Behavioral Patterns

Command

Memento

We will discuss

- Separating request generator and request handler
  1. What to do when you want to give more than one object a chance to handle a request
     - Observer
     - Chain of Responsibility
  2. Encapsulate method invocation
     - Command pattern
- Without exposing internal representation
  1. of an object, save/restore the state of it
     - Memento
  2. of an aggregate object, access the elements of it sequentially
     - Iterator

- Decoupling sender and receiver of a request
  - Observer ✓
  - Command -- Today
  - Chain of responsibility
  - Mediator

COMMAND

Turning a method invocation into an object

Motivation Example

- Editor with Buttons, Menu, Shortcut Keys
- Req1: I want to save document with
  - Button click
  - Selecting a MenuItem
  - Using a shortcut CTLR+S
- Req2: I want to open and print a document with a Button instance, MenuItem, and shortcuts

Motivation Example

- Editor with Buttons, Menu, Shortcut Keys
- Req1: I want to save document with
  - Button click
  - Selecting a MenuItem
  - Using a shortcut CTLR+S
- Attemp1:
  - Hard code the request in all three classes

```
Button::onClick(){ document.save();}
```

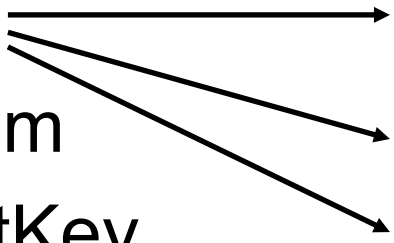
Motivation Example

- Editor with Buttons, Menu, Shortcut Keys
- Req2: I want to open and print a document with a Button instance, MenuItem, and shortcuts
  - Attempt1 hard coded the request in all three classes

  Button::onClick(){ document.save();}
  - Subclassing?
    - SaveButton, OpenButton, PrintButton
    - SaveMenuIte, OpenMenuItem, PrintMenuItem

The Problem

- Problem: Different **Invokers** need to issue requests to objects without knowing anything about **the operation** being requested or **the receiver** of the request.
- Invokers:
  - Button
  - MenuItem
  - ShortcutKey
- Operation and Receivers:
  - Document.save();
  - Document.open();
  - Document.print();



  - Subclassing leads to 9 sublasses

Command Pattern

- **Intent**
  - Encapsulate a **request** as an **object**, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
  - *Convert an operation into an object*
    - an object can be stored, passed, staged, shared, loaded in a table, ….

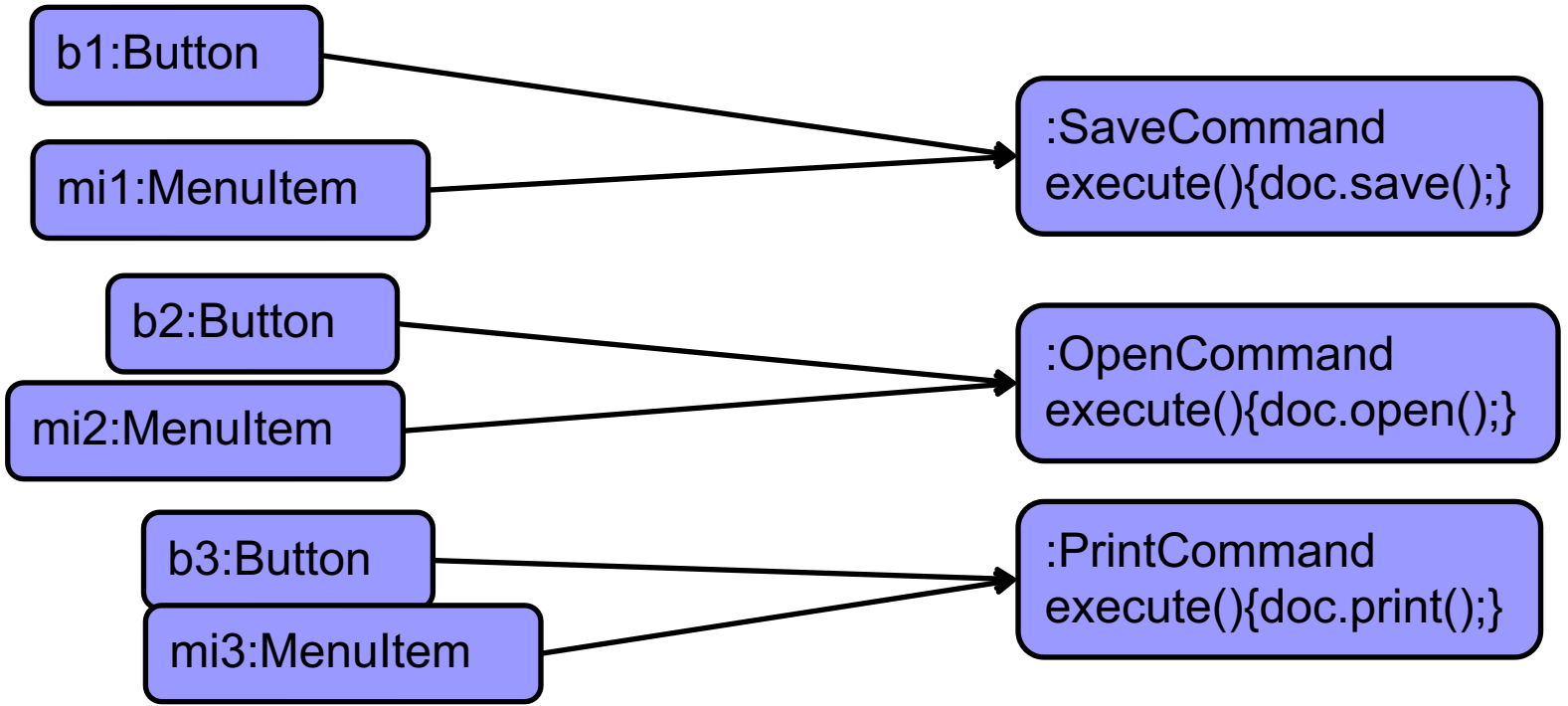
Command Pattern

- Command pattern turns the request itself into an **object**
- Each concrete Command class specifies a **receiver-action** pair by storing the *Receiver* as an instance variable
  - *callee*.**action**(); becomes an object

```
class Command{
        private Receiver callee;
        public void execute(){ callee.action();}
 }
```

