Object-Oriented Programming
Command Pattern

CSIE Department, NTUT
Woei-Kae Chen
Command: Intent

- Encapsulate a request as an object
  - thereby letting you parameterize clients with different requests
  - queue or log requests
  - and support undoable operations.

- Also known as
  - Action, Transaction
Command: Motivation (1)

- Issue requests without knowing anything about the operation being requested (including the receiver of the request)
Command: Motivation (2)

Example: PasteCommand

- receiver is the document object (supplied upon instantiation)

```
Document
Open()
Close()
Cut()
Copy()
Paste()

Command
Execute()

PasteCommand
Execute()

Supplied upon instantiation
```
Example: OpenCommand

```cpp
name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()
```
Example: Macro Command

```
for all c in commands
  c->Execute()
```

Composite pattern
Command: Motivation (5)

- Command pattern decouples the object that invokes the operation from the one performing it → flexibility
  - two user interfaces may share an instance of the same concrete Command subclass.
  - commands can be replaced dynamically (for context-sensitive menus).
  - macro commands.
  - all because the command requester only needs to know how to issue it; it doesn't need to know how to perform it.
Use the Command Pattern when you want to:
- parameterize objects (invoker) by an action to perform (commands). Commands are an object-oriented replacement for callbacks.
- specify, queue, and execute requests at different times → a command object can have lifetime independent of the original request.
- support undo: add Unexecute() and store executed commands in a history list.
- support logging: can be reapplied in case of a system crash (add Load() and Store() operations).
- structure a system around high-level operations built on primitive operations (e.g., transactions).
**Command: Structure**

- **Invoker**: Component responsible for executing commands.
- **Command**: Interface that defines the `Execute()` method.
- **ConcreteCommand**: Subclass of `Command` that implements the `Execute()` method.
- **Receiver**: Component that performs the action specified by the command.
- **Client**: Component that interacts with the command via the invoker.

**Interaction diagram**:
- **Client** interacts with **Invoker**.
- **Invoker** interacts with **Command**.
- **Command** interacts with **ConcreteCommand**.
- **ConcreteCommand** interacts with **Receiver**.
- **Receiver** performs **Action**.

**Key points**:
- **Decoupled**: Receiver performs its action without being tied to the invoker.
- **May be Template Method**
- **Create**
  - Receiver performs Action
  - May be Receiver-less

**Actions**:
- **Invoker**: `Execute()`
- **ConcreteCommand**: `Execute()`, `state` (receives from receiver)
- **receiver**: `receiver->Action()`
Command: Participants

- **Command**
  - declare an interface for executing an operation.
- **ConcreteCommand (PasteCommand, etc.)**
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client (Application)**
  - creates a ConcreteCommand object and sets receiver.
- **Invoker (MenuItem)**
  - asks the command to carry out the request.
- **Receiver (Document, Application)**
  - Knows how to perform the operations
Command: Collaboration

Interaction (sequence) diagram

new Command(aReceiver)

StoreCommand(aCommand)

Active

Life time

Action()

Execute()
Command: Consequences

- Command **decouples** the object that invokes the operation from the one that knows how to perform it.
- Commands are **first-class objects**. They can be manipulated and extended like any other object.
- **MacroCommand**: composite commands are an instance of the Composite pattern.
- It’s easy to add **new commands**.
Command: Implementation

● How intelligent should a command be?
  – receiver perform all actions ⇔ receiver-less

● Supporting undo and redo.
  – add Unexecute
  – history list
    ● Template Method (auto store) and Prototype (copy) pattern

● Avoiding error accumulation in the undo process
  – apply Memento pattern to give command access to information without exposing the internals of other objects.

● Using C++ templates
  – for commands that are not undoable and do not require arguments.
A Composite pattern can be used to implement MacroCommands.

A Memento can keep state that the command requires to undo its effect.

A command that must be copied before being placed on the history list act as a Prototype.

Patterns using similar ideas (inheritance and polymorphism)

- Command: command as object
- Strategy: algorithm as object
- Iterator: pointer as object
- State: state as object
- Composite: composite as object (with uniform interface)
- Decorator: decorator as object (with uniform interface)
- Proxy: proxy as object (with uniform interface)