Behavioural Design Patterns

Part 4

Behavioural Design Patterns

- State
- Strategy
- Observer
- Command
- Visitor

Behavioural Patterns: Command

- Simple Example: Restaurant
  - Client of a restaurant wants to request food
  - Cook can only cook (execute) one food request at a time (and may currently be busy)
  - Waiter creates an abstraction of the request (the client's order) and places this request in the cook's queue

Behavioural Patterns: Command

- Suppose we are creating a set of classes to present a text-based menu system to a user

- We want to release the menu system as a library that can be integrated into other products

Behavioural Patterns: Command

Menu.h

```
class Menu
{
  public:

    virtual ~Menu();
    void add(MenuItem* item);
    MenuItem* getChoice();

  private:
    std::vector<MenuItem*> _items;

};
```

Behavioural Patterns: Command

- Problem: how do we design the `MenuItem` class?
- We don’t know in advance what actions might be taken when a menu item is selected
- In other words, we need to issue requests to objects without knowing anything about the operation being requested, or the receiver of the request

Behavioural Patterns: Command

Design Pattern:

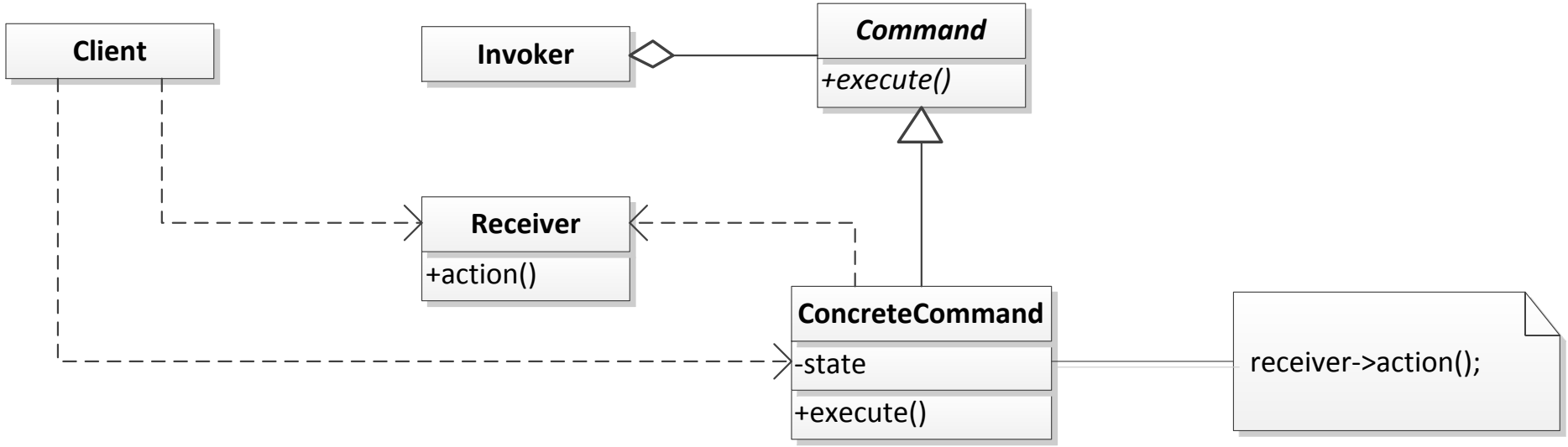
Command

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

Behavioural Patterns: Command

- Applicability:
  - You want to parameterize objects by an action to perform
  - You want to specify, queue, and execute requests at different times
  - You want to support undo operations