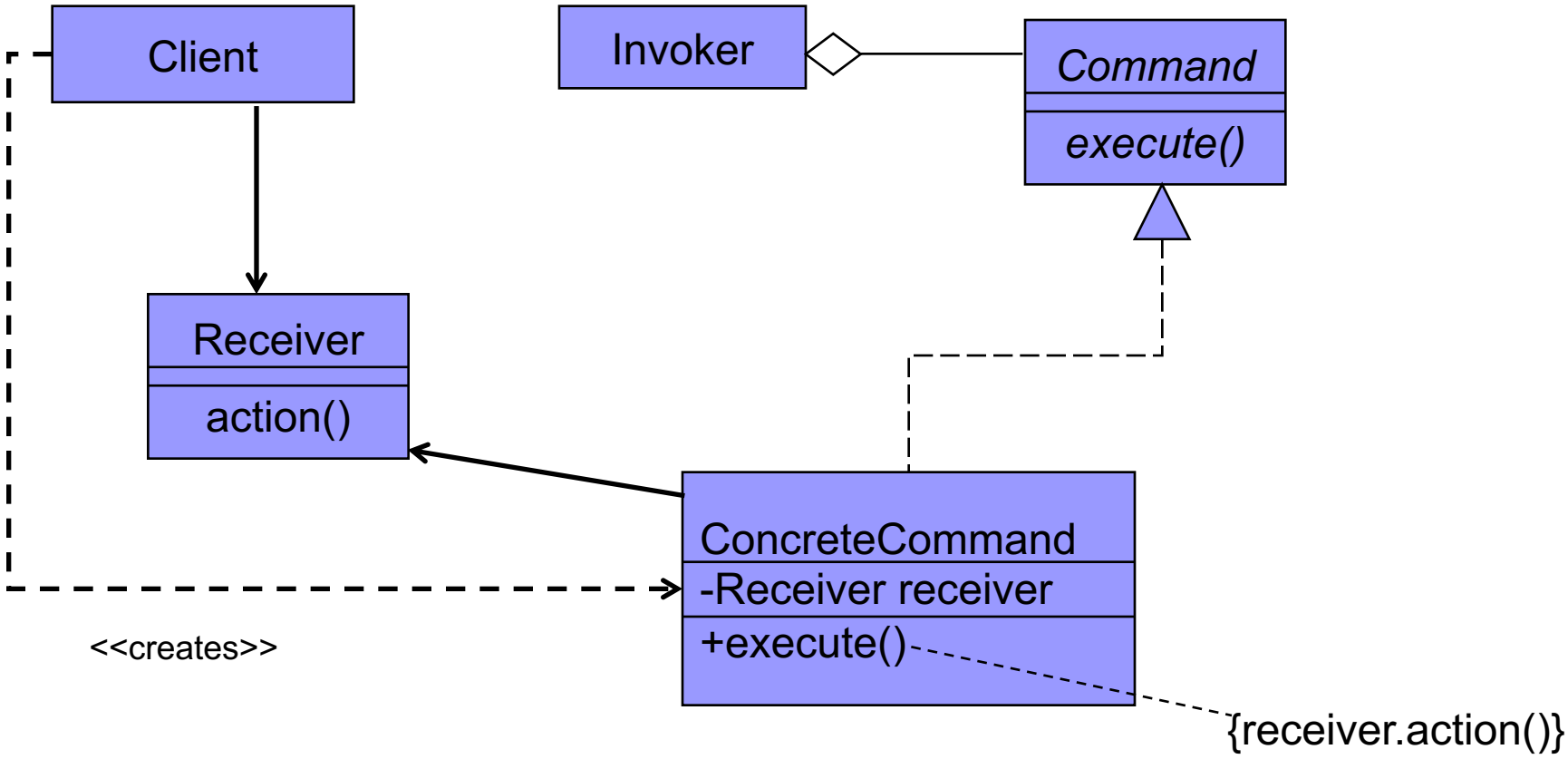
Compose invokers with Commands

- Invokers:
  - Button
  - MenuItem
  - ShortcutKey
- Operation and Receivers:
  - Document.save();
  - Document.open();
  - Document.print();



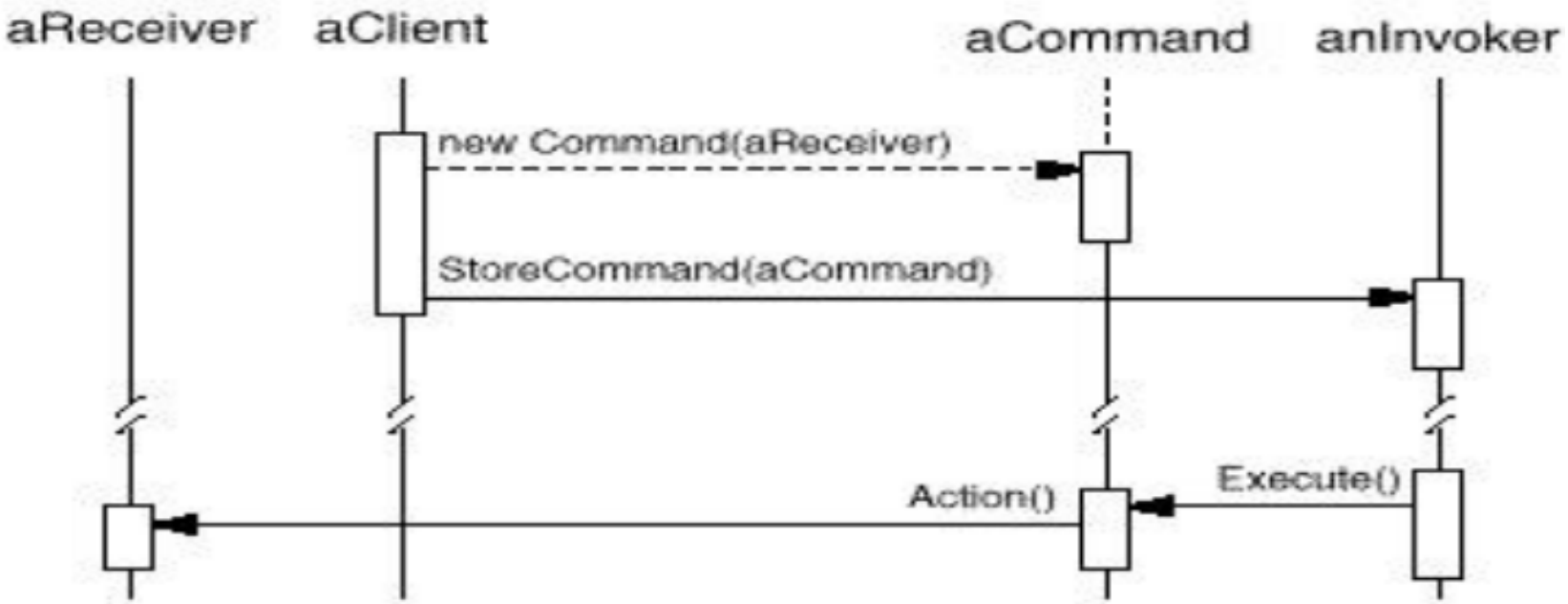
Structure – Command Pattern

Participants?

