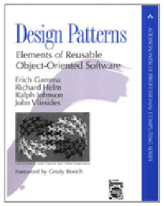


Design Pattern Definitions from the GoF Book

The Command Pattern

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

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

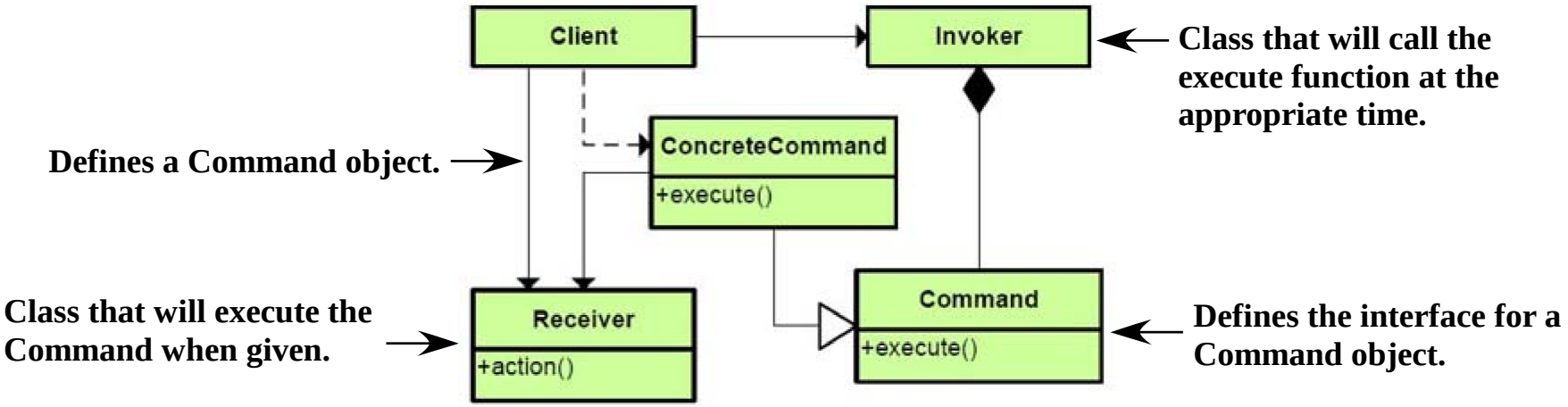
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

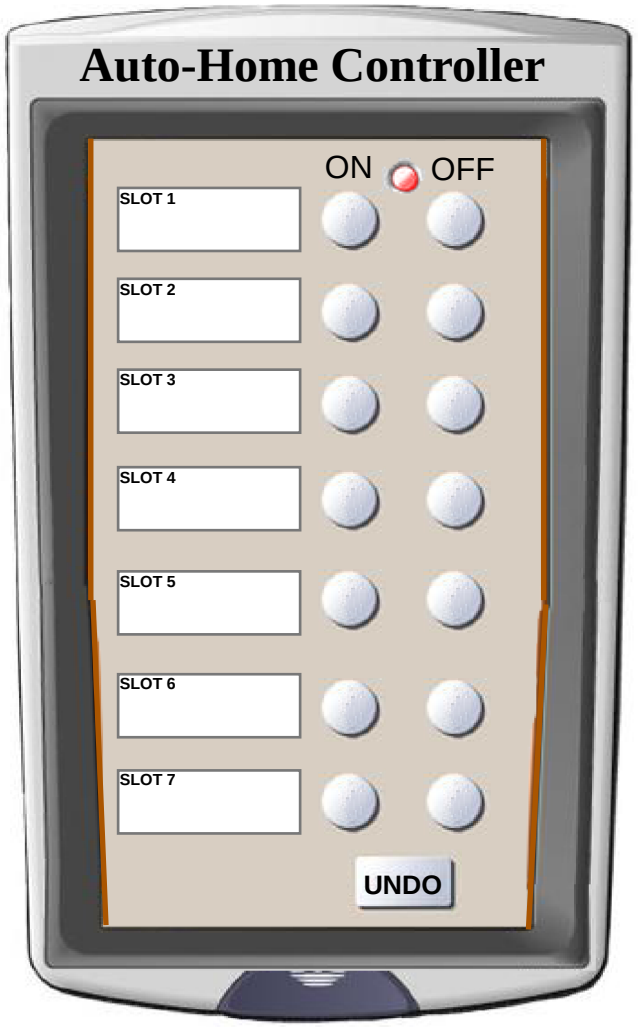
Design Patterns: The Command
Quick Overview

