Undo

The BIG item

- UNDO
- Why?
- Varieties of Undo/Redo
- The two pattern that serve as a basis to code it
- How to make undo/redo overview
- In code overview

What?

You have all seen undo at this point…moving on…

Why?

To err is human, you need a computer to really mess up

William E. Vaughan (probably)

We make mistakes, we need way to undo, we also need a way to undo an undo!

Why?

- Allow, people to go “hey, what does this do?”

- Not only is this a safety net, but…

NOT having a clear “undo” is a common cause of failure! Users are scared of damaging their computer!

Undo varieties

- Undo, is not always “undo”
- There levels of…
  - What is permitted
  - How far back
  - Details

Types of undo

- Functional
  - Undo the last action I took

- This is the most basic, but can be
  - Blind - “just undo what ever happened last”

  Or

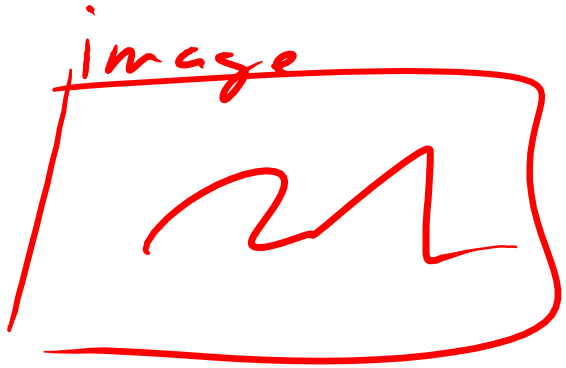
  - Explanatory – states what the undo does

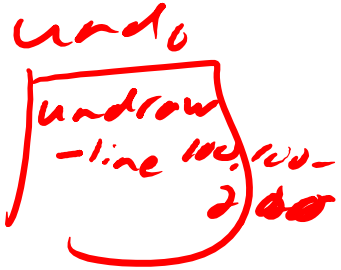
Types of undo

- Blind
  - This is relatively easy to code
  - A problem when a single click actually does 2+ "commands"
  - Case study:
    - MSVS pasted in code often aligned everything way to the right. One undo undoes the alignment, and the second removes the code

Types of undo

- Explanatory
  - States what to undo
    - Harder and more information to process
    - Benefit of clarity





Type of undo

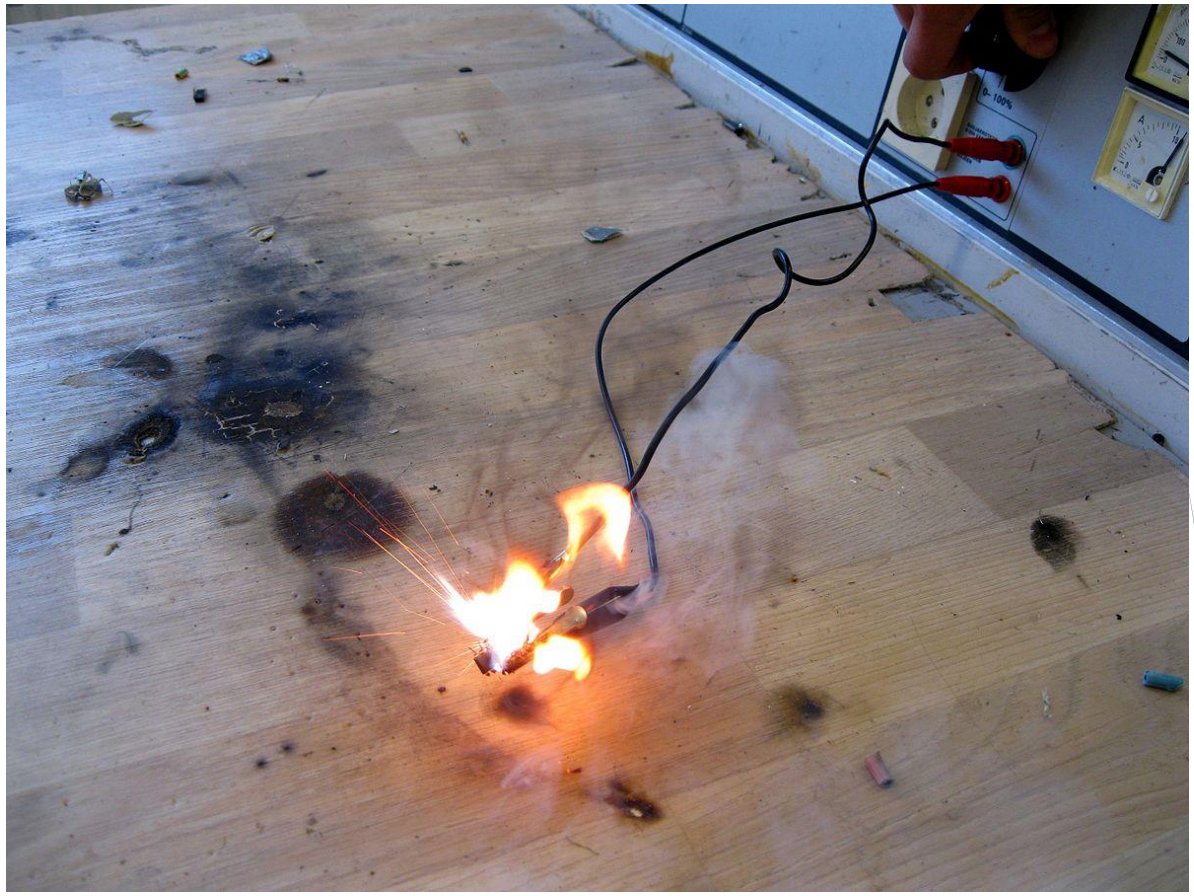
- Single versus multiple
  - Single is one function at a time
  - Multiple is multiple

Type of undo

- This may seem straightforward, but…
  - Single is assumed, but easy to short circuit

How often have you hit undo and then accidently hit space and now you have to redo your work?

  - Multiple is harder to code, but more efficient



Type of undo

- Often undo operations can be grouped to make them more efficient
  - This could be only store a version
  - This could be a category too
  - Versioning another example

Gitlab

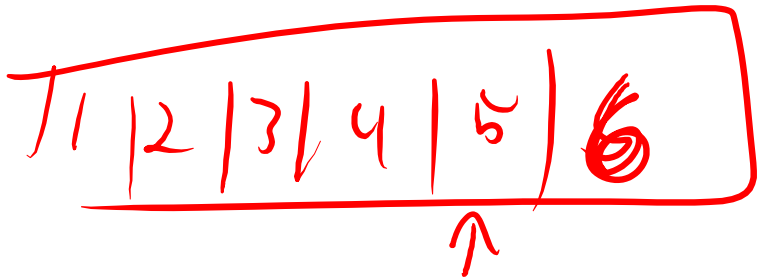
Word (typing)

- Some users will implement “undo” themselve with having multiple versions of documents!

Redo

- This "semi" fixes the undo short circuit

- Redo stores the functions and now we have a list rather than a stack
  - The challenge is when do we clear it?



