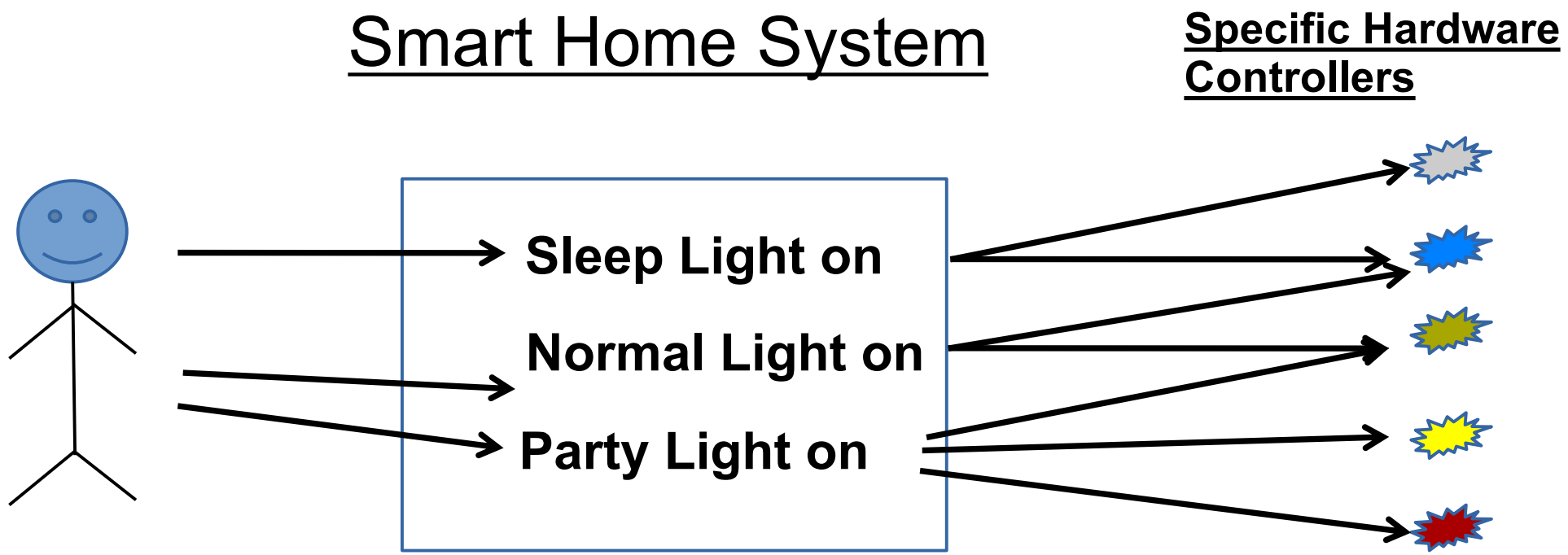
Command Pattern

Problem – Commands

- Sometimes in your application, you need to *issue* ***requests/commands to objects without knowing details*** *about the operation being requested, or the receiver of the request*.

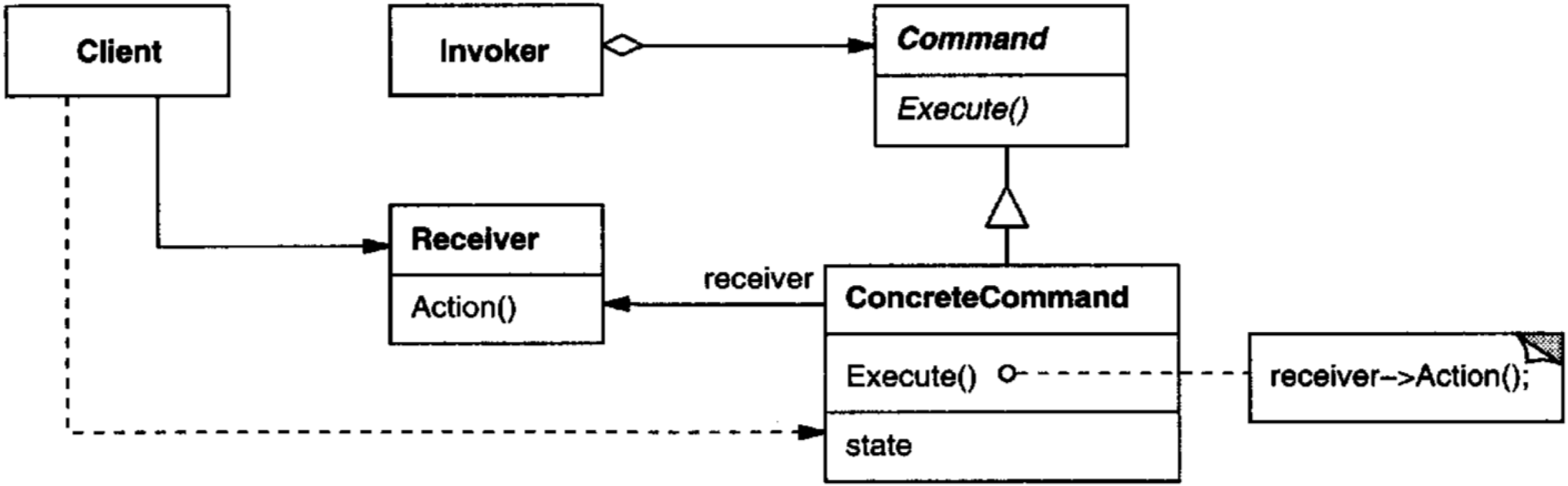
- You want to encapsulate groups of commands that control certain functionalities/behaviors together so that all of the details about sequence of commands are hidden from the client-side by encapsulating method invocation.