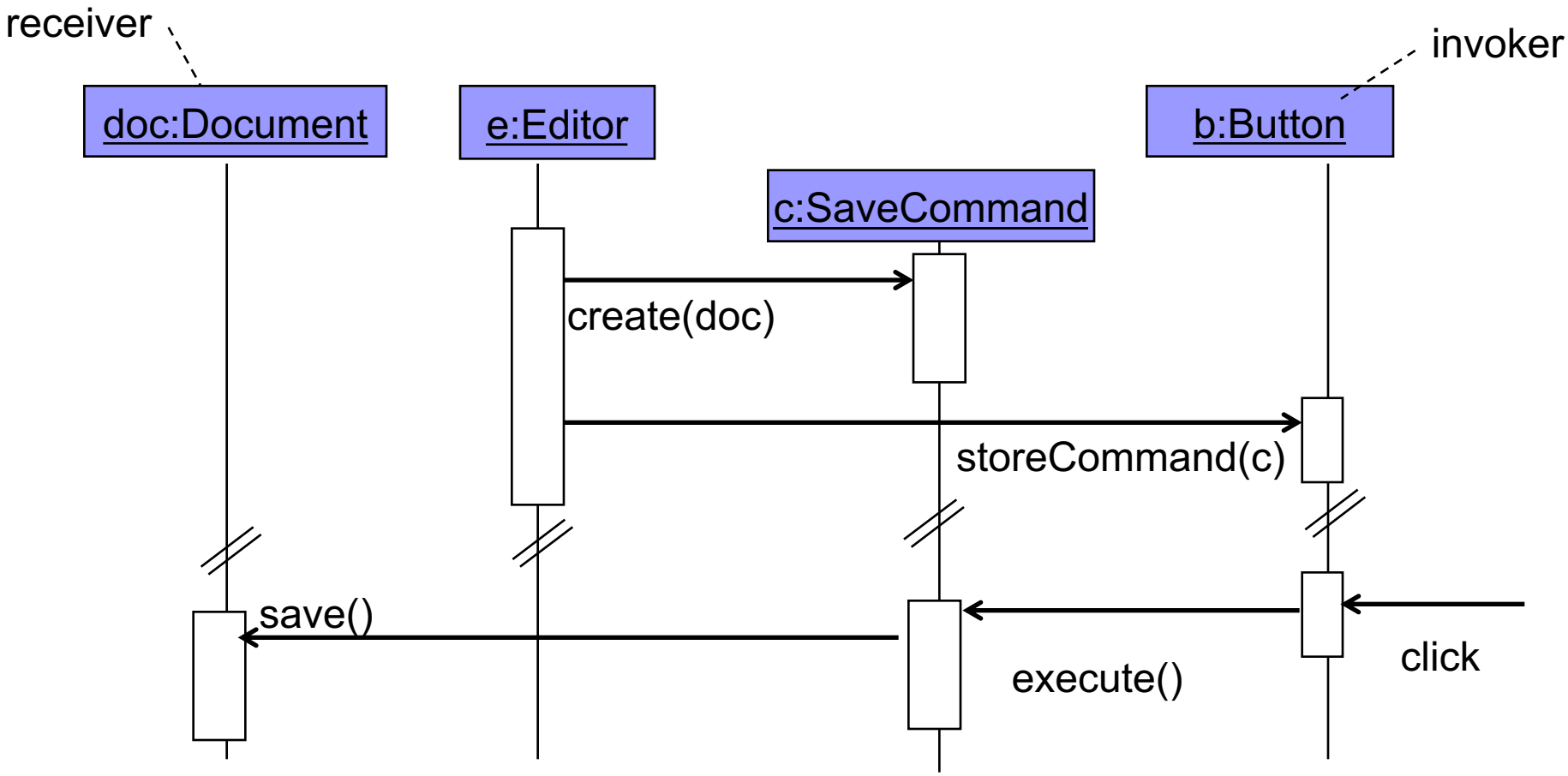


A concrete command has an instance of the receiver of the action and an execute method that invokes the receiver's operation