Freezing and comparing

Another option to undo/redo

Sometimes data is so important we don't want to risk it

- Freeze it with a dialog box
  - This add a layer of security
  - But overused and typically ignored…
- Compare it
  - This allow the user to see the change before it becomes permanent

Pattern: undo

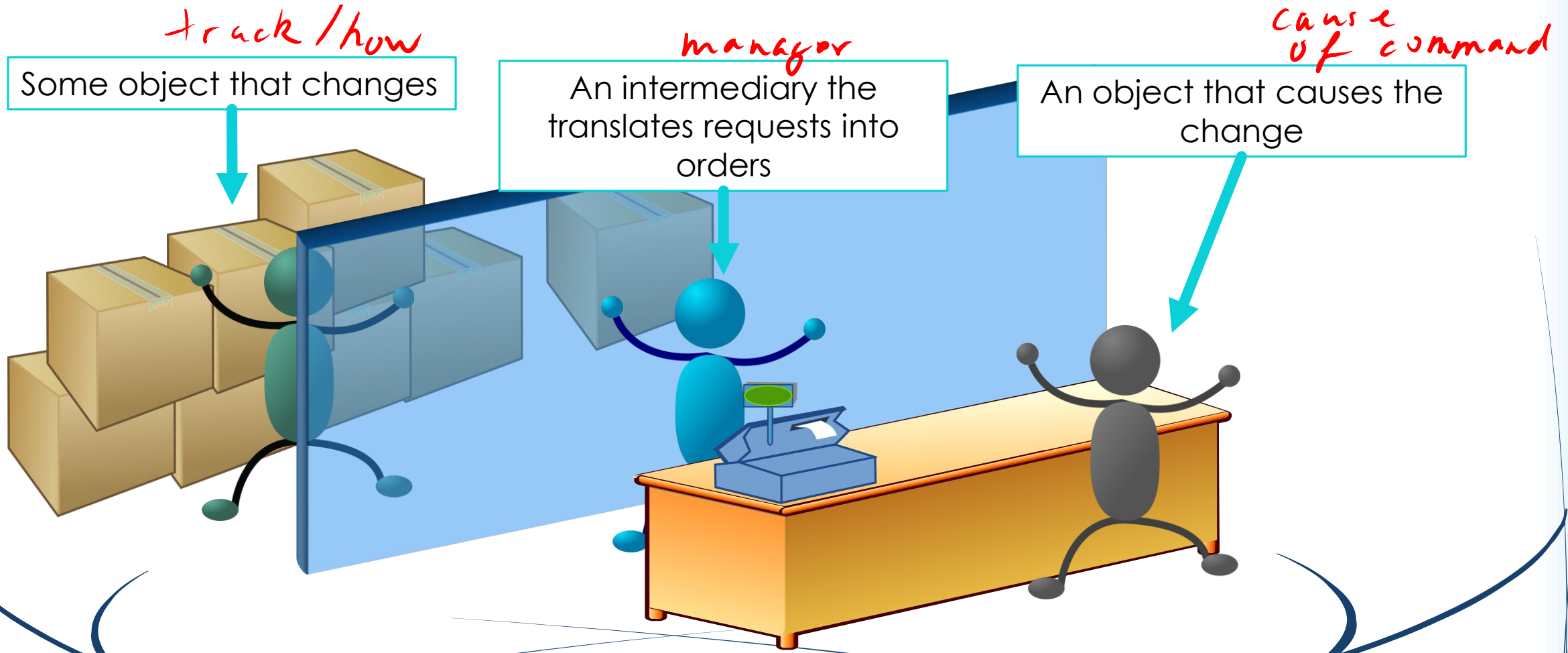
- This is a pretty major implementation detail
- If you don't design to have undo right away, it is a nightmare to add it later!

Two foundational code patterns

- Undo isn't a OOP pattern unto itself
- It is actually a subset of variants of
  - Command
  - Memento

Two foundational code patterns

- Both follow this paradigm



Two foundational code patterns

- The big difference in using them for an UNDO
  - 1) Memento stores a COPY to handle undo
  - 2) Command has an INVERSE function

could be a file

$f() \leftrightarrow f^{-1}()$

- Memento's original form is a bit closer to UNDO

Pattern: undo

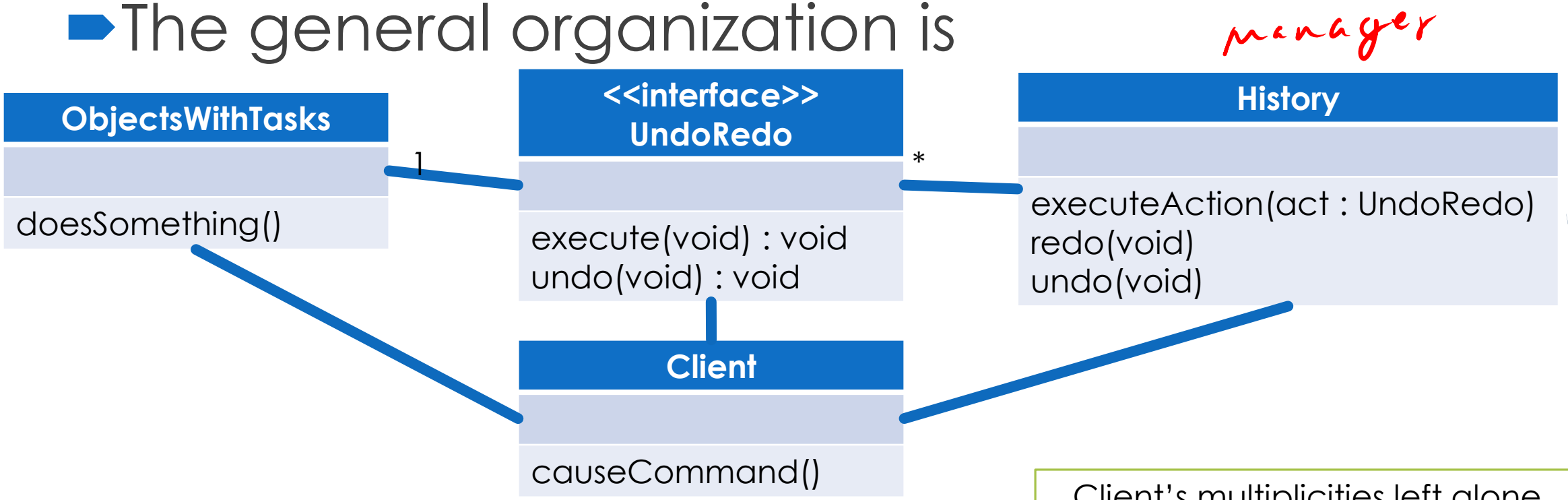
- Option 1 (Memento based): store a copy of the previous state
  - Advantages:
    - EASY to code
  - Disadvantages
    - Horrible memory hog

Pattern: undo

- Option 2 (Command based): have an inverse function
  - Advantages:
    - Flexibility
    - Less coupling of data and undo
    - Much lower memory requirements
  - Disadvantages
    - Usually adds another layer of OOP
    - Funding the inverse function isn't always an option
      - Particularly when using the additional layering

Pattern: undo

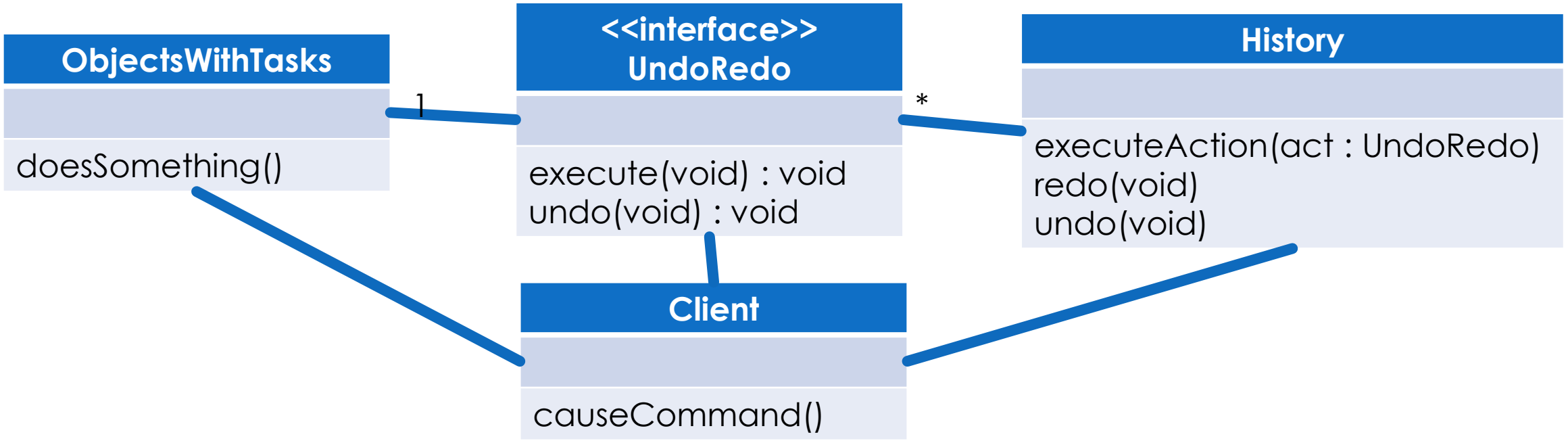
- The options can be mixed!
- The general organization is



