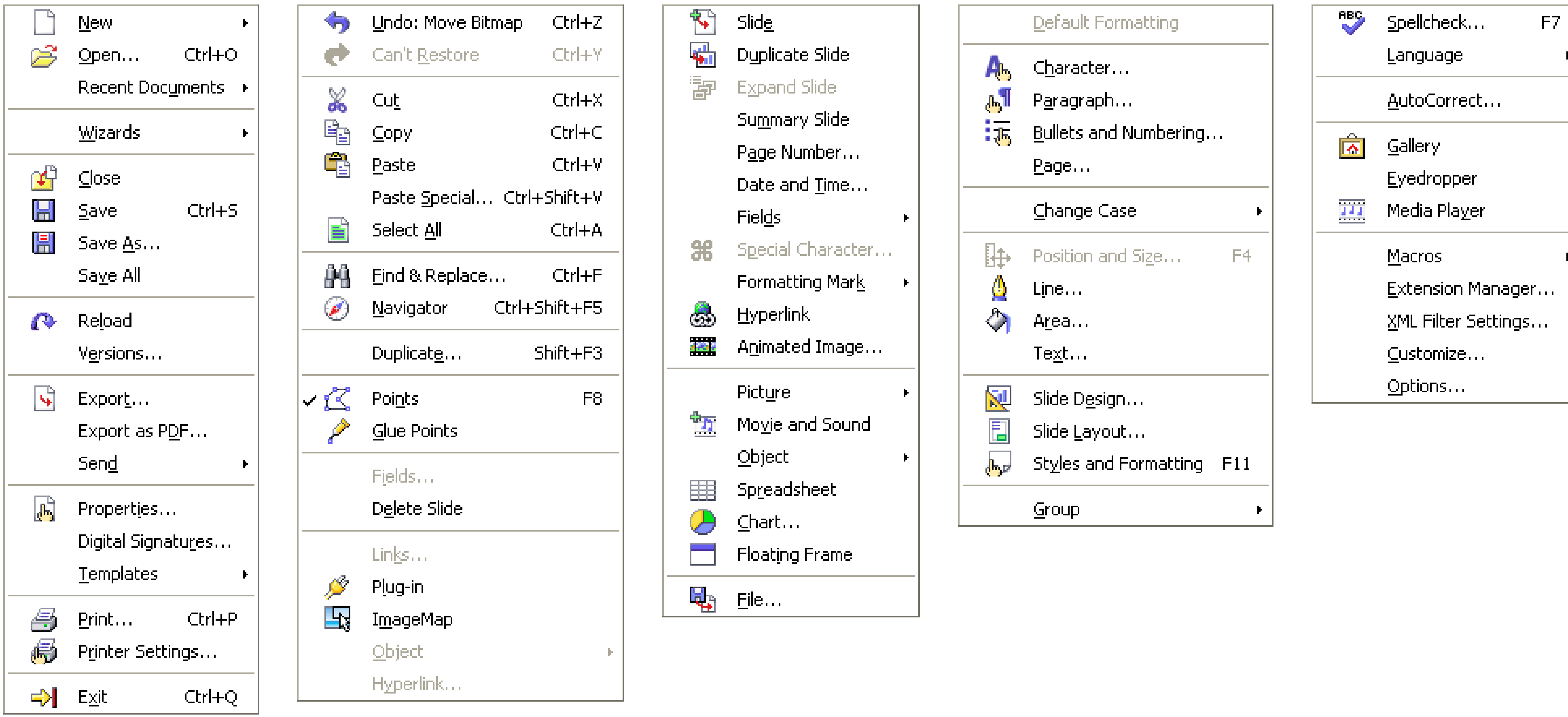
Comp-304 : Command
Lecture 22

Alexandre Denault
Original notes by Marc Provost
and Hans Vangheluwe
Computer Science
McGill University
Fall 2007

Classic Example



Problem

- User interface toolkit includes buttons and menus that carry out a request corresponding to user input.
- The buttons and menus can't explicitly implement the action, because only an application knows what should be done on which object.
  - GUIs only provide a button construct. It has no behavior.
  - It's up to the programmer to give the button a behavior.
- How do we encapsulate behavior?