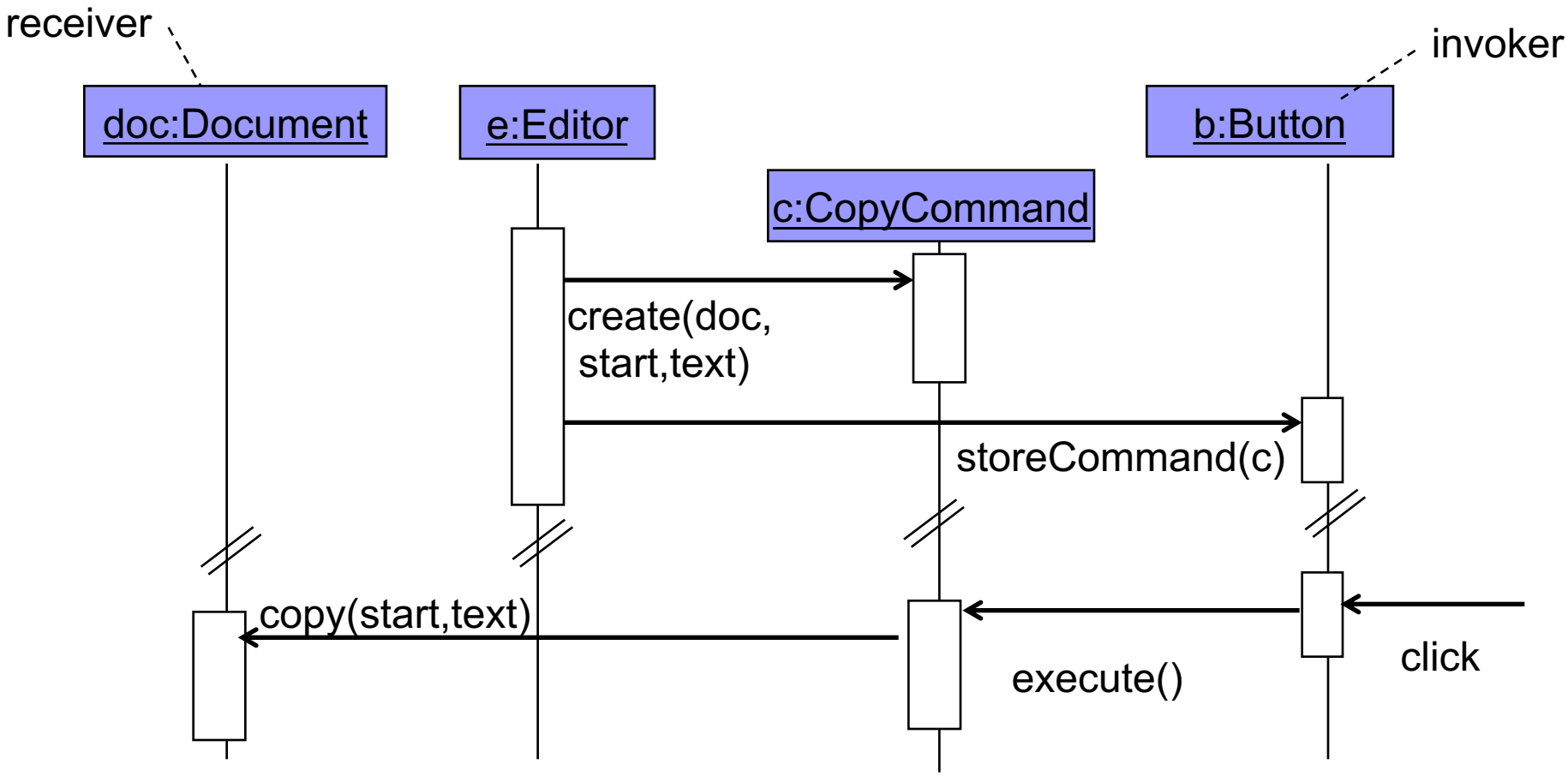
Structure – Command Pattern



Sequence Diagram – Save Button

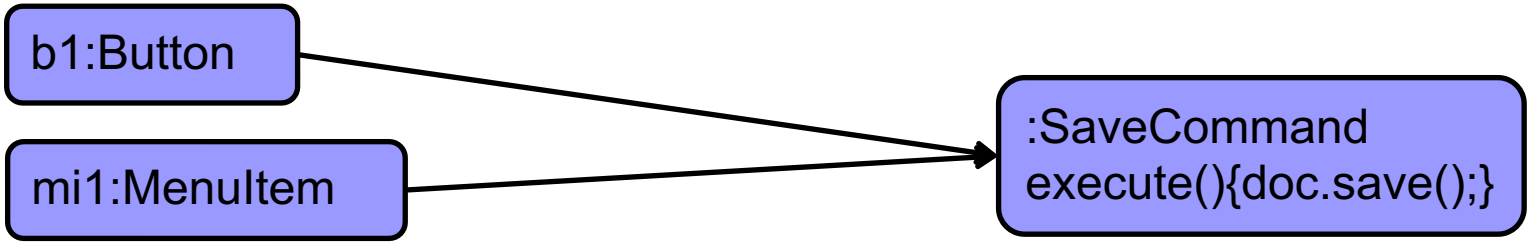


Sequence Diagram --Copy



We can also encapsulate an operation with parameters as a Command object

Configuring Invoker with Commands



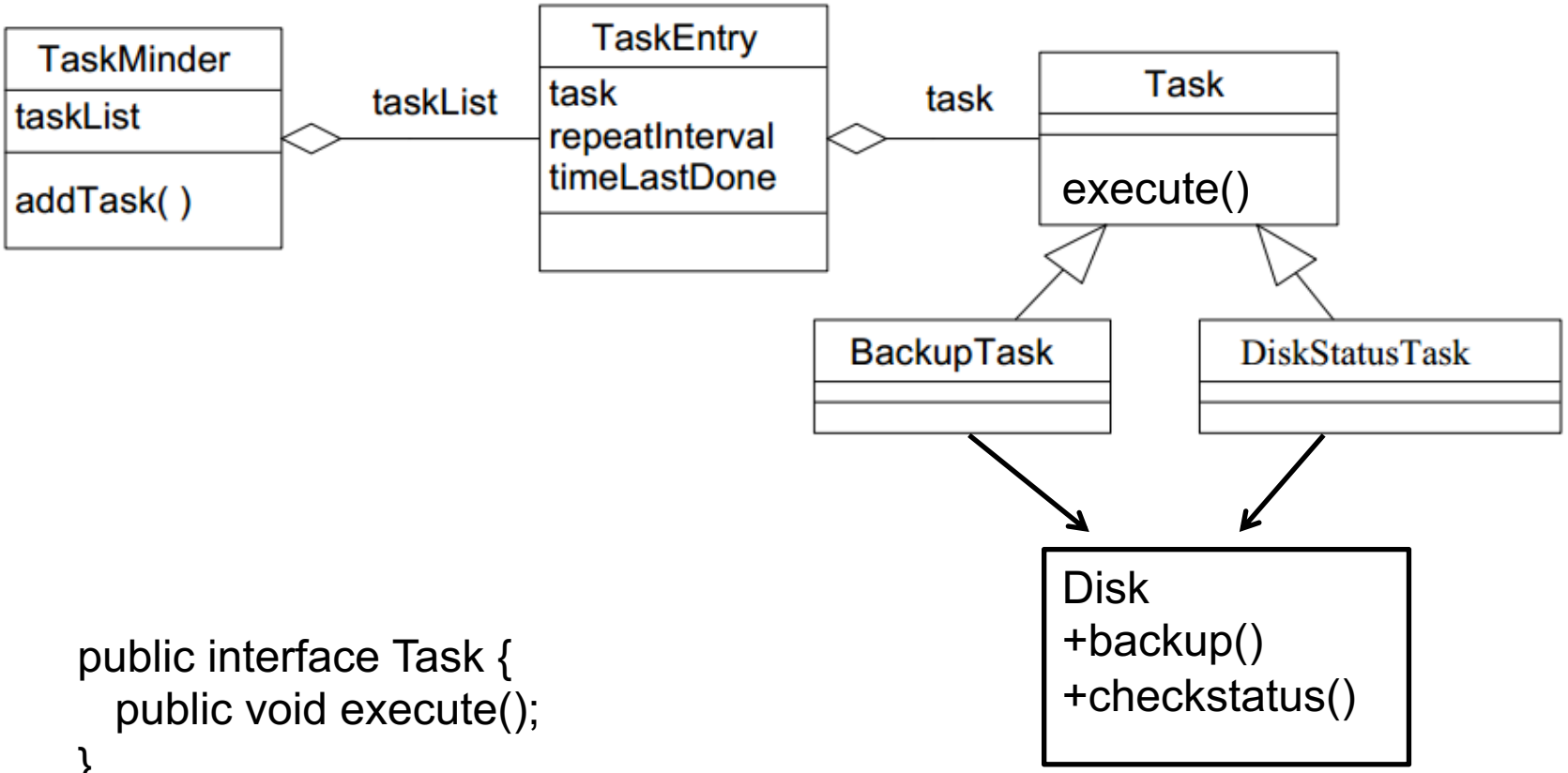
- You can express such parameterization in a procedural language with a **callback** function,
  - callback: a function that is registered somewhere to be called at a later point.
- Commands are an object-oriented replacement for callbacks.

Exercise : Task Schedular

- We want to write a class that can periodically execute one or more methods of various objects.
  - Example: we want to run a backup operation every hour and a disk status operation every ten minutes.
- But we do not want the class to know the details of these operations or the objects that provide them.
- We want to **decouple** the class that **schedules** the execution of these methods with the classes that **actually do** the behavior we want to execute.

Draw a class diagram

Exercise: Class diagram




```
public interface Task {
   public void execute();
}
```

Command Pattern

- Can do fancy things
  - Attach a method call to an object at runtime
  - Logging the requests
  - Queuing the requests
  - Deferring the execution of the request
  - Package the request and send it to another process
  - Undo the request
  - Reuse a command
    - multiple invokers, macro-commands
- Cannot do these easily if the client directly calls the receiver of the request
  - Invoker::foo(){receiver.request();} cannot do these

Applicability –when..

- Need to parametrize objects with actions
- Need to specify, queue, and execute requests at different times.
  - *Since a Command object can have a lifetime independent of the original request.*
  - Queuing the requests
  - Deferring the execution of the request
  - Package the request and send it to another process
    - If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.

Applicability –when..

- Need to support logging the requests
  - Log them 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.
    - Recovering from a crash involves reloading logged commands from disk and re-executing them with execute() operation.

Applicability –when..

- Need to support transactions
  - Transactions consists of series operations as a single logical work of unit.
  - A Macro-Command consisting of multiple commands (Composite)
    - Sequencing of commands
    - Same interface as Command
    - easy to extend the system with new transactions.

Macro Command Implementation

```
class MacroCommand : public Command {
   public:
     virtual void add(Command*);
      virtual void remove(Command*);
      virtual void execute();
       //…constructors, destructor
 private:
     List<Command*>* cmds;
};
void MacroCommand::execute () {
   for (const auto &cmd: *cmds)     cmd->execute();
 }
```

C++

Macro Command Implementation

Java

```
class CompositeCommand implements Command {
    List<Command> commands;
    public void execute() {
        commands.forEach(Command::execute);
    }
    //add, remove, constructor implementations
}
//alternative
    public void execute() {
        for(Command cmd: commands)  cmd.execute();
    }
```

Applicability –when..

- Need to support undo the request.
  - Undo last operation
  - Support both undo and redo
  - Multi-level undo and redo

Implementation issue-1: Undo

Supporting undo and redo.