Client's multiplicities left alone for now

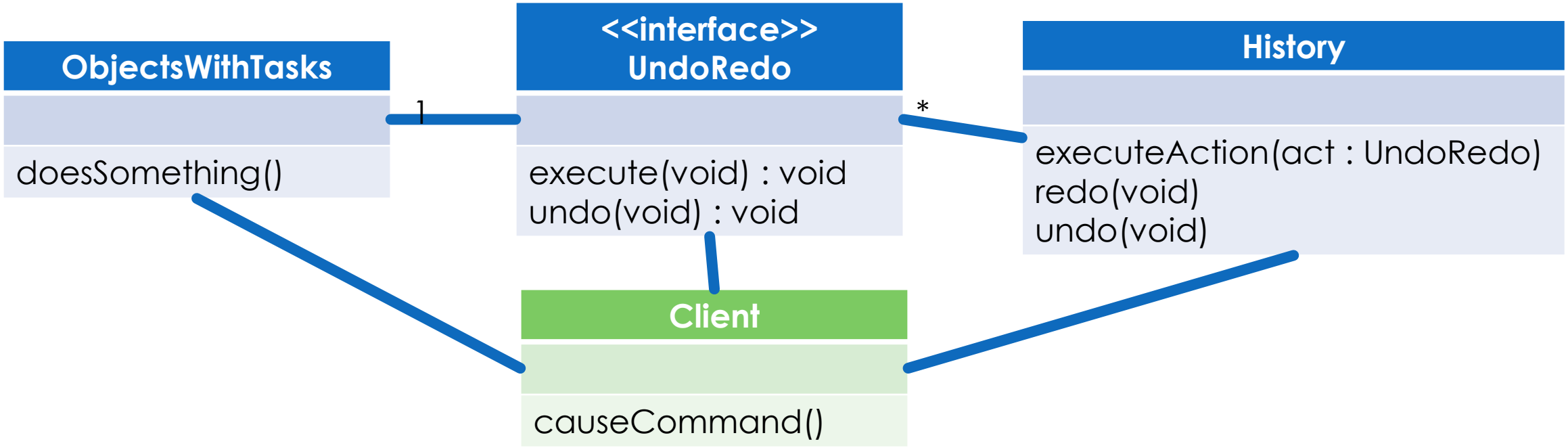
Pattern: undo

- So let's walkthrough an task, then an undo and redo



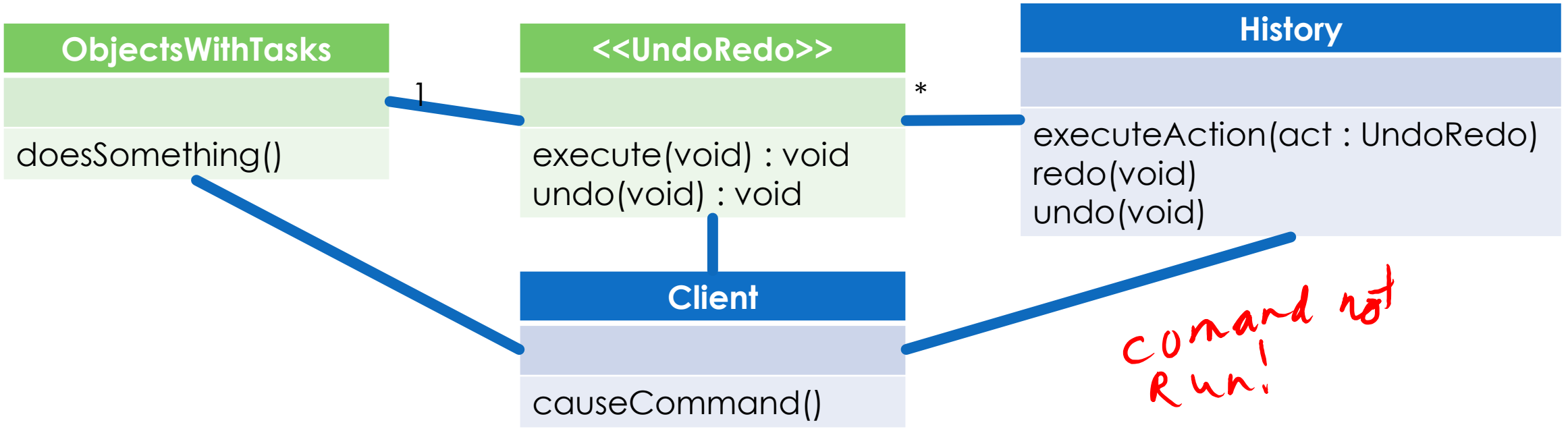
New Task

- Step 1: Trigger an undoable task



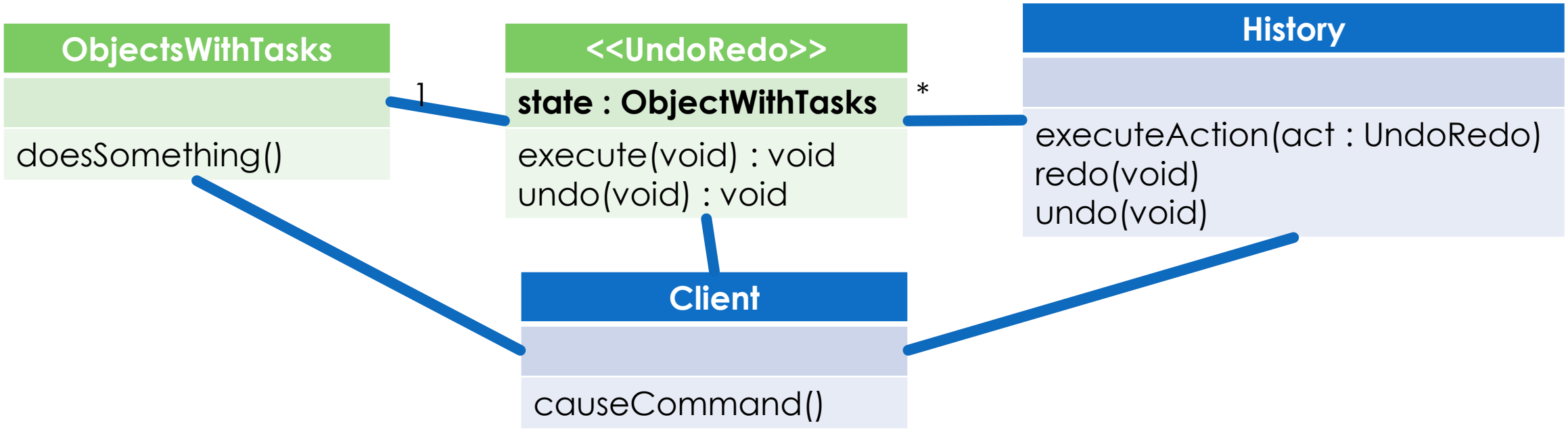
New Task

- Step 1a: Make a new UndoRedo using the ObjectWithTasks doesSomething (Command)