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

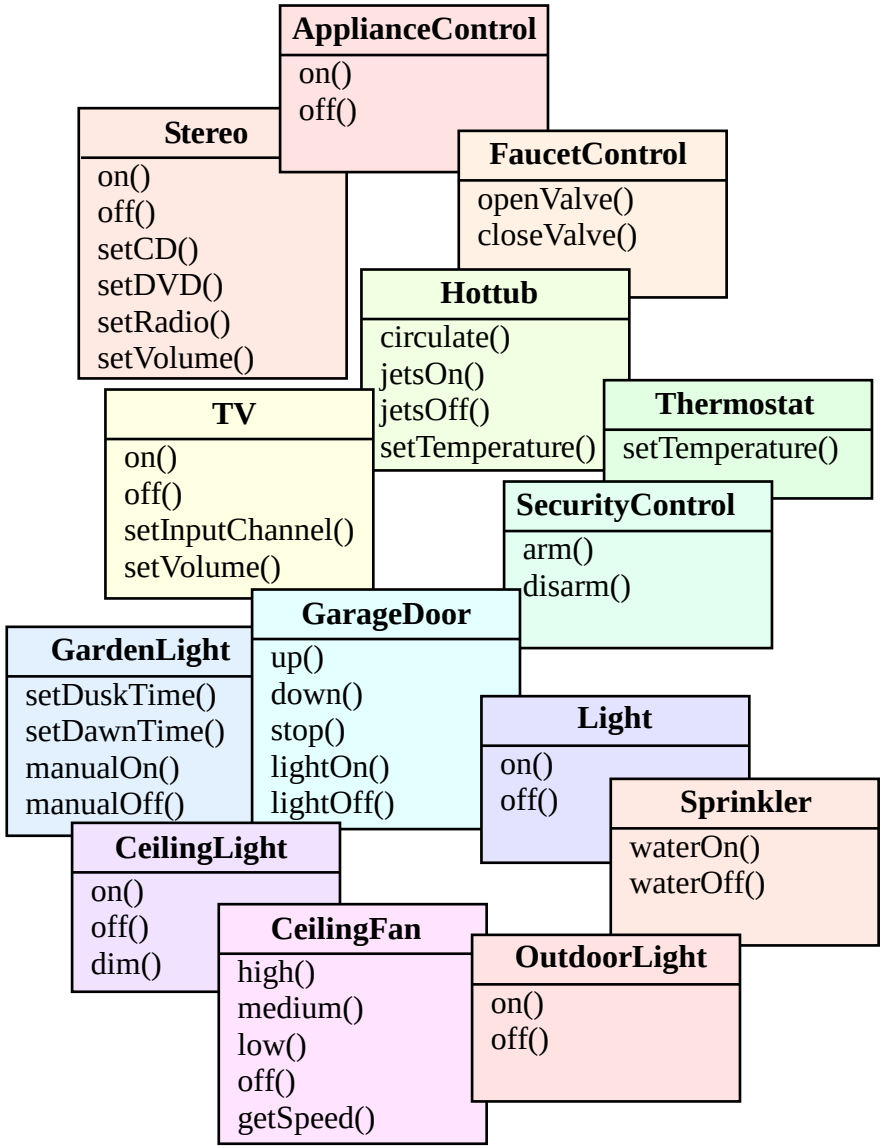


Design Patterns: The Command

We need to program this...



...to interface with and control any of these.



Design Patterns: The Command

How would you design it?

Some starting points to consider

- *None of the different devices have a common interface.*
- *There will probably be many more such devices added in the future, i.e. we can expect the system to CHANGE.*
- *The remote control should know how to interpret button presses and make requests, but it shouldn't know any of the details about how a device works.*
- *We don't want the remote to consist of a long series of* **if** *statements, like*

```
if slot1==Light then light.on()
else if slot1==Hottub then hottub.jetsOn()
```

Just a hint...

