2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Overview

- Testing of `KakuroCombinationGenerator`
- Menus and Graphics in a Java GUI
- Command Design Pattern: *Do*, *Undo*, *Redo*

Questions about Testing `KakuroCombinationGenerator`

1. Why is it not a good idea to include unnecessary test cases?

2. What would be the consequences for testing of `generate(int, int)` in `KakuroCombinationGenerator`, when the postcondition would be weakened by dropping the requirement of *lexicographic order*?

   That is, when method `generate()` would be allowed to generate combinations in any order.

Answers on Testing of `KakuroCombinationGenerator` (1)

Why is it not a good idea to include unnecessary test cases?

- Costs extra test development time.
- Gives false impression of application code quality.
- Gives false impression of test quality.
- Costs extra test time when executing tests.
- Adds to the maintenance burden.

Answers on Testing of `KakuroCombinationGenerator` (2)

What would be the consequences for testing of `generate(int, int)` in `KakuroCombinationGenerator`, when the postcondition would be weakened by dropping the requirement of *lexicographic order*?

Concern	**With lex. order**	**Without lex. order**
No missing	Count, and compare with expected count	
No duplicates	Compare with previous	Store all and compare
Order	Compare with previous	Not a concern
Cost	Keep track of previous	Keep track of all

Answers on Testing of `KakuroCombinationGenerator` (2)

- Checking lexicographic order is easy: one extra variable `preceding` in inner class `Checker`
- Checking for *'no duplicates'* is easy with order:

  Compare to preceding
- Checking for *'no duplicates'* is harder without order:

  Store *all* combinations, and search among stored

Note: Lexicographic order was imposed *on purpose* to ease testing.

– O – O – O – O – O –

NEW TOPIC

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)

Taxonomy of Design Patterns

- Creational patterns

  Factory Method, Singleton

- Structural patterns

  Composite, Façade, Adapter, Decorator

- Behavioral patterns

  Strategy, Iterator, Observer, Command, State, Template Method

- Concurrency patterns

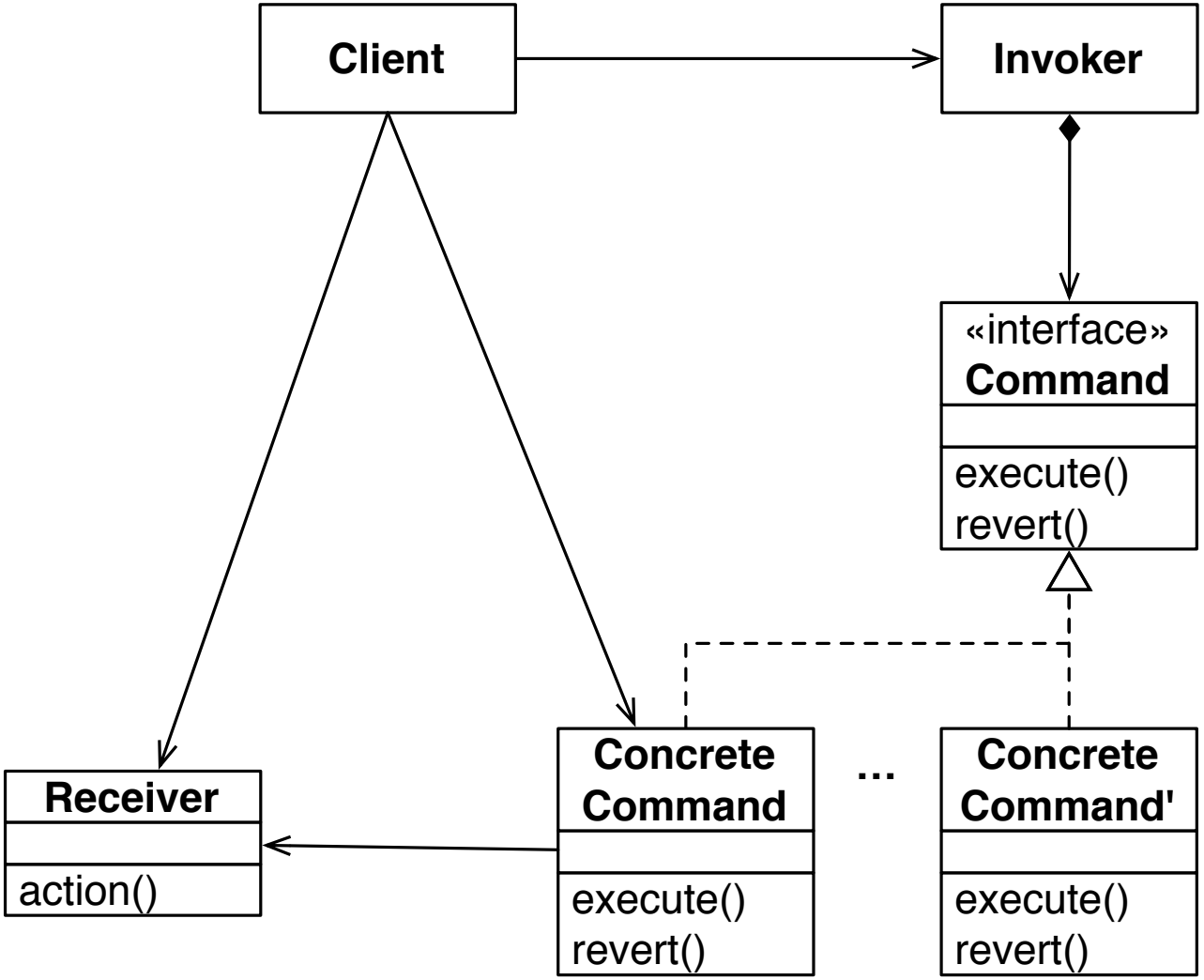
  "`SwingWorker`"