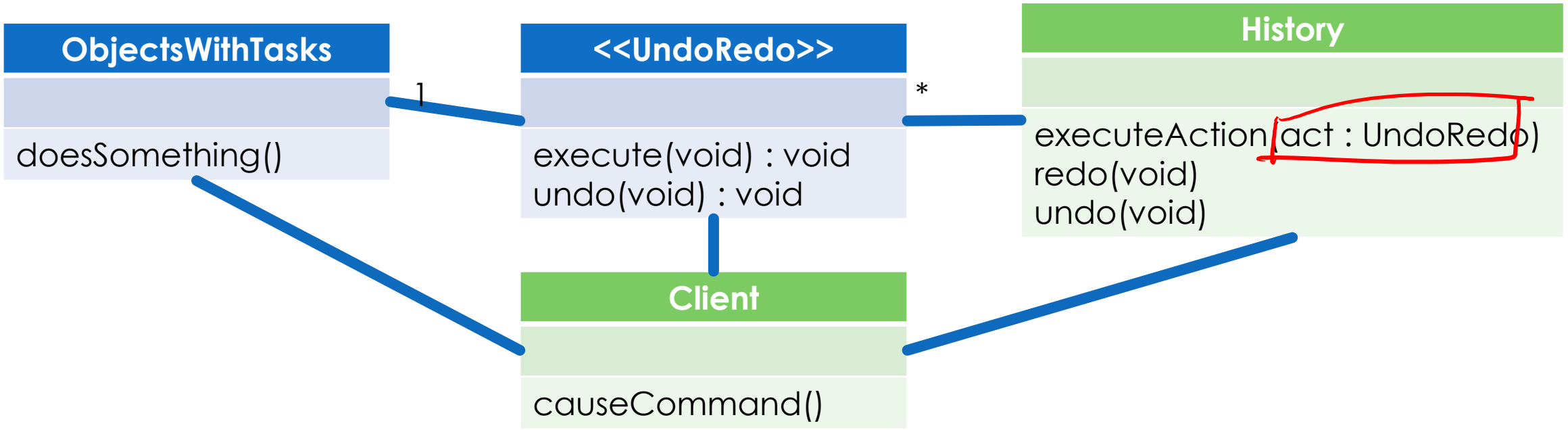
New Task

- Step 1b: OR Make a new UndoRedo using the ObjectWithTasks as state (Memento)



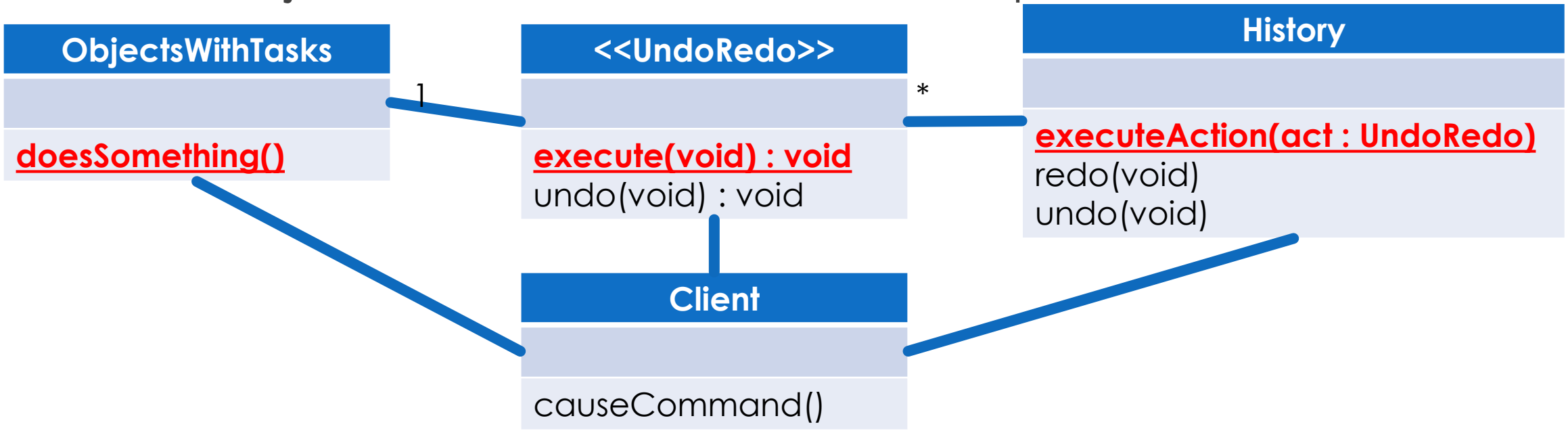
New Task

- Step 3: Add the new UndoRedo to History with executeAction()



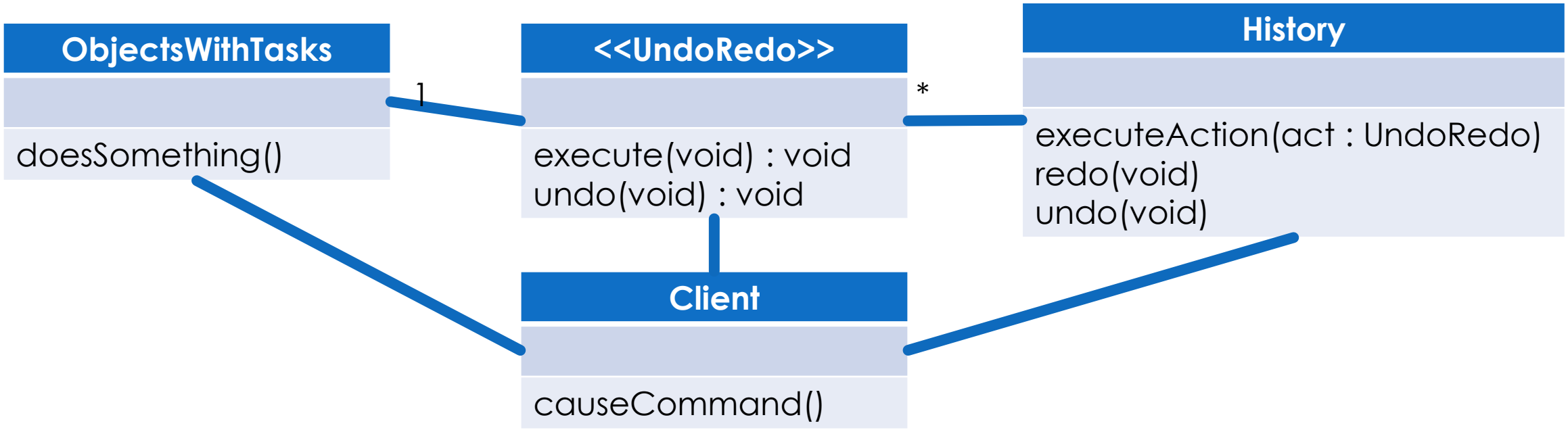
New Task

- Step 4: executeAction() adds act to its list (trimming redo section if needed), and call's act's execute(), which calls the ObjectWithTasks function to complete the task



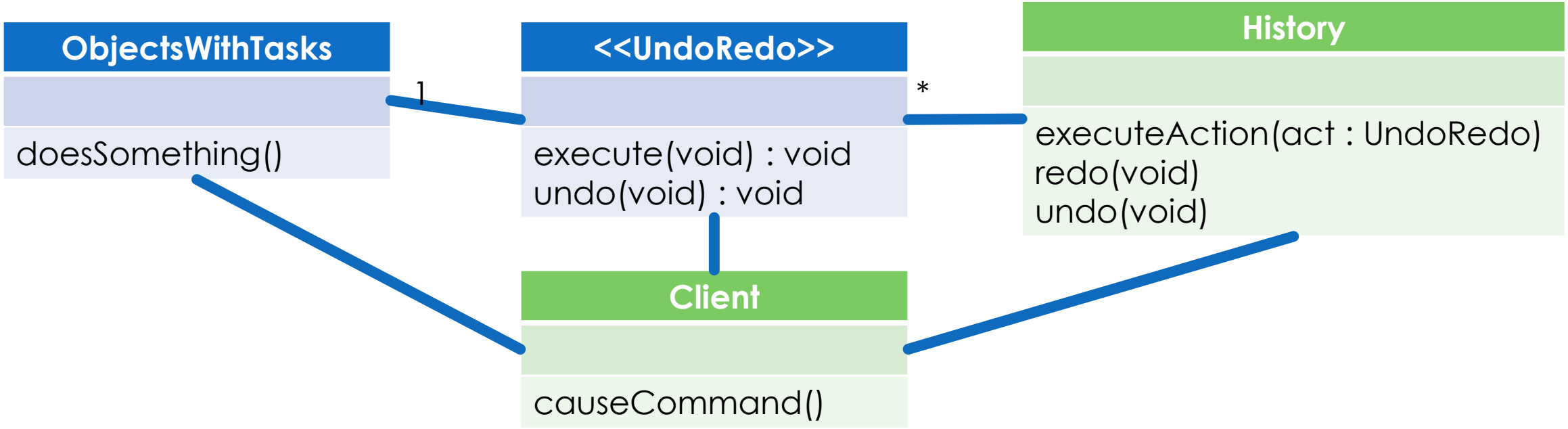
undo

- At this point, we have a command on our stack, and we can call undo()