- Command interface supporting undo

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

- A ConcreteCommand class might need to store additional data to undo the operation.
  - the arguments to the operation performed on the receiver, and
  - any original values in the receiver that can change as a result of handling the request.
- Receiver has operations that lets command restore the state.
  Light::on() and Light::off()

Example: Undoable Command

```
class MoveUnitCommand : public UndoableCommand{
public:
      MoveUnitCommand(Unit* unit, int x, int y)
                  : receiver(unit),  xBefore_(0),  yBefore_(0),    x_(x),    y_(y)  {}
      virtual void execute() {
           // to remember the unit's position before the move
           xBefore_ = receiver->x();
           yBefore_ = receiver->y();
          //action
        receiver->moveTo(x_, y_);
      }
     virtual void undo() {    receiver->moveTo(xBefore_, yBefore_);   }
private:
  Unit* receiver;  int xBefore_, yBefore_;  int x_, y_;
};
```

Example: Undoable Command

- Occasionally, we do not have to save the state

```
public class LightOnCmd implements UndoableCommand{
    private Light receiver;
    public LightOnCmd(Light light){receiver=light;}
    public void execute(){
          receiver.on();
    }
    public void undo(){
         receiver.off();
    }
   }
```

Undo/Redo

- Saving the last command executed is sufficient for one-level undo.

- Multi-level undo
  - Need a history list
  - A CommandManager that keeps a history
    - Undoable command list(?)
    - Redoable command list(?)

How does undo work?

- Ever used PhotoShop?
  - There is a stack of commands on the right
- Execution sequence
  - When ResizeCommand is activated, put it into the history
  - After a while, user chooses undo several times until that resize
    - All of the commands unexecute until then
  - ResizeCommand unexecutes and resize back
  - Suggest a data structure…

```
public class CommandManager{
   private Stack<Command> undoHistory;
   private Stack<Command> redoHistory;
   public void invoke (Command c){
       if(c instance of UndoableCommand){
           undoHistory.push(c);
       }else{
          undoHistory.clear(); redoHistory.clear(); //my choice
       }
       c.execute();
 }
 public void undo(){
      UndoableCommand c=(UndoableCommand) undoHistory.pop();
      c.undo();
      redoHistory.push(c);
 }
 public void redo(){
      Command c=redoHistory.pop();
      c.execute();
      undoHistory.push( c);
 }}
```

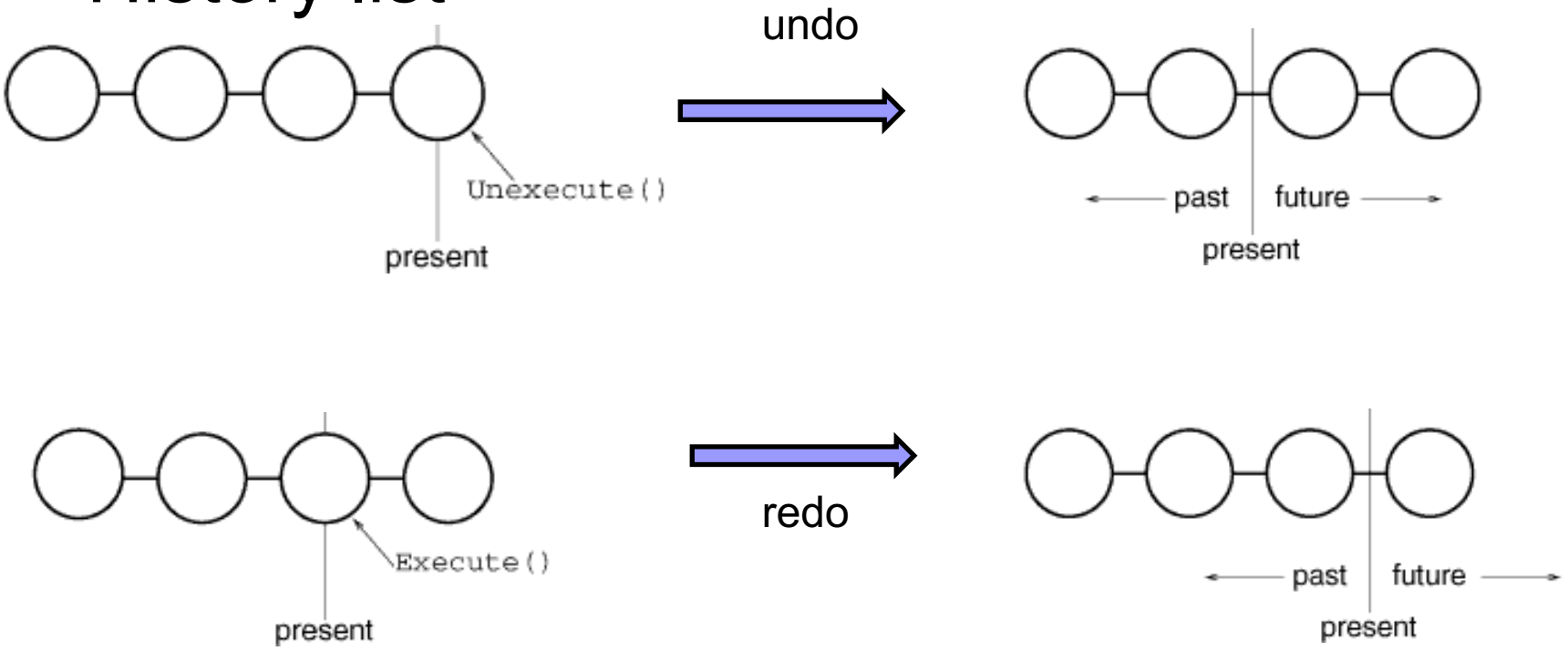
Multi level undo/redo

- Photoshop has a list of commands executed on the right
- Execution sequence –undo any action
  - User chooses one of the operations in the history list to cancel/undo
  - Undo: traversing backward through the list while calling undo() of commands
  - Redo: traversing forward and execute() of commands

Suggest a data structure

Command History

- two stacks, or
- History list



Impl. Issues – Undo/Redo

- Save a copy of the Command object in the history
  - When Command object has state how to undo that operation
  - for distinguishing different invocations of the same command if its state can vary across invocations.
- Example: a DeleteCommand that deletes selected text must store different text and position each time it is executed.
  - Save a copy of the current DeleteCommand in the history.
    - Which pattern?
  - DeleteCommand object can delete some other text later
- if the command's state never changes, put only a command reference in the history
  - E.g. LightOnCommand

```
public class DeleteCmd implements UndoableCommand{
  private Document doc;   private int start, end;
  private FormattedString text; //saving state
  public DeleteCmd(Document d, int start, int end){..}
  public void execute(){
       text=doc.delete(start,end); //remember for undo
  }
  public void undo(){
       doc.insert(start, text);
  }
}

//setup
 JButton deleteButton=new JButton("insert");
 deleteButton.addActionListener(){
    new ActionListener{
       public void actionPerformed(ActionEvent e){
          Command c=new DeleteCmd(doc,getSelection().start(),
   getSelection().end());
            commandManager.invoke(c.clone());}
     });
```