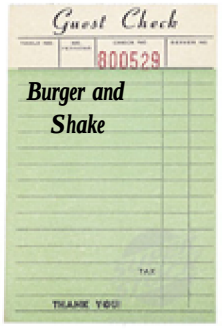
The Command Pattern allows you to decouple the requester of an action from the object that actually performs the action.

Now think about that for a minute...

Design Patterns: The Command

You go into the diner and your order is taken by Flo.

Earl, the short-order cook picks up the order and prepares your meal then calls "Order up!"



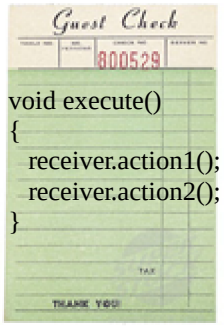
Who writes down your order, and takes it to the order counter.

Design Patterns: The Command

From the Diner to the Command Pattern

Invoker

Receiver



Command

Client

The client creates a command object (your order) and sets its Receiver (Earl)

createCommandObject(receiver)

The client gives the command to the Invoker (Flo)

setCommand(command)

The Receiver (Earl) executes the command.

action1()
action2()

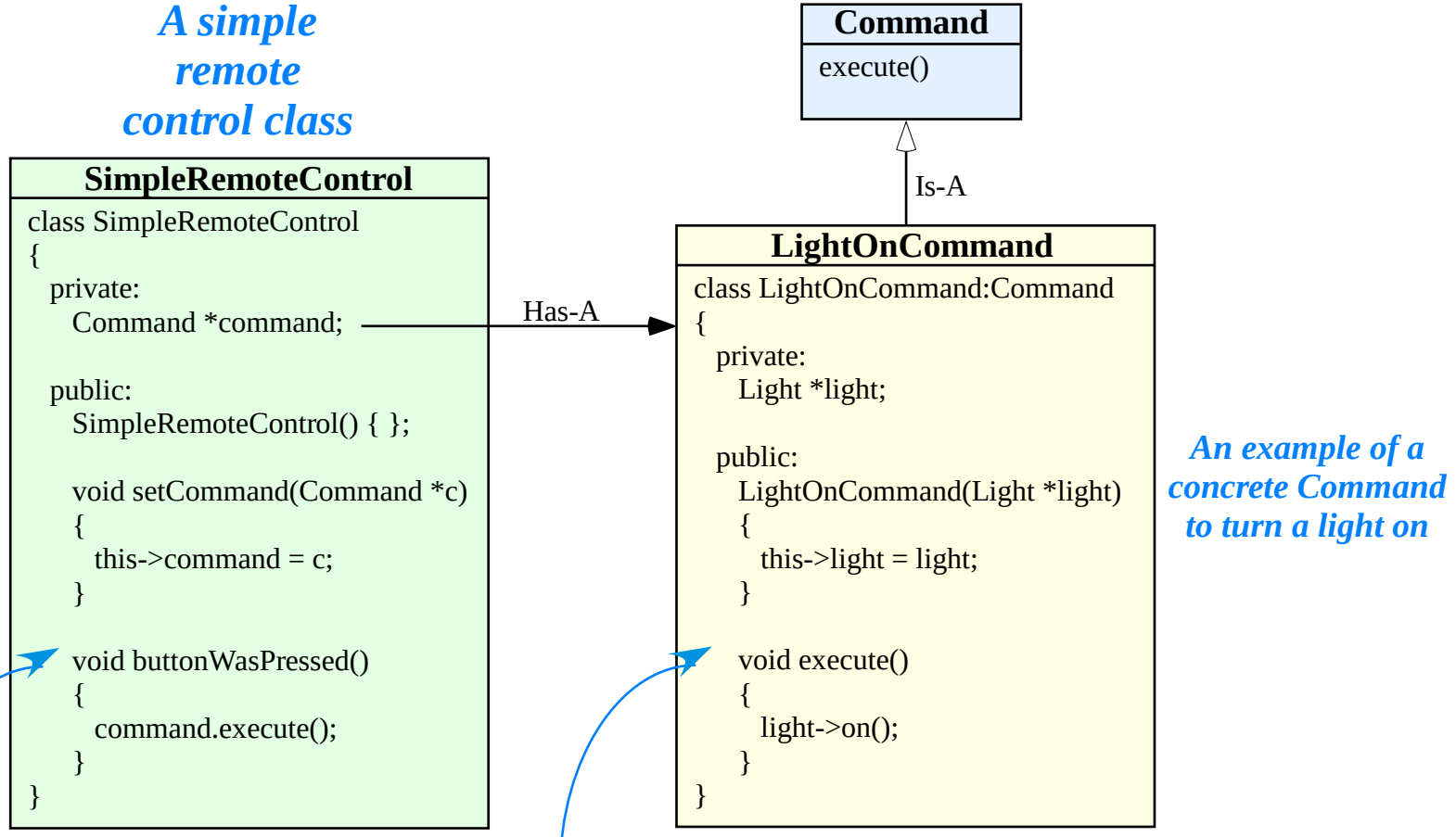
The Invoker (Flo) calls the execute command at the appropriate time.