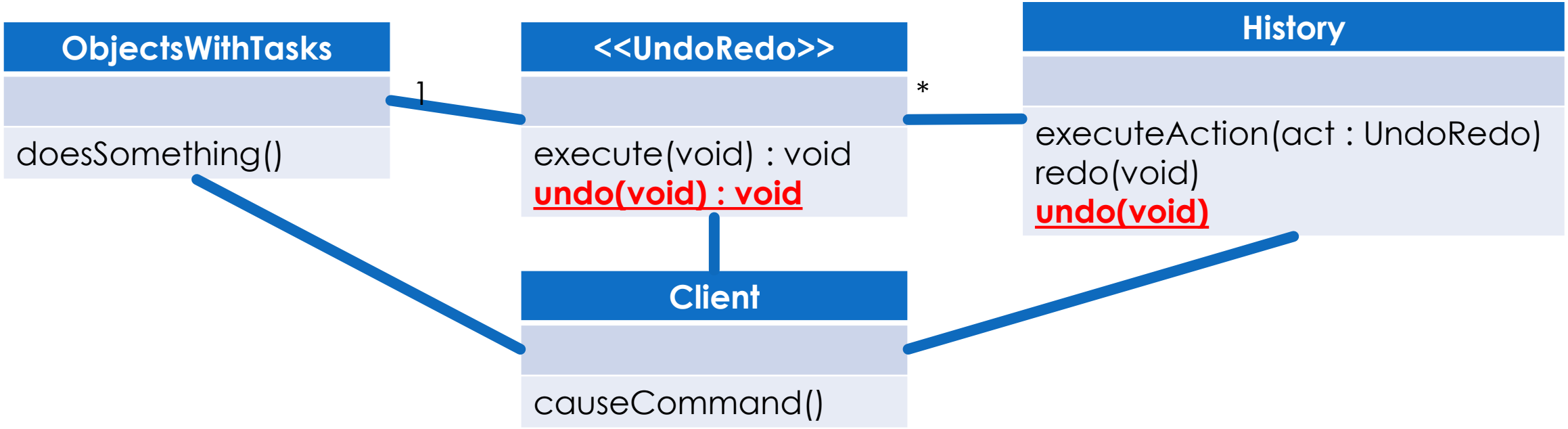
undo

- Step 1: the client calls history's undo



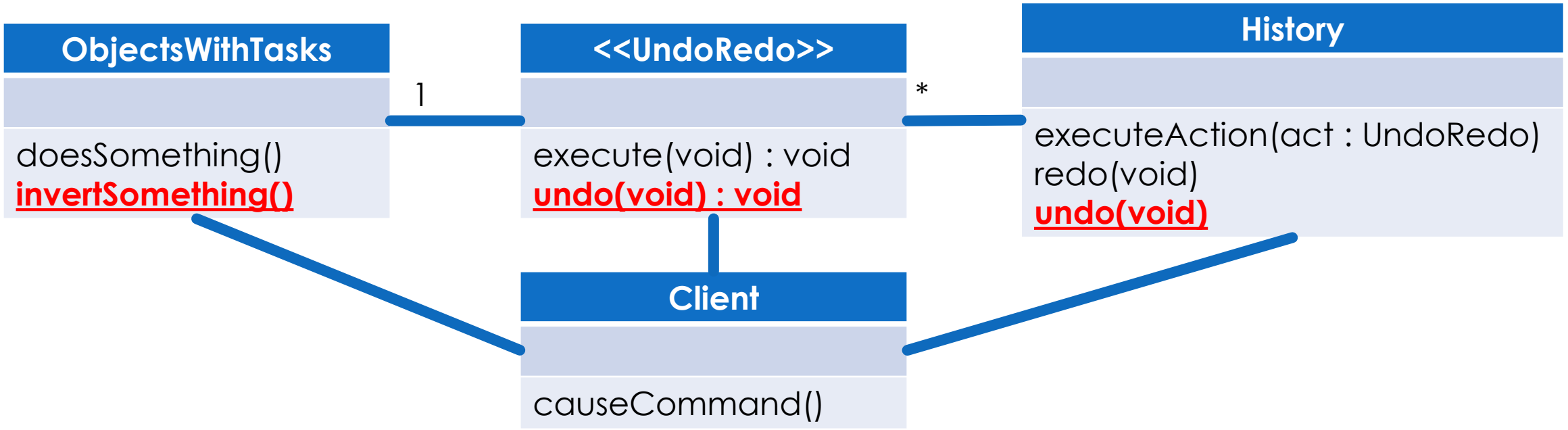
undo

- Step 2a: undo() calls act's undo(), which calls ObjectWithTasks to undo the task



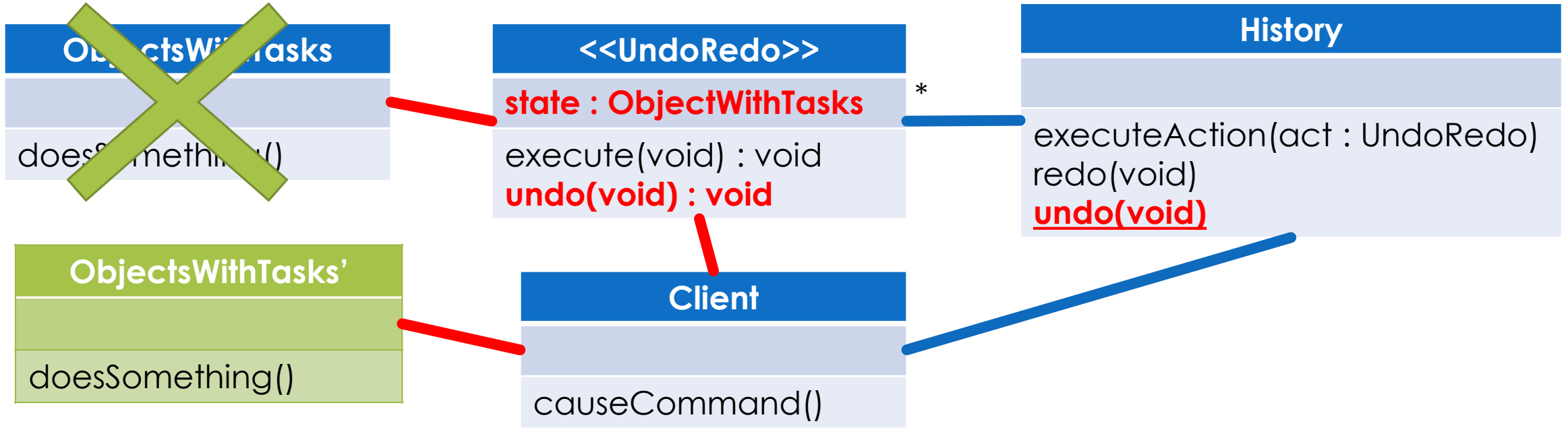
undo

- Step 2b: if necessary, ObjectWithTask may require an invertSomething()