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 revert 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



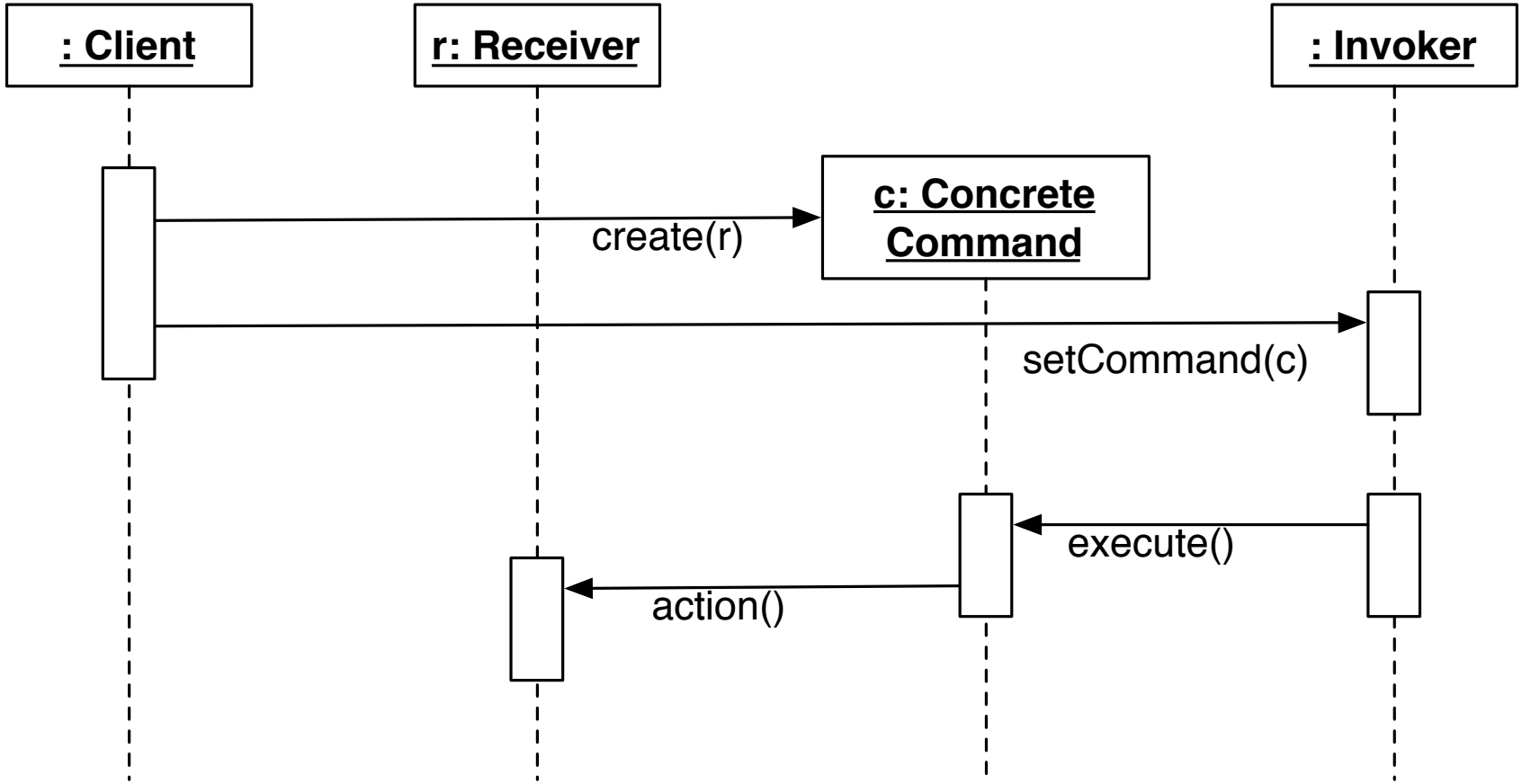
Command Design Pattern: Relation to Strategy Pattern