Commands can be shared

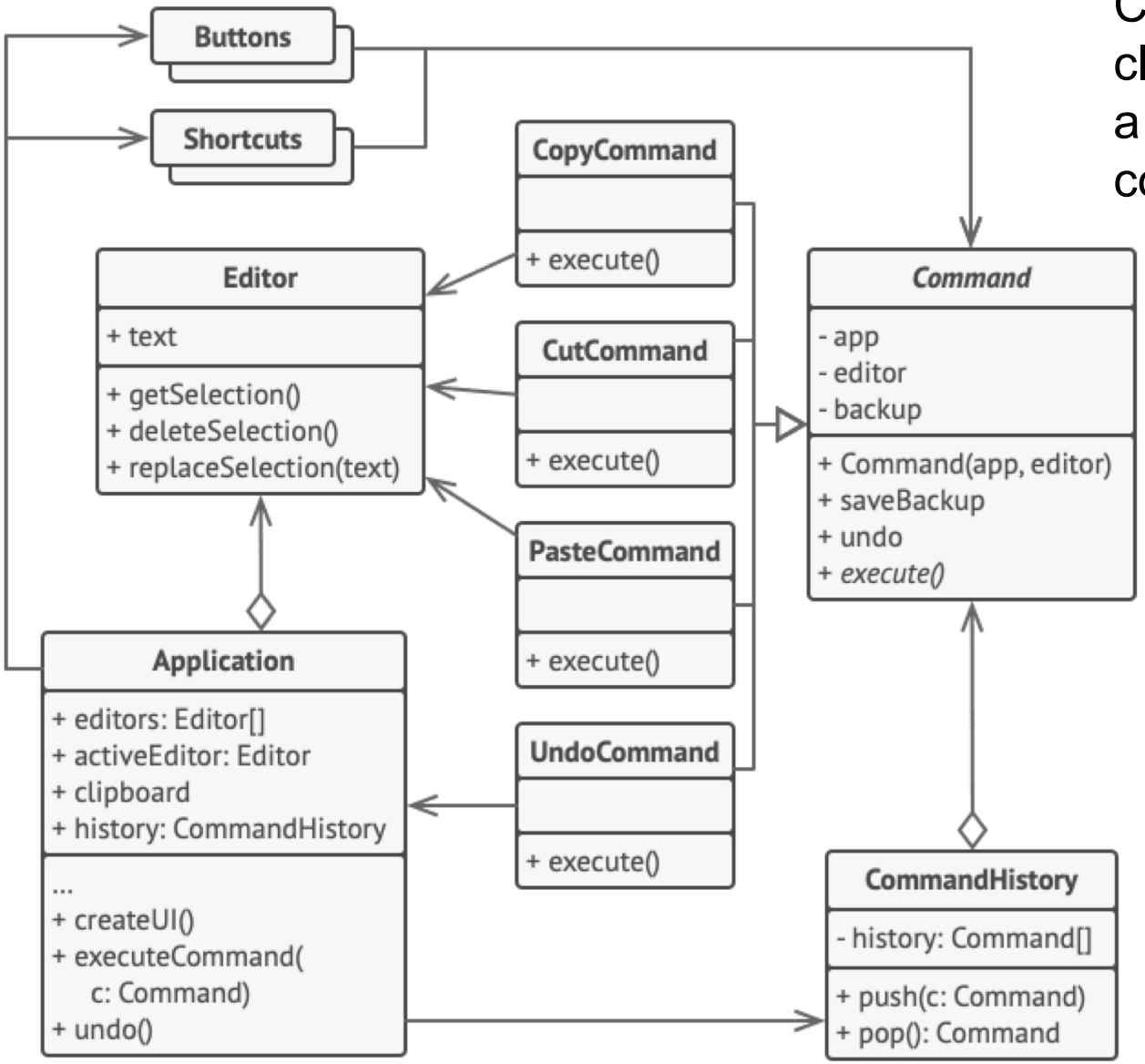
- Undo action can be shared

```
ActionListener undoAction=  new ActionListener{
   public void actionPerformed(ActionEvent e){
     commandManager.undo(); }
   });    //yes, ActionListener is both observer and command

JButton undoButton=new JButton(“undo”);
undoButton.addActionListener(undoAction);

JMenuItem undoItem=new JMenuItem(“undo”);
undoItem.addActionListener(undoAction);
```

An alternative design. Command is abstract class implemented as a base for Editor commands.



Application acts as command manager

https://refactoring.guru/design-patterns/command

Implementation issues -2

How intelligent should a command be?

- At one extreme it merely defines a binding between a receiver and the actions that carry out the request.
- At the other extreme it implements everything itself without delegating to a receiver at all.
  - When you want to define commands that are independent of existing classes,
  - When no suitable receiver exists, or
  - When a command knows its receiver implicitly.
  - e.g., a command that creates another application window may be just as capable of creating the window as any other object.
- Somewhere in between these extremes are commands that have enough knowledge to find their receiver dynamically.

Implementation issues -3

Function pointer and C++ templates.

- For commands that (1) are not undoable and (2) do not require arguments

```
template <class Receiver>
class SimpleCommand : public Command {
  public:
     typedef void (Receiver::* Action)();
     SimpleCommand(Receiver* r, Action a) : receiver(r), action(a) { }
      virtual void execute();
  private:
      Action action;
      Receiver* receiver;
};
```

```
template <class Receiver>
void SimpleCommand<Receiver>::execute
() { (receiver->*action)(); }
```

Implementation issues-3

Function pointer and C++ templates.

```
template <class Receiver>
class SimpleCommand : public Command {
  public:
     typedef void (Receiver::* action)();
     SimpleCommand(Receiver* r, Action a) : receiver(r), action(a) { }
      virtual void execute();
  private:
      Action action;
      Receiver* receiver;
};
```

```
template <class Receiver>
void SimpleCommand<Receiver>::execute
() { (receiver->*action)(); }
```

```
int main(){
MyClass* receiver = new MyClass;
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::operation);
```

C++ Functors as Command

```
class  Command { //FUNCTOR
 public: virtual ~Command(){}
            virtual void operator ()()=0;
};
class LightOnCmd : public Command {
public:
    LightOnCmd(Light& l,const string&
m) :
                              light(l), msg(m)
{}
    // Override the "execute" operator
    void operator()() override {
        light.on(msg); }
private:
       Light& light; string msg;
};
class Light{ //Receiver
```

```
int main() {
    Light light;  RemoteControl remote;
    // Turn on the light
    remote.setCmd(
          new LightOnCmd(light, "on"));
    remote.pressButton();
}
// Invoker
class RemoteControl {
public:
    void setCmd( Command* cmd) {
        command.reset( cmd);}
    void pressButton() {
         if (command) (* command)(); }
//calling!
private:
     std::unique_ptr<Command>
```

STL Functors as Command

```
#include <functional>
// Command
using Command = std::function<void()>;
// Invoker
class RemoteControl {
public:
    void setCmd(const Command& cmd) {
        command = cmd;}
    void pressButton() {   command(); }
private:
    Command command;
};
class Light{//Receiver
  public: void on(const string& msg);
   void off();
```

```
int main() {
    Light light;
    RemoteControl remote;

    // Turn on the light
    remote.setCmd([& ]{ light.on("on"); });
    remote.pressButton();

    // Turn off the light
    remote.setCmd([&] { light.off(); });
    remote.pressButton();

}
```

Using Lambda as command

- Assume
  - class Invoker{public void addCommand(Command c){..}..}
  - public interface Command{
    public void execute();}
- Creating a command object

```
public void someMethod(Invoker invoker){
    Document receiver=createDoc();
    invoker.addCommand( () -> receiver.save() );
```

- Lambda *captures* the receiver in the lexical closure.

Lambda in remote command

- Invoker has addCommand(MyCommand c){..}
- public interface MyCommand extends Runnable, Serializable { }
  - This is a marker interface
  - We might send the command on another JVM
  - Using the public void run() of Runnable as the execute method
- invoker.addCommand(

  () -> System.out.println(“a simple command"));

https://blogs.oracle.com/javamagazine/post/the-command-pattern-in-depth

Command- Consequences

- Complete decoupling between the sender and the receiver
  - Receiver knows how to perform the action
  - Invoker is unaware who performs the action
- A request becomes a command object that can be manipulated and extended like any other object
  - Command is a first-class object
- Commands can be assembled into a composite command
- Enables to implement deferred execution of operations
- Easy to add new commands, because you don't have to change the existing classes.