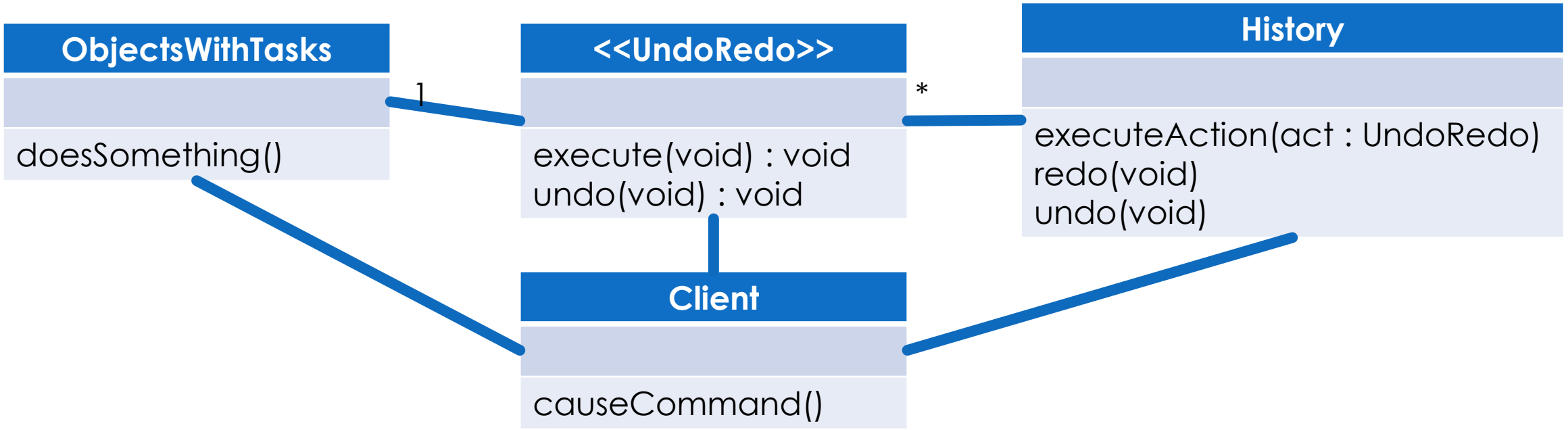
undo

- Step 2c: if the Momento version is used, Object with tasks is replaces with a stored version



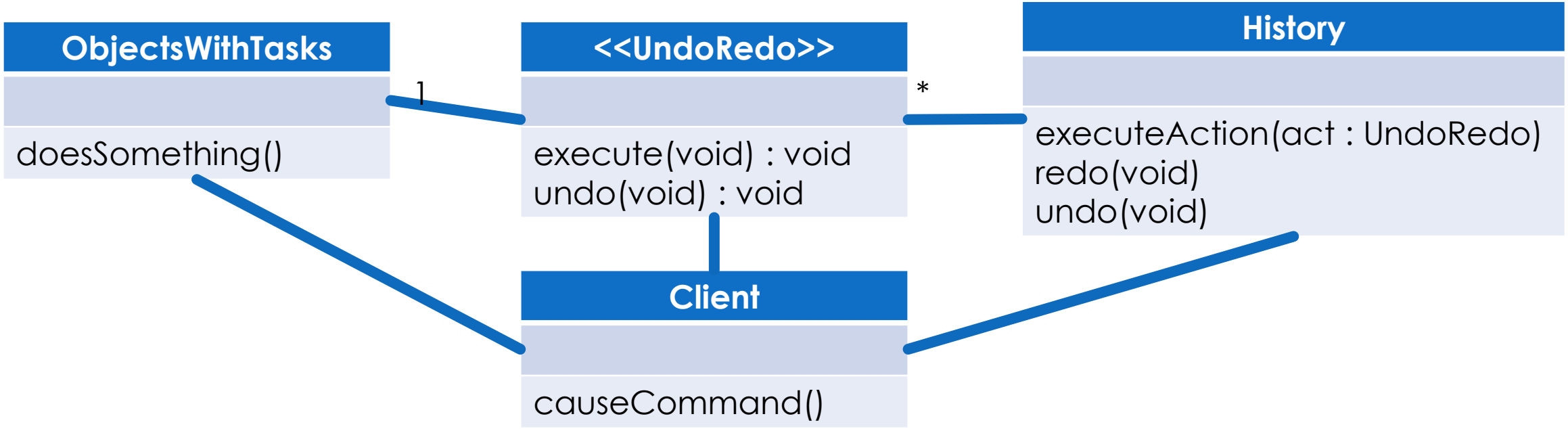
Undo

- Step 3: update index into the history list



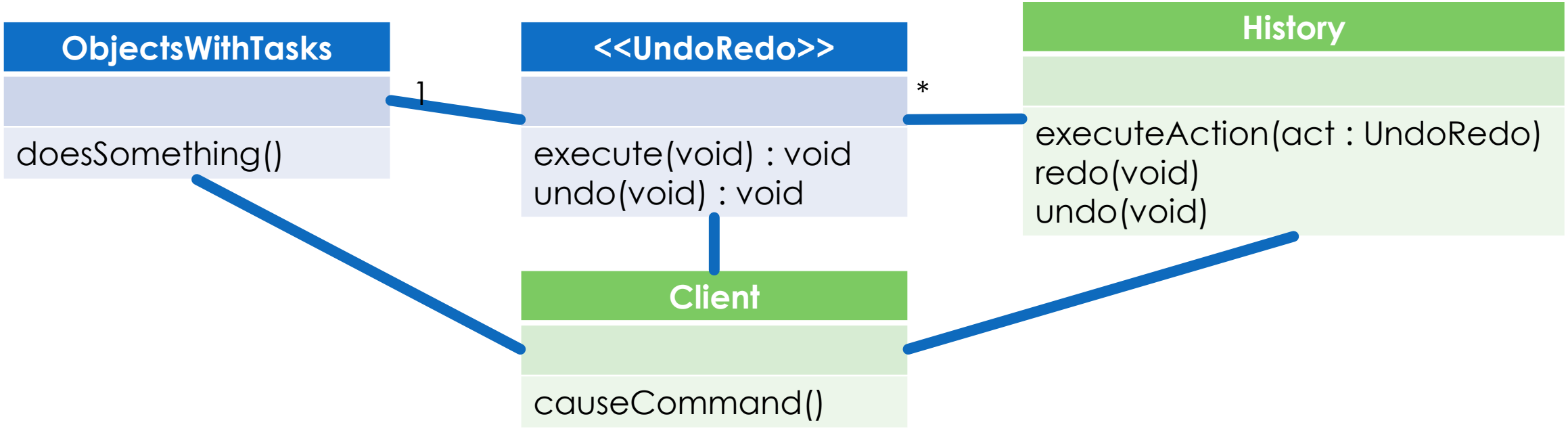
Redo

- We are now ready for redo. Redo is a mix of undo() and executeAction()



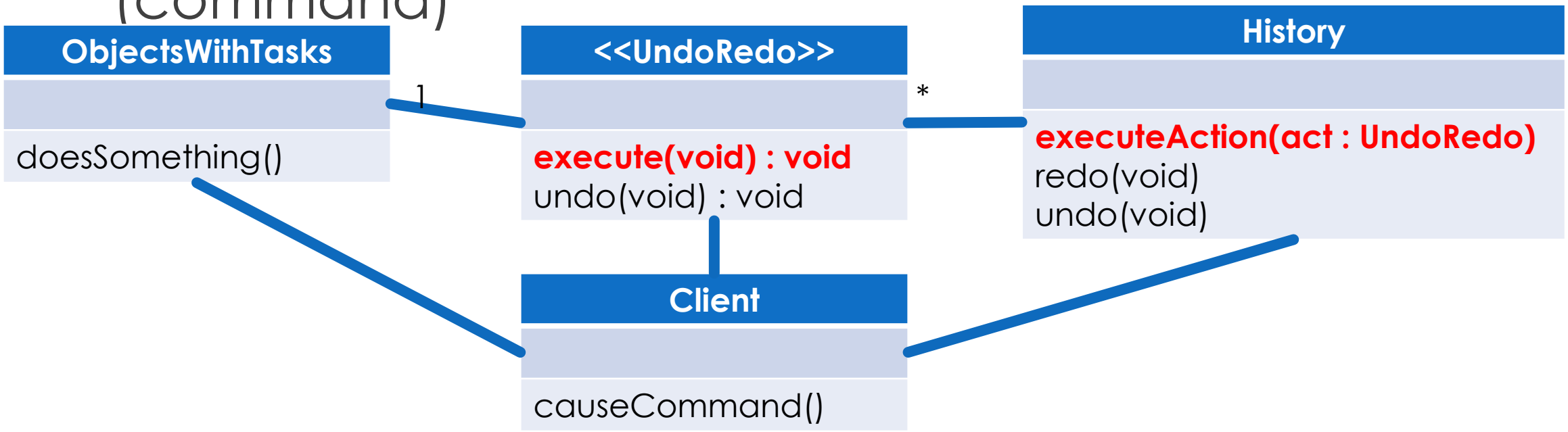
Redo

- Step 1) call history's redo()