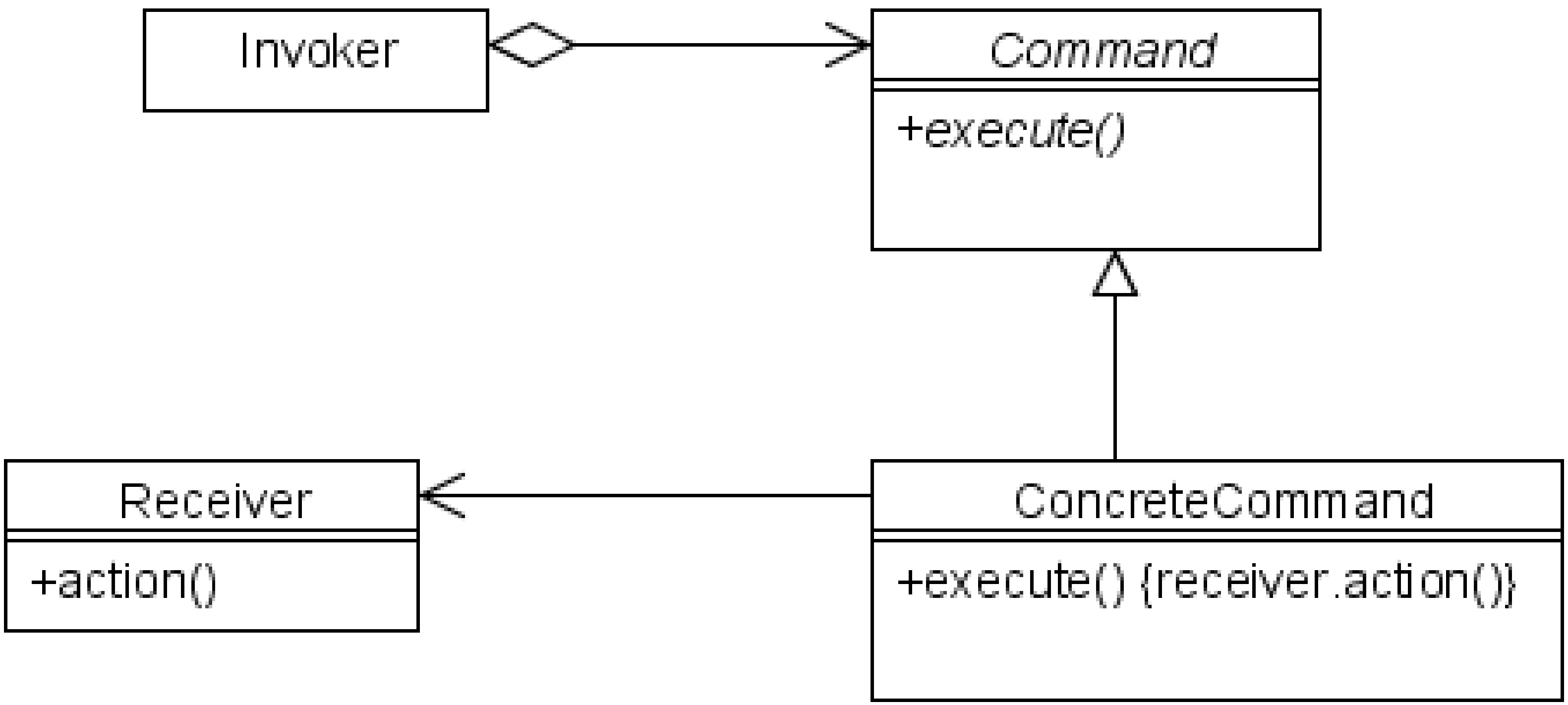
Command Pattern

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

Motivation

- Separates an operation from the object that executes it.
- With the Command Pattern, it is possible to parametrize an object with an operation.
- Support undo/redo
- Possible to execute the request at a different time. By passing the command object to another process.