Known uses

- All implementations of java.lang.Runnable
- All implementations of javax.swing.Action
- ActionListener is both Observer and Command
  - addListener(new ActionListener{

```
    public void actionPerformed(ActionEvent e){
        //this is the execute method
        //it is the update method as well
    }
});
```

Thread safety

- Command manager –use a thread safe data structure and make it final
  - Final to eliminate race conditions at initialization
  - Thread safe data structure for swapping command between lists safely
  - Same goes for command queues
- The receiver methods that Command::execute invokes should be thread safe
  - Be careful about deadlock: threads holding locks waiting each other to release lock
- Commands do not have state, but if they do make it immutable

Related patterns

- Macrocommands with **Composite**
- **Chain of Responsibility** can use Command to represent *Requests* as objects.
- Handlers in **CoR** can be implemented as Commands.
  - we can execute a lot of different operations over the same context object, represented by a request.
- A command that must be copied before being placed on the history list acts as a **Prototype**

Related patterns

- Decoupling request sender and receiver
  - **Observer** broadcast
  - **Mediator** is centralized communication control
  - **CoR** sends request down the chain
  - **Command,** invoker is unaware of receiver and the action
- **Strategy** and Command
  - use both to parameterize an object with some action.
  - Intents! Strategy lets us swap algorithms in a context
  - Command converts an operation into an object, puts them into operation queue, make history, undo them
- Memento (next)

Can I undo any command?

- PhotoShop with a history list on the right
- Execution sequence- undo last action
  - When BlurCommand is activated, put it into a stack – keep a history
  - After a while, user chooses undo the last operation
  - BlurCommand unexecutes the blurring action
    - What? How?
- How to undo irreversible action?
  - Do not support undo; print a message
  - Save the previous image with Memento (next pattern)

MEMENTO

Restoring a previous state

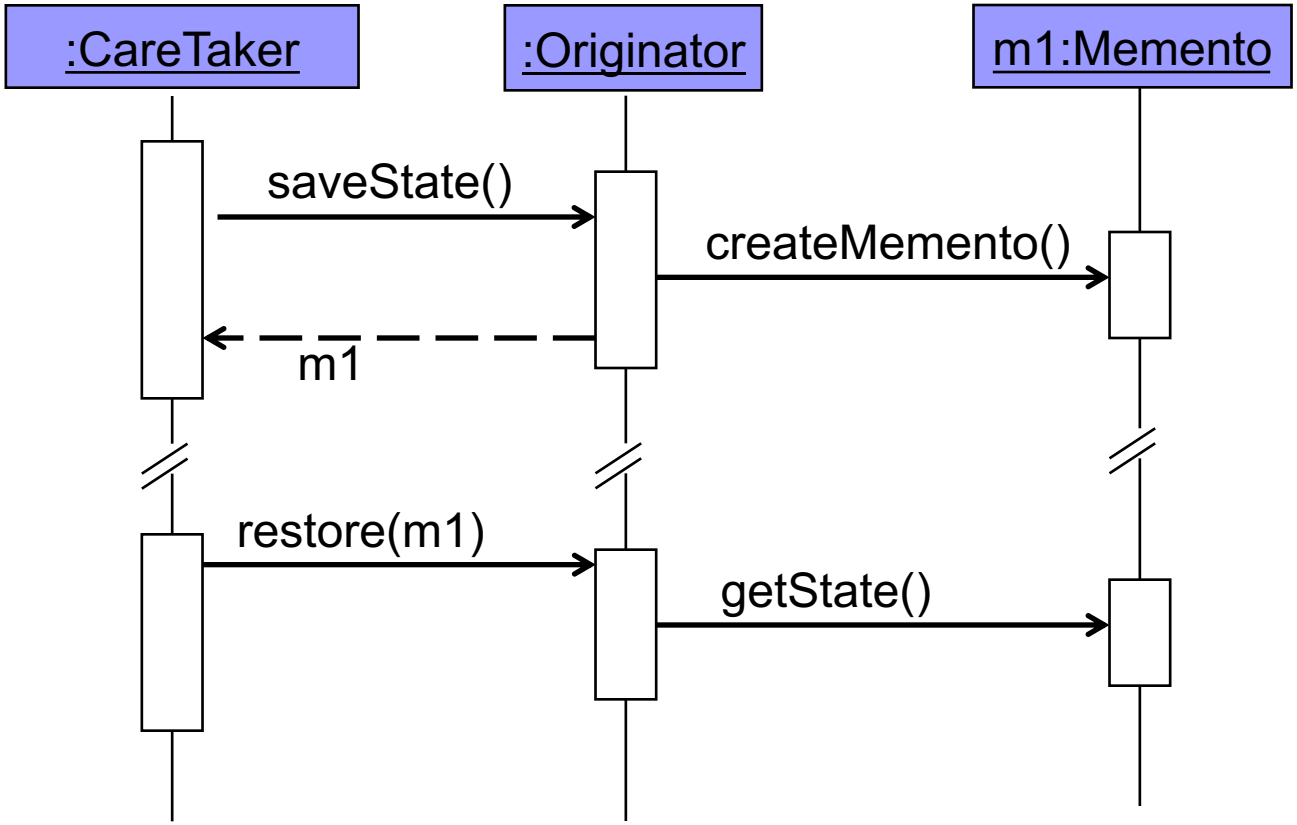
- Undo an irreversible action
  - How much information to store about the old state?
  - Who creates the info to store?
  - Who should store this info for later use? Command?
  - How to keep the visibility restrictions of the document intact?
  - How would the image restore itself the previous state?
- Restore the game to a checkpoint
- Restore a repository to one of the previous state

Memento

- **Intent**

  Without violating encapsulation, capture and externalize an object's state so that the object can be restored to this state later