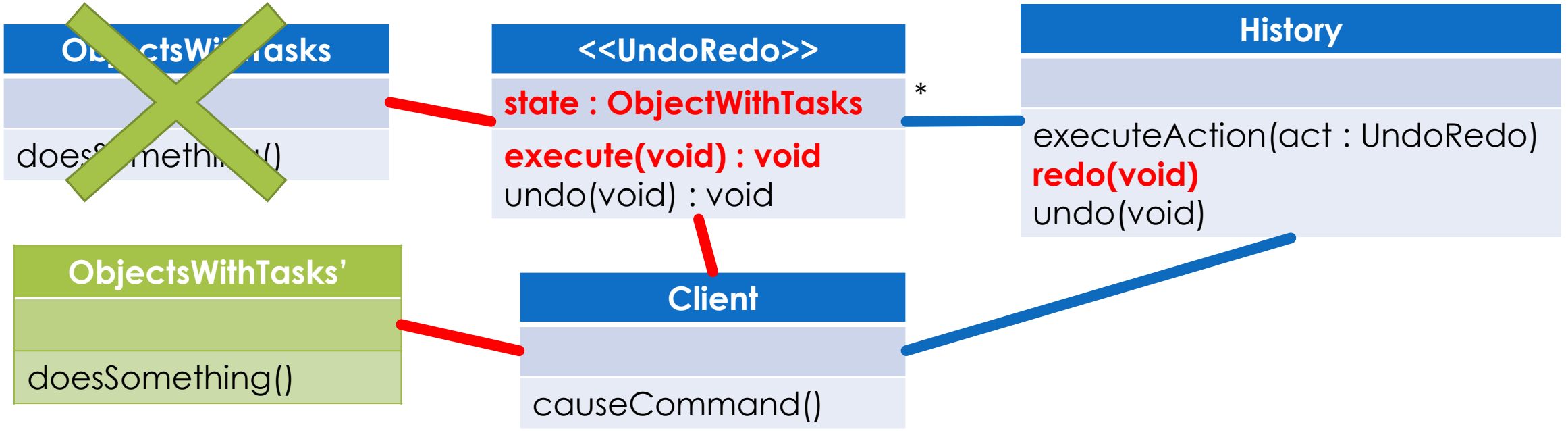
Redo

- Step 2a) history's redo() calls act's execute, and then ObjectWithTasks associated function (command)



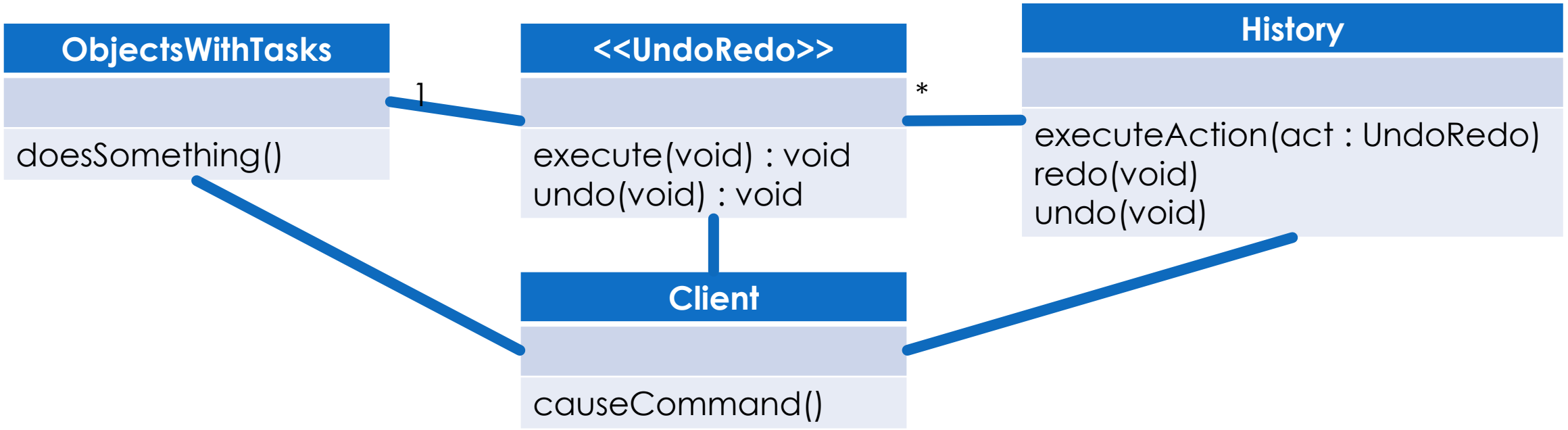
Redo

- Step 2b) history's redo() calls act's redo, and then replaces ObjectWithTasks with the stored state