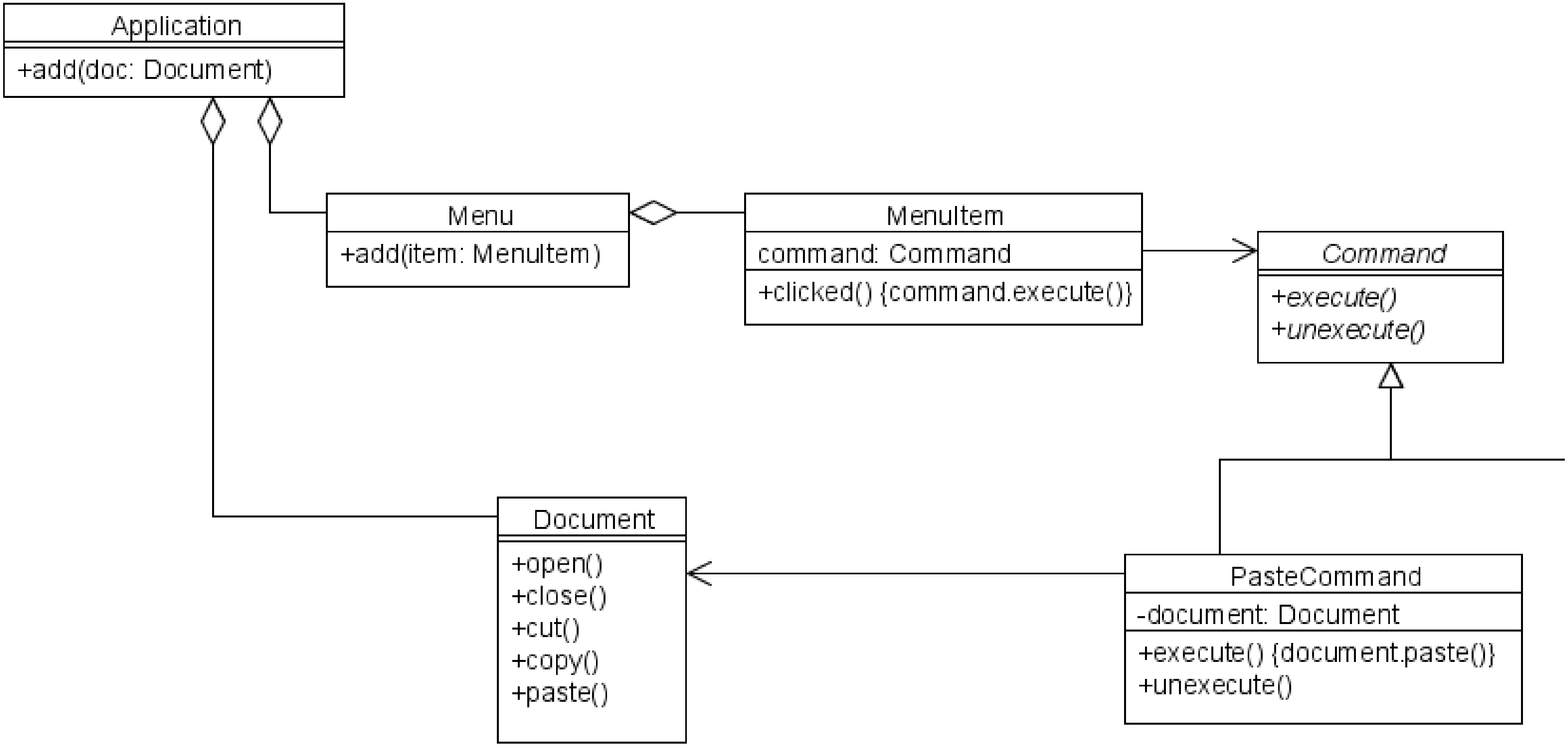
Participants



Why?

- Each item in the menu is conceptually the same object.
- The only difference is with the action that is taken when pressed.
- Solution: parametrize the menu item object with a command.