command.execute()

Design Patterns: The Command

The Command interface

A simple remote control class

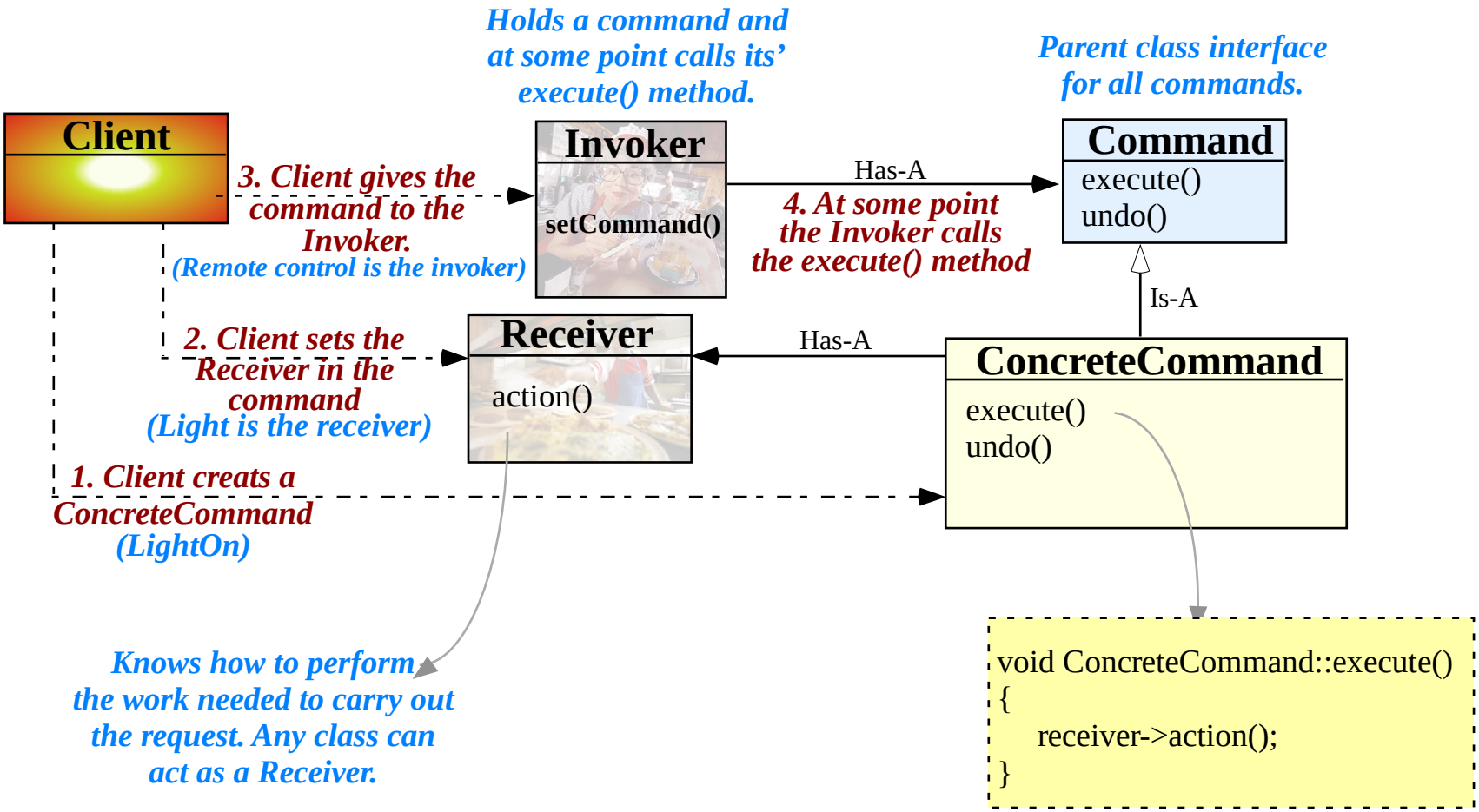


An example of a concrete Command to turn a light on

When the button is pressed this method is called.