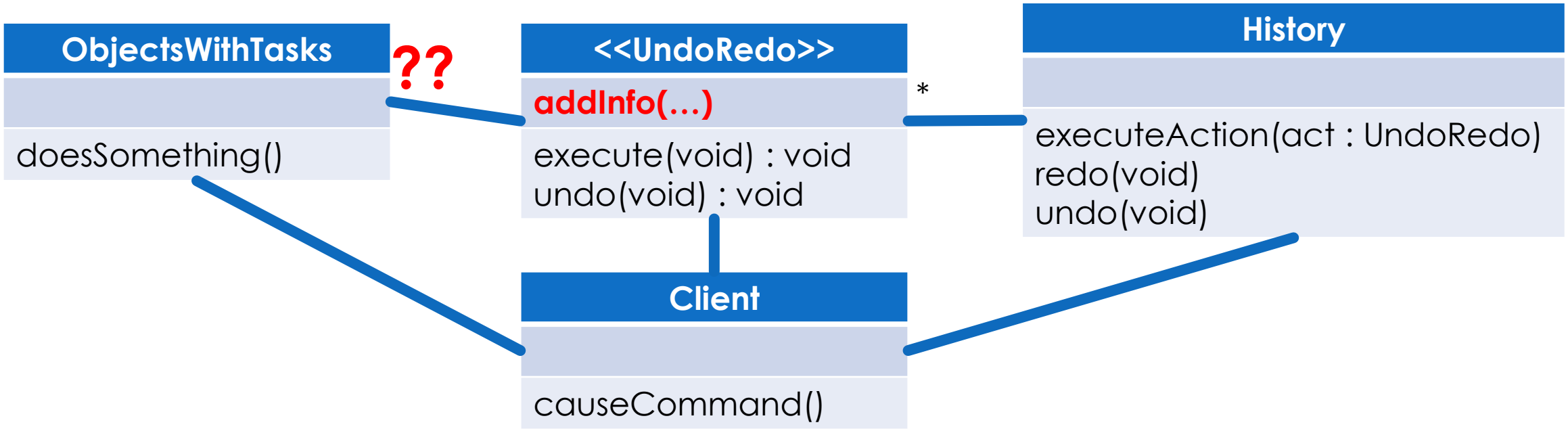
Redo

- Step 3) update the index in to history's list



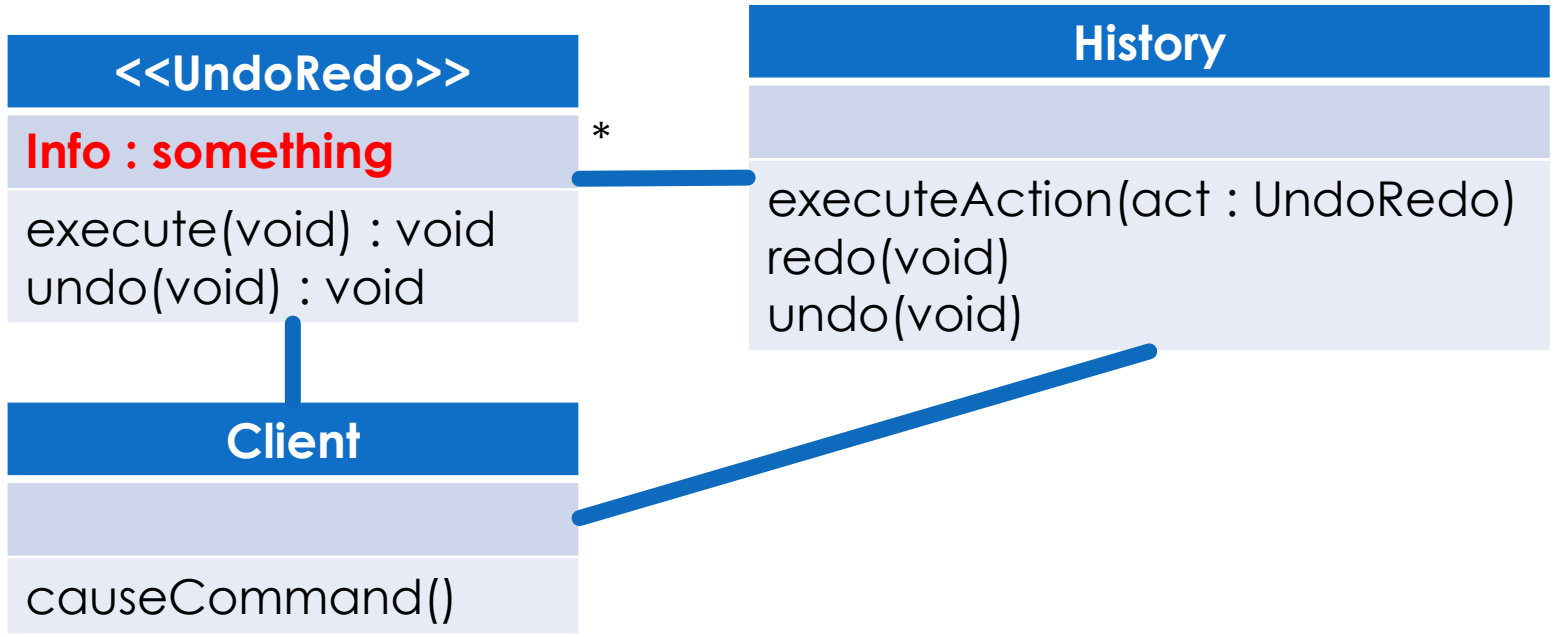
Variant

- Most of the variant are with how many Objects are involved, and how to store sufficient information to undo and redo. These must be adjusted for the type of tasks and data being operated on



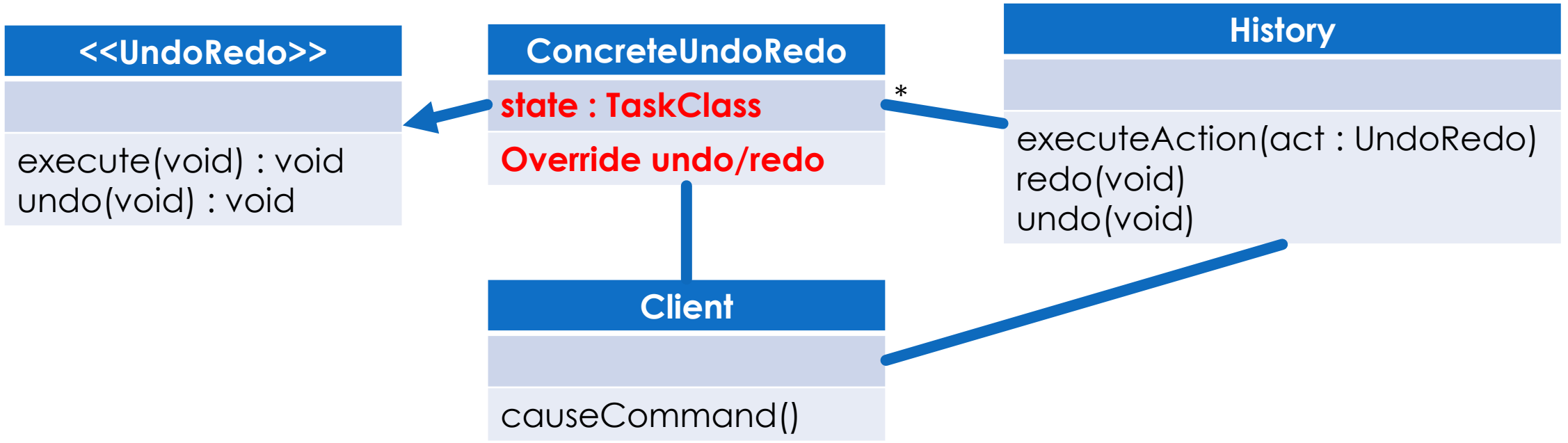
Variant

- Another common variant is that for small tasks, the Object and UndoRedo can be combined.



Variant

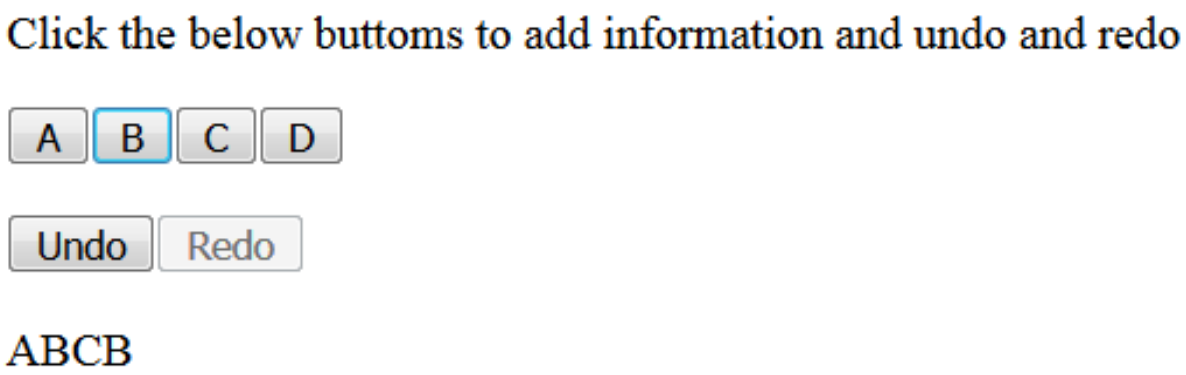
- Yet another common variant is to send in the task class, and call it's functions directly inside undo and redo



JavaScript Example

JavaScript Example

- I'm using the combined Object and Undo variant for this example.



JavaScript Example

- First, I need a "class" to manage my stack

```
function History() {
    var commands =[];
    var index = 0

    this.executeAction = function(cmd){
        commands.length = index;
        commands.push(cmd);
        index = commands.length
        cmd.exec();
        }

…
```

JavaScript Example

- First, I need a "class" to manage my stack

```
…

        this.undoCmd = function(){
                if(index > 0) {
                        var cmd = commands[index-1];
                        cmd.undo();
                        index= index - 1;
                }
        }

        this.redoCmd = function(){
                if(index < commands.length) {
                        var cmd = commands[index];
                        cmd.exec();
                        index = index + 1;
                }
        }

}
```

JavaScript Example

- UndoRedo is a small class
- In a “split” example, I could either send in a class with a guaranteed function or this would forward the call onto the task class

```
function UndoRedo(letter){
	this.letter = letter

	this.exec = function(){
		var out = document.getElementById("result")
		out.innerHTML += letter

	}

	this.undo = function(){
		var out = document.getElementById("result")
		out.innerHTML = out.innerHTML.slice(0,-1)
	}
}
```

JavaScript Example

- Next, I associate the event with the command
- I also have the event make a new UndoRedo and add it to the history.
- Reminder: executeAction() will run the command and add it to the list

```
//in window.onload
document.getElementById("buttonA").onclick =
        letterEvent;
…


var hist = new History();

//map UndoRedos onto buttons
function letterEvent(event) {
        if( event.target.id == "buttonA" )
                hist.executeAction(new UndoRedo("A"))
        …
}
```

JavaScript Example

Setting up undo and redo are just associated the function with the event!

Other than updating the redo/undo UI, we are good to go!

```
document.getElementById("undo").onclick =
hist.undoCmd;

document.getElementById("redo").onclick =
hist.redoCmd;
```