Memento - Collaborations

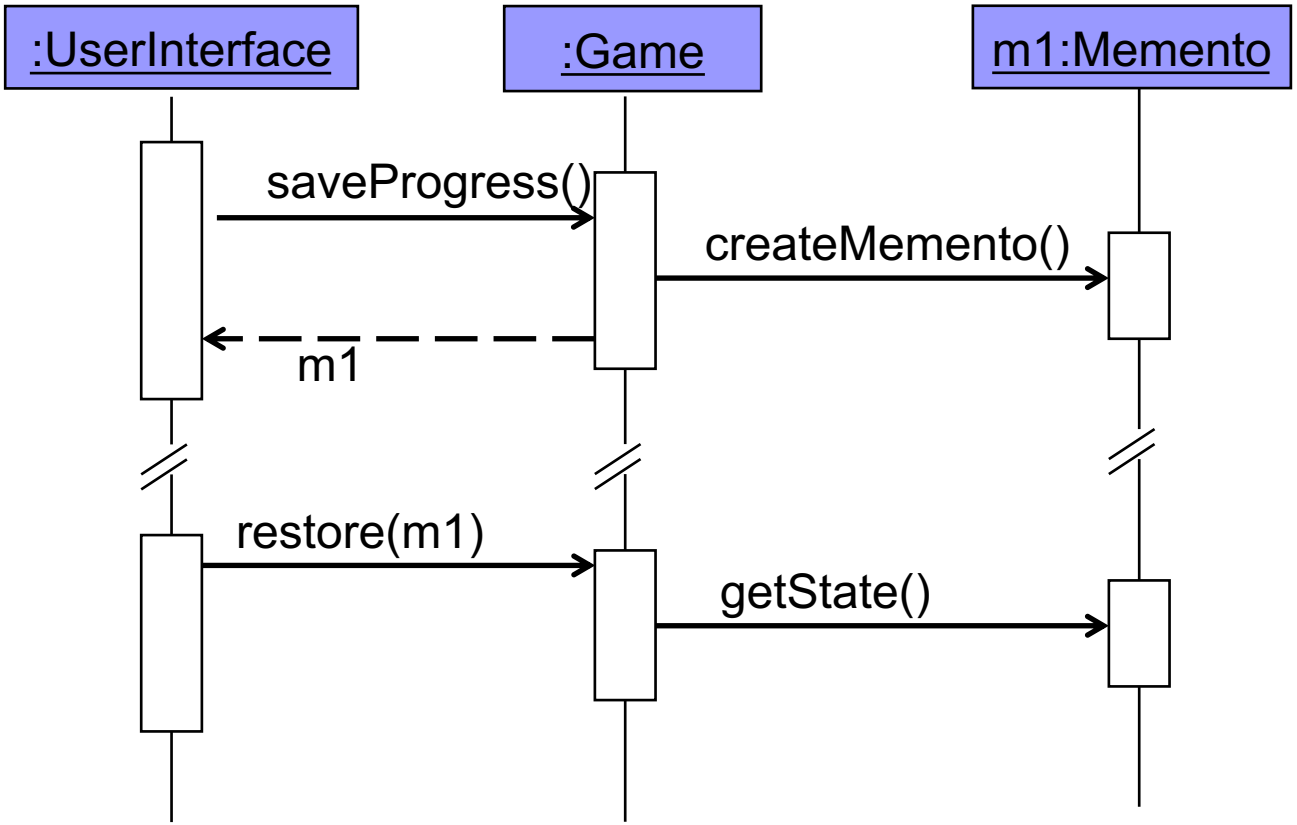


The Caretaker holds the Memento, but *cannot* look inside.

Memento-Collaborations

- A backup is requested from an originator,
  - a memento object is created,
  - populated with the originator's private information,
  - returned to the caretaker.
- The caretaker cannot view this information
- Originator and memento classes share information without exposing any of it to other classes
- Caretaker wants to restore
  - return the memento to the originator
  - have the originator's private attributes reset to the earlier state

Restore to a checkpoint



- Promote undo or rollback to full object status.
- A magic cookie that encapsulates a "check point" capability.

Memento Answers

- How much information to store about the old state?
  - Originator, the object itself knows best what it needs to restore back.
- Who creates the info to store, i.e. the Memento?
  - Originator creates the Memento. Originator puts its confidential information inside the Memento
- Who should store this info for later use?
  - Just like in games and code repositories, we need a caretaker. Database, file, dictionary, etc.
- How to keep the visibility restrictions of the document intact?
- How would the Originator restore itself the previous state?

Design Forces

- Do not violate encapsulation!
  - Originator's state is stored in a memento
  - ***Only*** the originator should be able to see the internal information
- When the caretaker wants to restore
  - Return the memento to the originator
  - Originator sets it state back
  - Caretaker *cannot* see/examine the memento's internals

How not to break encapsulation?

- Memento has two interfaces.
  - Wide interface for originator
    - Getter methods or access to fields
  - Narrow interface for everyone else
    - Private Getter methods and fields
- C++ friends keyword makes it easy
  - Caretaker sees the memento interface that expose no information
  - Originator is a “friend” of memento
- In Java, a little complicated…
- BAD: public methods and 'hoping' no one misuses the memento.

Originator is a friend of Memento

```
class Memento {
public:
    // narrow public interface. Zero service.
    virtual ~Memento();
private:
    // private members accessible only to Originator
    friend class Originator;
    Memento(); //nobody else should create a memento
    //setter and getters for attributes that helps originator to
restore
    void setAttribute(Field*);
    Field* getAttribute();
private: //fields that stores the originator's info
};
```

Originator sees the wide interface

```
class Originator {
  public:      // ...
     Memento* createMemento();
     void restore(const Memento*);
  private:
     Field* f1;   ….   // internal data structures
  };
Memento* Originator::createMemento(){
    Memento* m=new Memento();    //private, but I am a friend
    m->f1=this->f1; /*save necessary info to remember*/
    return m; }
void Originator::restore(const Memento* m){        if(m==0) return;
     this->f1=m->f1;        //some mechanism to restore the internal
values
 }
```

Memento is an inner class Object

```
public class Originator{
    private Field1 f1; //..and other fields.
    public Object saveState(){ //or createMemento
            return new Memento();
     } //narrow interface is the Object interface
    public void restore(Object o){
            if(!o instanceof Memento)) { //it is not mine
                printErrorMsg(); return;}
            Memento m=(Memento) o;  //now it has wide interface
            f1=m.mf1; ….//restore the fields using memento's fields
    }
    private  class Memento{
        Field mf1; /// and other fields
     Memento(){mf1=f1; …..}   //save the values of necessary Originator
fields
```

This is a suggestion. Any implementation that gives a wide interface to the Originator and a narrow interface to anything else is welcome

Implementation issues

- Memento could have less state information than the originator
  - Store only the changing parts
  - Memento could just save incremental changes when there is a predictable change sequence
- If you need all private information, create the memento by cloning the original object.
  - Alternatively, toString() and write to a file
- Serialization is a creation of memento

Undo and memento

- Many possibilities
  - **Command** object itself can be the **caretaker** (better)
  - If not, save the command and memento as a pair in the undo/redo history
  - Commands in the history are in execution order, so memento could save only incremental changes
- The memento pattern does not specify how caretaker works other than it should see the narrow interface

```
class TypingCommand implements Command { //caretaker
  private TextEditor editor;
  private String textToAdd;
  private Memento backup; // Magic token!
  public TypingCommand(TextEditor editor, String text) {
        this.editor = editor; this.textToAdd = text; }
  public void execute() {
     backup = editor.save();
     editor.addText(textToAdd);
  }
 public void undo() {   //restore the originator using memento
     if (backup != null)  editor.restore(backup); }
}
```

Originator of Memento AND Receiver of Command

Originator creates memento. Command is the caretaker of it.

Memento - Consequences

- Simplifies the originator
  - No storage management for copies in the originator
  - Not keeping track of its previous state since this is the responsibility of the CareTaker
- Memento **avoid exposing** information anyone other than the originator
  - Preserving encapsulation
- Might be expensive
  - Depends on copy cost, storage cost

Known uses

- All implementations of java.io.Serializable
- All implementations of javax.faces.component.StateHolder
- Load and save checkpoints

Related patterns

- **Command** can use Memento to maintain the state required for an undo operation.
- Command and Memento act as magic tokens to be passed around and invoked at a later time.
  - In Command, the token represents a request;
  - in Memento, it represents the internal state of an object at a particular time.
- An **iterator** can use a Memento to capture the state of an iteration.
  - Iterator stores the memento internally

References

- Design Patterns: Elements of Reusable Object-Oriented Software. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994).
  - GoF book
- Dive Into Design Patterns, Refactoring guru, A. Shvets, (2021)
- Game programming patterns, R. Nystrom (2014)
- Head First Design Patterns. Freeman, E., Robson, E., Sierra, K., & Bates, B. (2004).