Which in turn calls this method

Design Patterns: The Command



Design Patterns: The Command

Implementing the Remote Control

```
class RemoteControl
{
    private:
        Command *onCommands[7];
        Command *offCommands[7];
        Command *undoCommand;
    public:
        RemoteControl();
        void setCommand(int slot,
                Command *onCommand,
                Command *offCommand);
        void onButtonPushed(int slot);
        void offButtonPushed(int slot);
}
```

?1

?1 *You'll see what this command is shortly.*

?2 *So we don't have to keep checking to see if there really is a command here.*

NoCommand

execute() {}; // Do nothing
undo(){}; // Do nothing

```
RemoteControl::RemoteControl()
{
    for(int i=0; i<7; i++)                    ?2
    {
        onCommands[i] = new NoCommand();
        offCommands[i] = new NoCommand();
    }
    undoCommand = new NoCommand();
}
void RemoteControl::setCommand(int slot,
                Command *onCommand,
                Command *offCommand)
{
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}
void RemoteControl::onButtonPushed(int slot)
{
    onCommands[slot]->execute();
    undoCommand = onCommands[slot];
}
 void RemoteControl::offButtonPushed(int slot)
{
    offCommands[slot]->execute();
    undoCommand = offCommands[slot];
}
```

Starting to get an idea?

Design Patterns: The Command

Implementing the Commands and Undo

LightOnCommand

```
class LightOnCommand:Command
{
  private:
    Light *light;
  public:
    LightOnCommand(Light *light)
    {
      this->light = light;
    }
    void execute()
    {
      light->on();
    }
    void undo()
    {
      light->off();
    }
}
```

LightOffCommand

```
class LightOffCommand:Command
{
  private:
    Light *light;
  public:
    LightOffCommand(Light *light)
    {
      this->light = light;
    }
    void execute()
    {
      light->off();
    }
    void undo()
    {
      light->on();
    }
}
```

StereoOnWithCDCommand

```
class StereoOnCommand:Command
{
  private:
    Stereo *stereo;
  public:
    StereoOnCommand(Stereo *stereo)
    {
      this->stereo = stereo;
    }
    void execute()
    {
      stereo->on();
      stereo->setCD();
      stereo->setVolume(11);
    }
    void undo()
    {
      stereo->off();
    }
}
```

...and so on.