- Note that a command (interface) can be viewed as a strategy (interface), and concrete commands as concrete strategies.
- Invoker plays role of context in strategy pattern.

  Invoker only knows about abstract command (strategy).
- Client plays client role of strategy pattern.

  Client selects (creates) concrete commands.
- Dependency Inversion: Invoker depends only on abstract command (interface), not on concrete commands

Command Design Pattern: Sequence Diagram



Undo via Command Pattern

- Provide each concrete `Command` with a `revert` 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 `revert` 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 `revert` 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 or reverted by a command

Encapsulate Undo-Redo Facility

- 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.

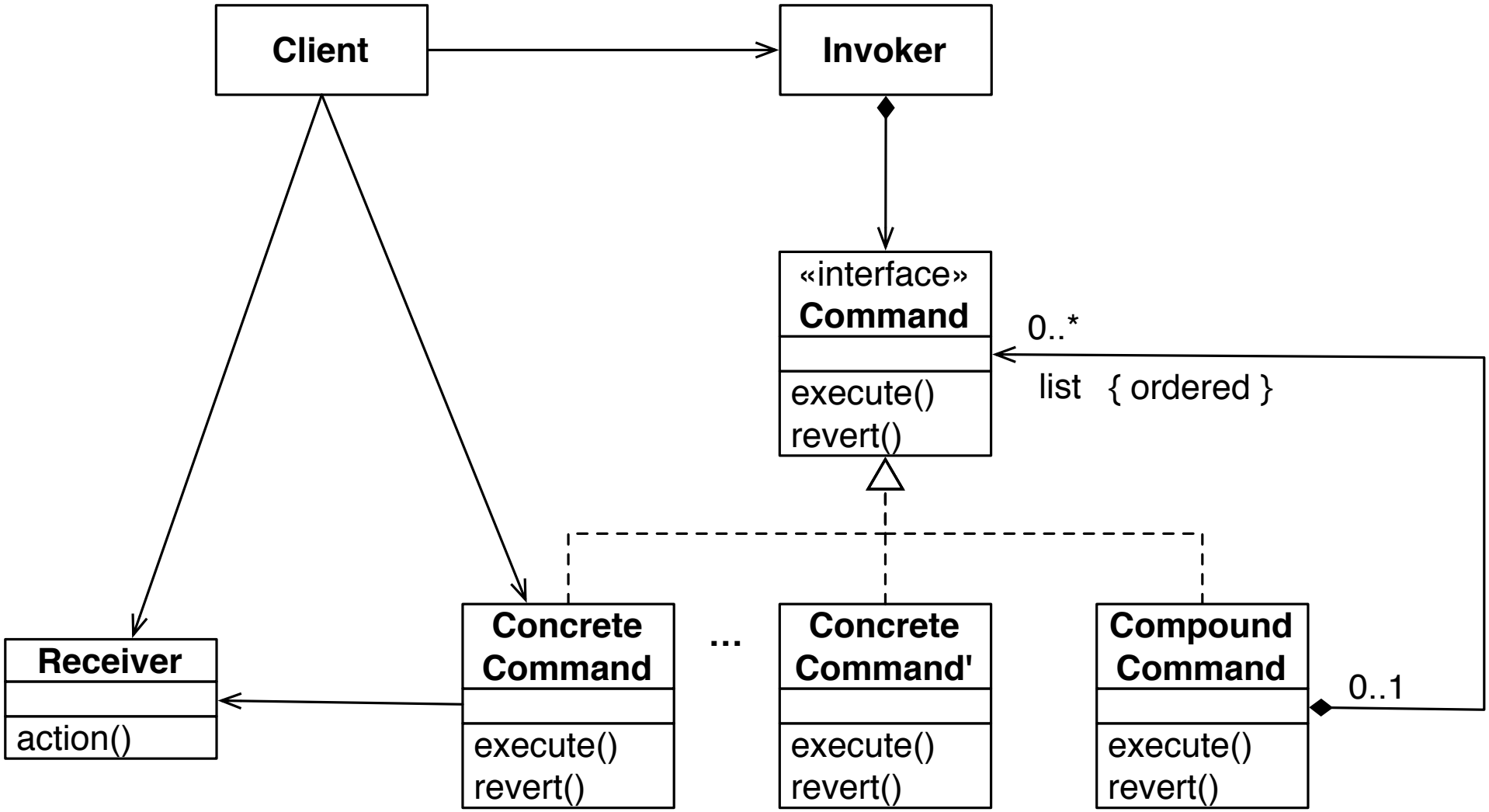
Compound Commands

- 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 need a receiver.
- It provides an `add` method to add commands.
- Its `execute` method executes the contained commands, in order.
- Its `revert` method reverts the contained commands, in *reverse* order.

Compound Command Design Pattern: Class Diagram



Backtracking

- Speculative technique to search through a state space.
- Recursive implementation for Kakuro Puzzle Assistant:
  1. Apply reasoning to find 'forced' digits (compound commmand).
  2. Find an open cell c (if not found, then puzzle is solved).
  3. For each possible cell state s: speculatively, set c to s, and if state still valid, solve remainder recursively.

Recursive Backtracking: Implementation Variations

- Copy entire state (onto the stack) for each recursive call.

  Undo is "automatic" by falling back to earlier stored state.

  Possible disadvantages:

  - Uses more memory if state is large
  - Needs ability to copy and re-instate the full state

- Maintain state in global variable, and modify it: do-undo.

  Here, the Command Pattern can be used. Take this approach!

  Compound commands are useful to deal with 'forced' digits.

Homework Series 6

- Modify *CommandPattern* demo program to include Redo
- Create a stand-alone Undo-Redo facility (Compulsory)
- Modify *Kakuro Puzzle Assistant* to include Undo/Redo, some reasoning strategies for forced moves, and a backtrack solver (Graded)

Summary

- Also see Wikipedia: `en.wikipedia.org/wiki/Command_pattern`
- The Command design pattern provides a way of capturing the data needed to call a method later elsewhere (Do facility).
- Can be extended to offer an Undo facility, optionally with Redo.

  See example NetBeans project `CommandPattern`.
- Backtracking can be based on a Do-Undo mechanism.