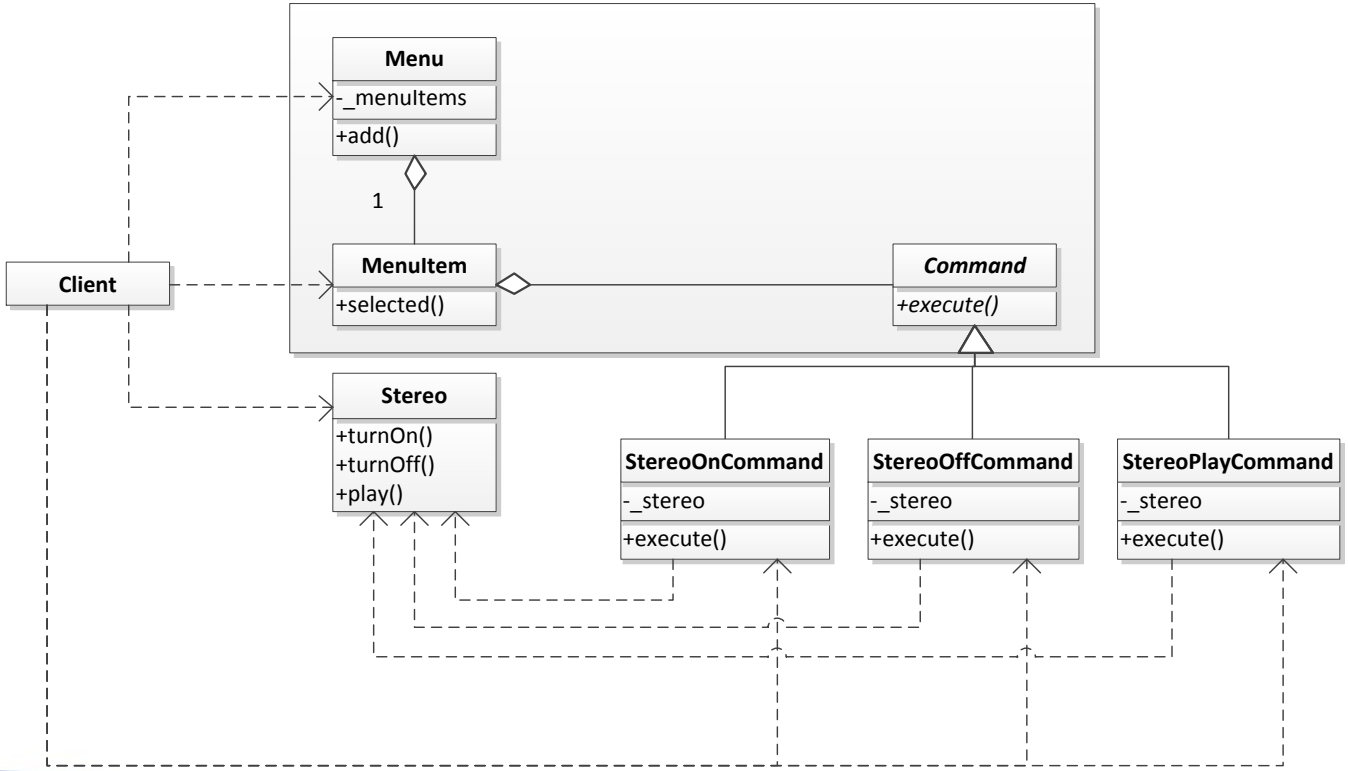
Behavioural Patterns: Command



Behavioural Patterns: Command

- Classes Involved:
  - Client
    - Creates the commands and sets the receiver object
    - Issues these commands to the invoker
  - Invoker
    - Maintains a queue (or stack, log) of commands
    - Execution of commands is done through invoker
  - Receiver
    - Implements any actions that may be executed by commands

Behavioural Patterns: Command



Behavioural Patterns: Command

main.cpp

```
Stereo* s = new Stereo("Living room stereo");

Command* cmd5 = new StereoOnCommand(s);
Command* cmd6 = new StereoOffCommand(s);
Command* cmd7 = new StereoPlayCommand(s);

Menu menu;

menu.add(new MenuItem("Turn on stereo", cmd5));
menu.add(new MenuItem("Turn off stereo", cmd6));
menu.add(new MenuItem("Play stereo", cmd7));

menu.add(new MenuItem("Turn on lamp", cmd1));
```

Behavioural Patterns: Command

- When a menu item is selected in the menu, the corresponding `MenuItem` is `selected()`
- It then invokes the `execute()` function from its associated `Command` object
- Requires dynamic dispatch

Behavioural Patterns: Command

Concrete Command – StereoOffCommand.h

```
void execute()
{
  this->_stereo->turnOff();
}
```