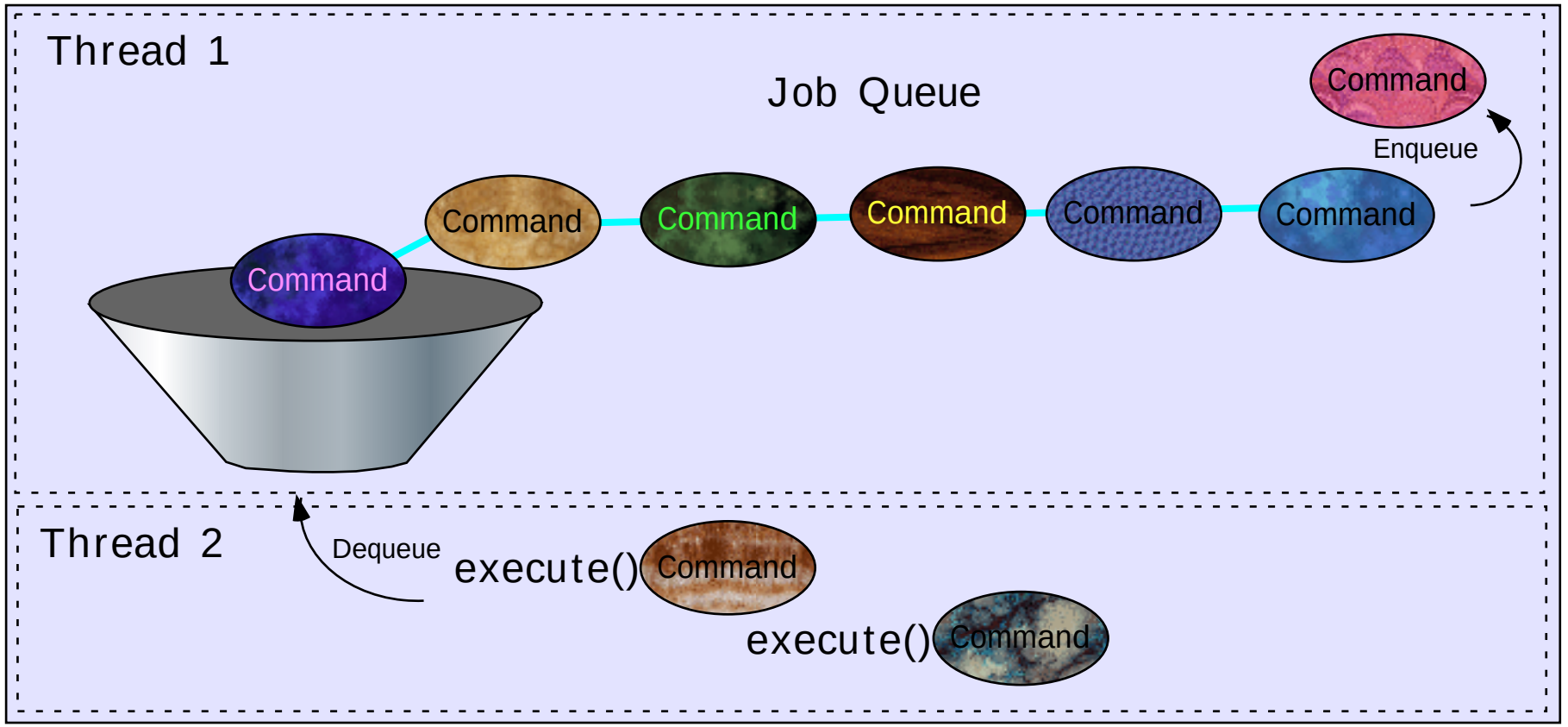
Notice we have now added an undo() function.

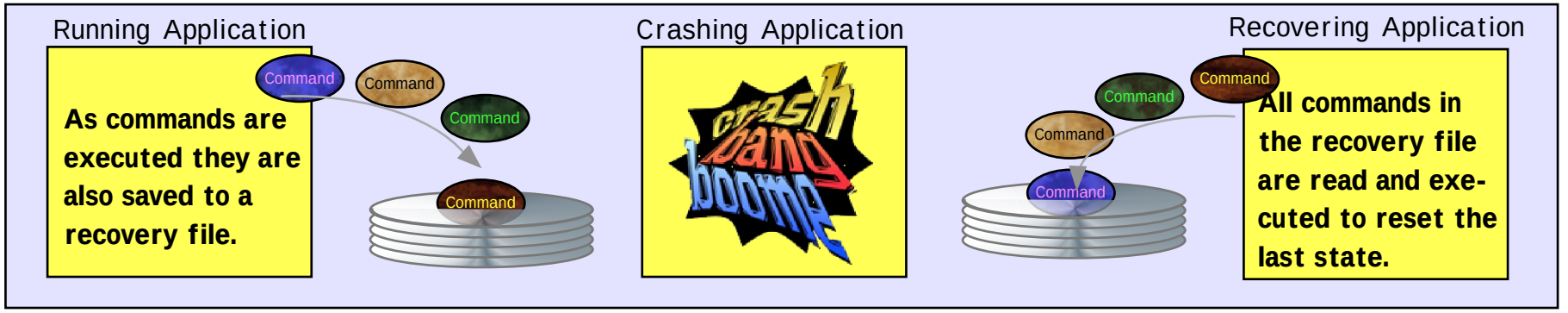
...and we add this function to the RemoteControl to handle the undo action.

```
void RemoteControl::undoButtonPushed()
{
    undoCommand->undo;
}
```

