the call
- A command object can be
  - created by one class: Client
  - stored in another class
  - executed (and optionally undone) in yet another class: Invoker
- Command Design Pattern decouples call data and call moment.

Command Design Pattern: Class Diagram



Command Design Pattern: Sequence Diagram



Undo via Command Pattern

- Provide each concrete `Command` with an `undo` method to revert the effect of `execute`.

  It can be necessary to equip the model with extra operations.

- Introduce a command stack: `Stack<Command> undoStack;`

- The Controller creates `Command` objects, executes them, and pushes them onto `undoStack`.

- Undo is implemented by popping a command from `undoStack`, and invoking its `undo` method.

See: example NetBeans project `CommandPattern`

Redo via Command Pattern

- Introduce another command stack: `Stack<Command> redoStack;`
- The Controller creates `Command` objects, executes them, pushes them on `undoStack`, and clears `redoStack`.
- Undo is implemented by popping a command from the stack, invoking its `undo` method, and pushing it onto `redoStack`.
- Redo is implemented by popping a command from `redoStack`, invoking its `execute` method, and pushing it onto `undoStack`.
- Concern: Ensure precondition of method executed by a command

Encapsulate Undo-Redo Facilities

- In example `CommandPattern`: Undo incorporated into Controller
- Better: Encapsulate Undo-Redo in separate class
- Representation: `Stack<Command> undoStack`, redoStack"
- Preserve rep invariant for `undoStack` and `redoStack`

  Commands on `undoStack` have been executed and can be undone

  Commands on `redoStack` are unexecuted and can be re-executed
- Operations: `did(Command)`, `undo()`, `redo()`, etc.

Compound Commands

- A compound command consists of a sequence of commands.

  Contained commands can be compound: nesting, hierarchy

- It can be defined via the Composite Pattern.

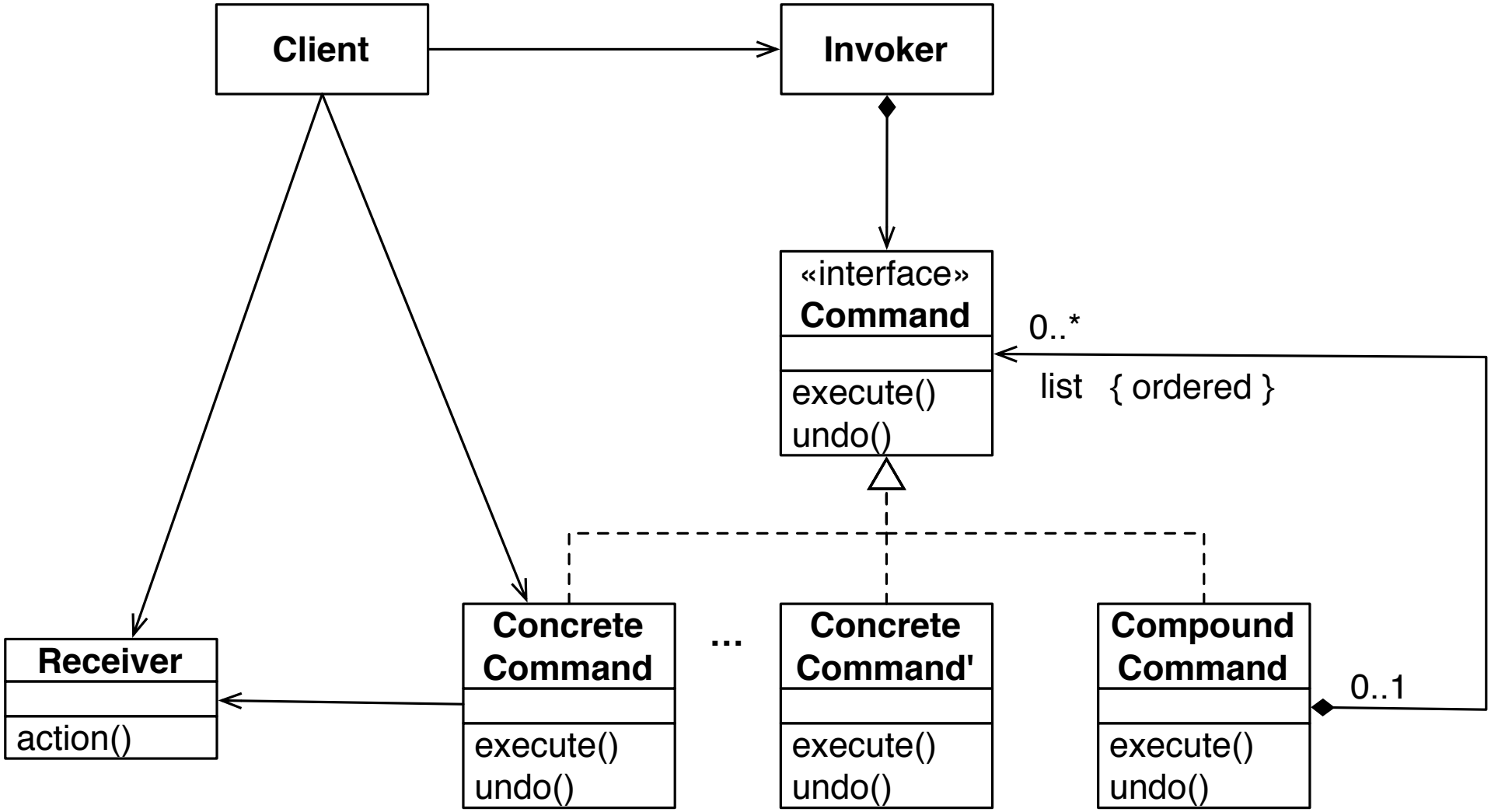
- It does not have a receiver.

- It provides an `add` method to add commands.

- Its `execute` method executes the contained commands, in order.

- Its `undo` method undoes the contained commands, in *reverse* order.

Compound Command Design Pattern: Class Diagram



Summary

- Menus and Graphics in a Java GUI
- Command design pattern, including compound commands