Each `ConcreteCommand`, when `execute()`'d will implement their command by calling functions on the associated receiver

Behavioural Patterns: Command

- Consequences:
  - 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
  - We can assemble commands into a composite command to allow for macro recording and playback
  - It is easy to add new Receivers and/or Commands, because we don’t have to change existing classes
  - We can have multiple receivers

Behavioural Patterns: Command

Modified Client – main.cpp

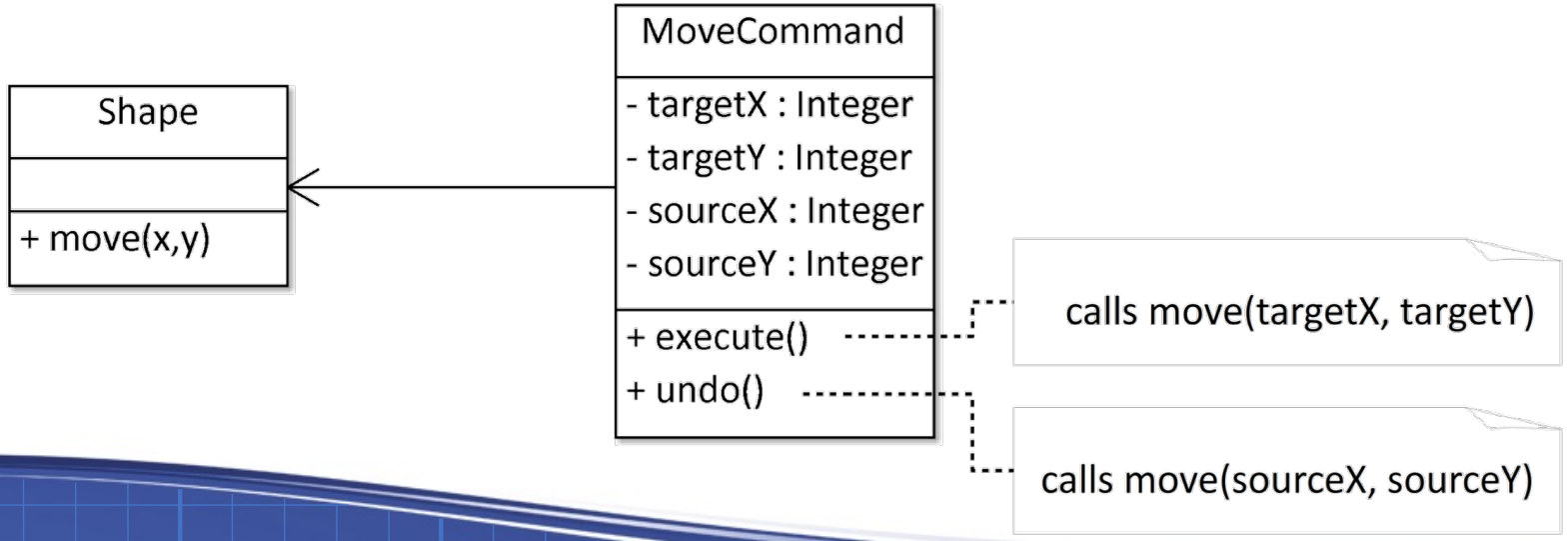
```
Light* l1 = new Light("Lamp");
Light* l2 = new Light("Porch light");

Command* cmd1 = new LightOnCommand(l1);
Command* cmd2 = new LightOffCommand(l1);
Command* cmd3 = new LightOnCommand(l2);
Command* cmd4 = new LightOffCommand(l2);
```

```
menu.add(new MenuItem("Turn off lamp", cmd2));
menu.add(new MenuItem("Turn on porch light", cmd3));
menu.add(new MenuItem("Turn off porch light", cmd4));
```

Behavioural Patterns: Command

- Consequences:
  - We can easily support rollback/undo operations by adding an `unexecute` or `undo` method



Behavioural Patterns: Command

- Consequences:
  - We can easily queue up commands to be executed as a batch
  - We can easily provide progress on a set of commands being executed