Design Patterns: The Command

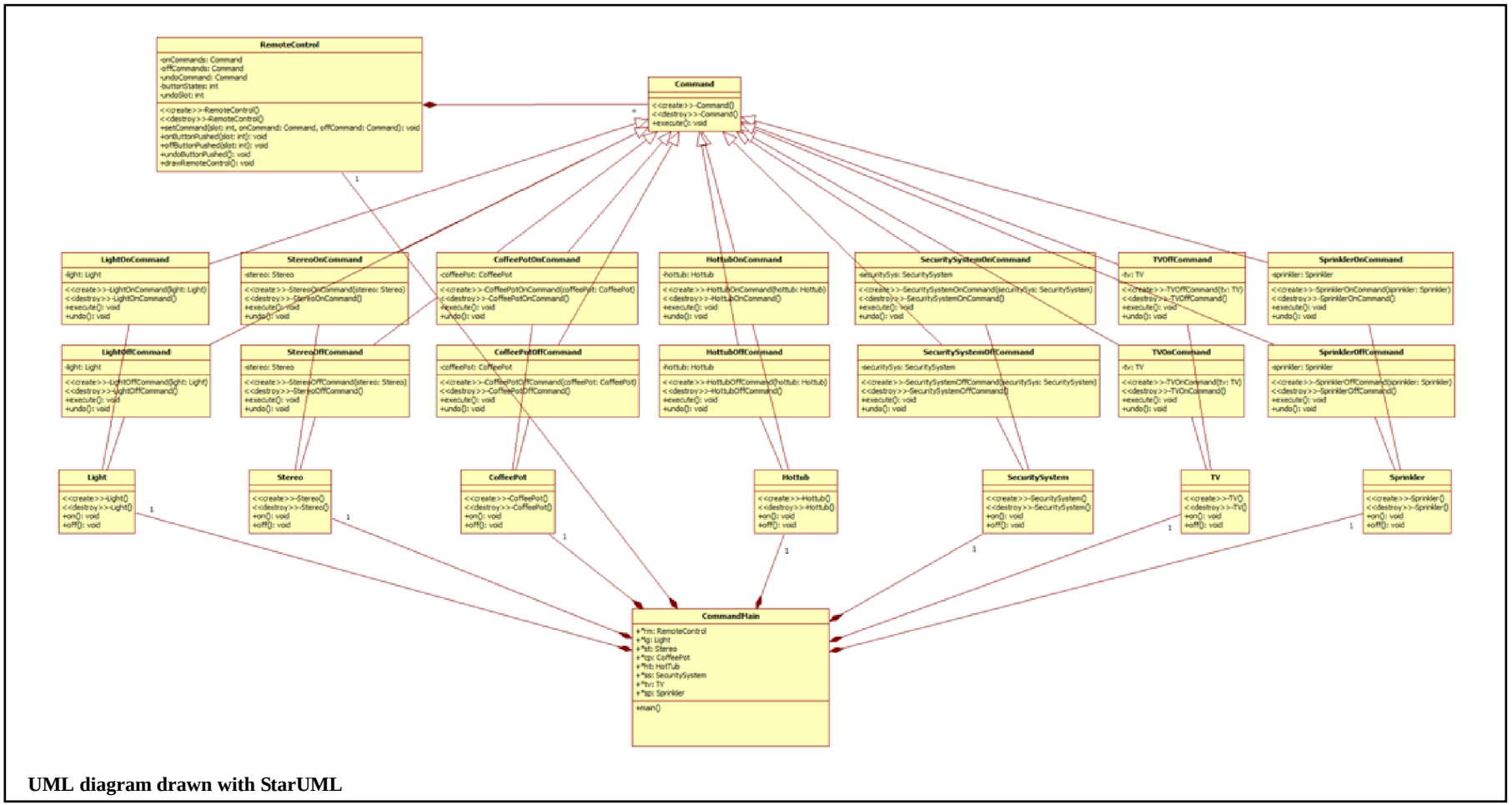
Doing More with Commands





Design Patterns: The Command

Code Sample



UML diagram drawn with StarUML

CommandMain

Creates a number of instances of Command and registers with the Invoker class RemoteControl

For each button input from user

Calls execute so Receiver can handle the command.

Let's look at the code and run the demonstration.