Smart Home Example – Commands



Command Pattern

- *"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo-able operations."*



Example – **RemoteLoader**

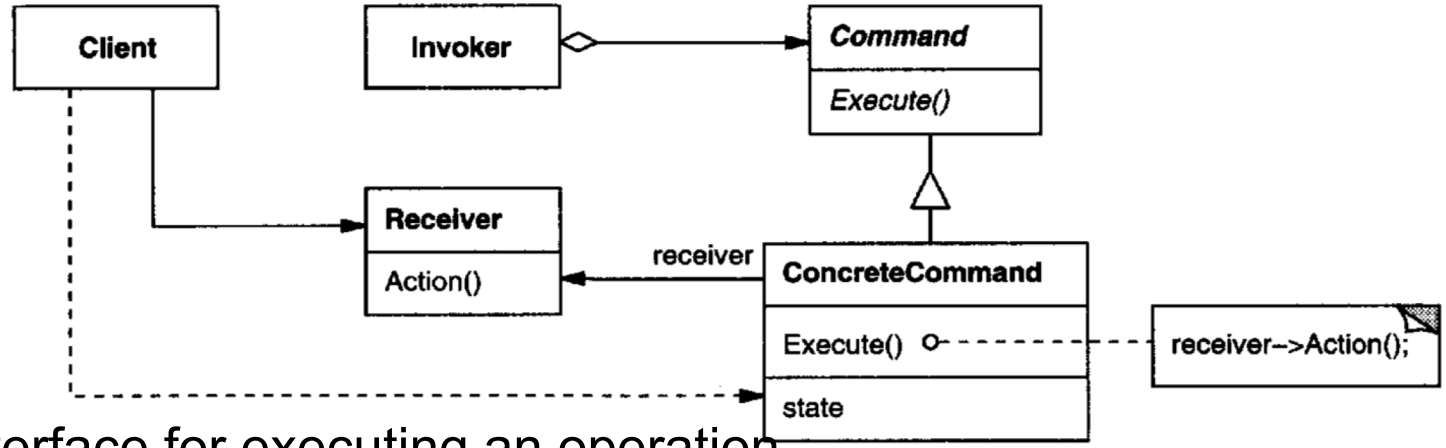
```
public interface Command {
        public void execute();
        public void undo();
}
```

```
public class MacroCommand implements Command {
        Command[] commands;

        public MacroCommand(Command[] commands) {
                this.commands = commands;
        }

        public void execute() {
                for (int i = 0; i < commands.length; i++) {
                        commands[i].execute();
                } }

   /**
    * NOTE:  these commands have to be done backwards to ensure
    * proper undo functionality
    */
        public void undo() {
                for (int i = commands.length -1; i >= 0; i--) {
                        commands[i].undo();
                }
        }.}
```

Example – SmartHomeController

```
public class SmartHomeController {
  public static void main(String[] args) {
        SmartHomeController remoteControl = new SmartHomeController();

        Light light = new Light("Living Room");
        TV tv = new TV("Living Room");
        Stereo stereo = new Stereo("Living Room");
        Hottub hottub = new Hottub();

        LightOnCommand lightOn = new LightOnCommand(light);
        StereoOnCommand stereoOn = new StereoOnCommand(stereo);
        TVOnCommand tvOn = new TVOnCommand(tv);
        HottubOnCommand hottubOn = new HottubOnCommand(hottub);

        Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};

        MacroCommand partyOnMacro = new MacroCommand(partyOn);
        remoteControl.setCommand(0, partyOnMacro, partyOffMacro);

...     }. }
```

Participants



- **Command**
  - declares an interface for executing an operation.
- **ConcreteCommand**
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client**
  - creates a ConcreteCommand object and sets its receiver.
- **Invoker**
  - asks the command to carry out the request.
- **Receiver**
  - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

When to use the Command Pattern

- Use Command Pattern when
  - **parameterize objects by an action to perform.** A **callback** function in a procedural language, is a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for **callbacks**.
  - **specify, queue, and execute requests at different times.** Commands can have a lifetime independent of the original request.
  - **support undo.** The execute operation can store state for reversing its effects in the command itself. The Command interface must have an added **Unexecute** operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list.

When to use the Command Pattern (continue)

- Use Command Pattern when
  - **support logging changes so that they can be reapplied in case of a system crash.** By augmenting the Command interface with load and store operations, you can keep a persistent log of changes.
  - **structure a system around high-level operations built on primitives operations.** Such a structure is common in information systems that support transactions. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. *("all or nothing"* in Atomicity of **Database - ACID Properties (Atomicity, Consistency, Isolation, Durability))**

Consequences of Command Pattern

- **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.
- **You can assemble commands into a composite command**.
  - In general, composite commands are an instance of the **Composite pattern**.
- It's **easy to add new Commands**, because you don't have to change existing classes.

Summery

- The Command pattern lets objects make requests of unspecified application objects by turning the request itself into an object.
- The command object can be stored and passed around like other objects. **The key** to this pattern is an **abstract Command class**, which declares an interface for executing operations.
- Command Pattern is **also known as Action or Transaction.**
- Command Pattern is one of the **Behavioral Patterns**