Class Diagram of Example



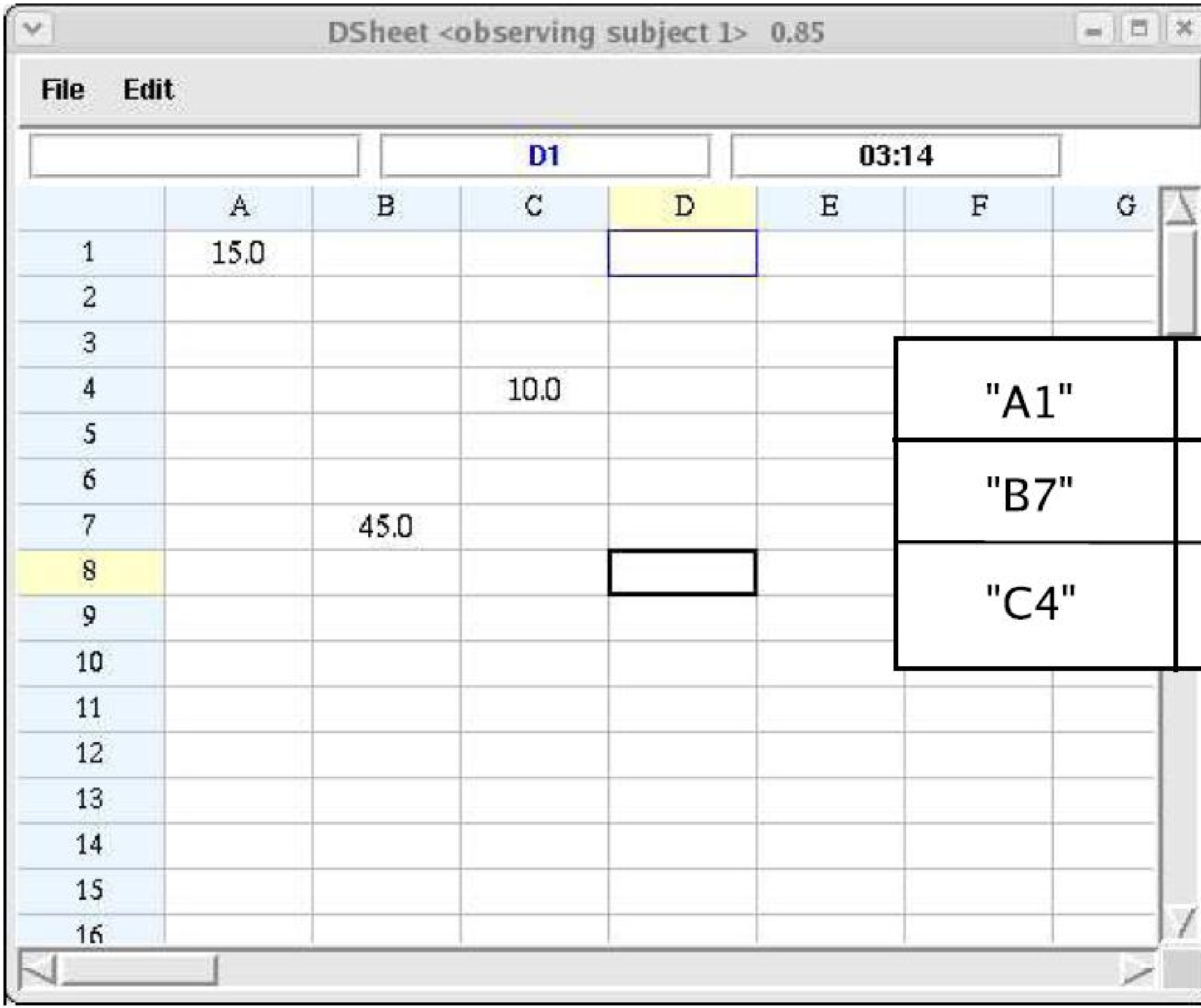
Supporting Undo/Redo

- Since a command is an object, it can hold a state.
- A command object could store the information required to undo or redo itself.
- Use an history list of commands objects.

Supporting Undo/Redo

- Each command should know how to undo and redo itself (one level).
- A command manager hold the history list of commands:
  - [commandA; commandB; commandC; :::]
  - Moving backward: undoing commands
  - Moving forward: redoing commands
- Let's go over an example...

Example

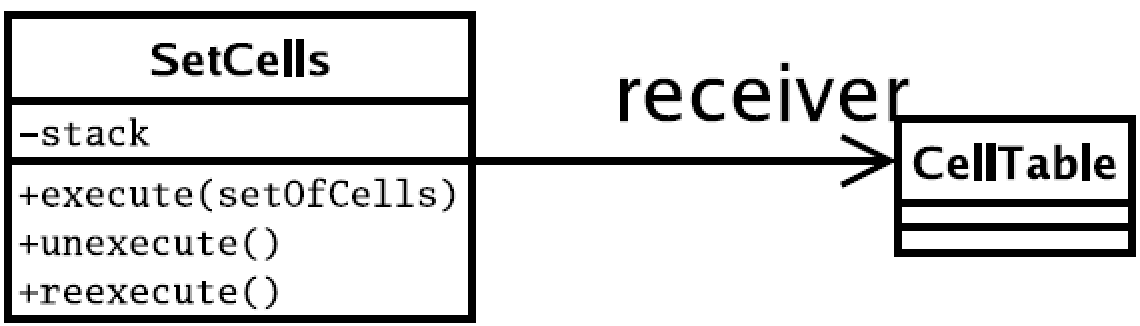


HashTable:

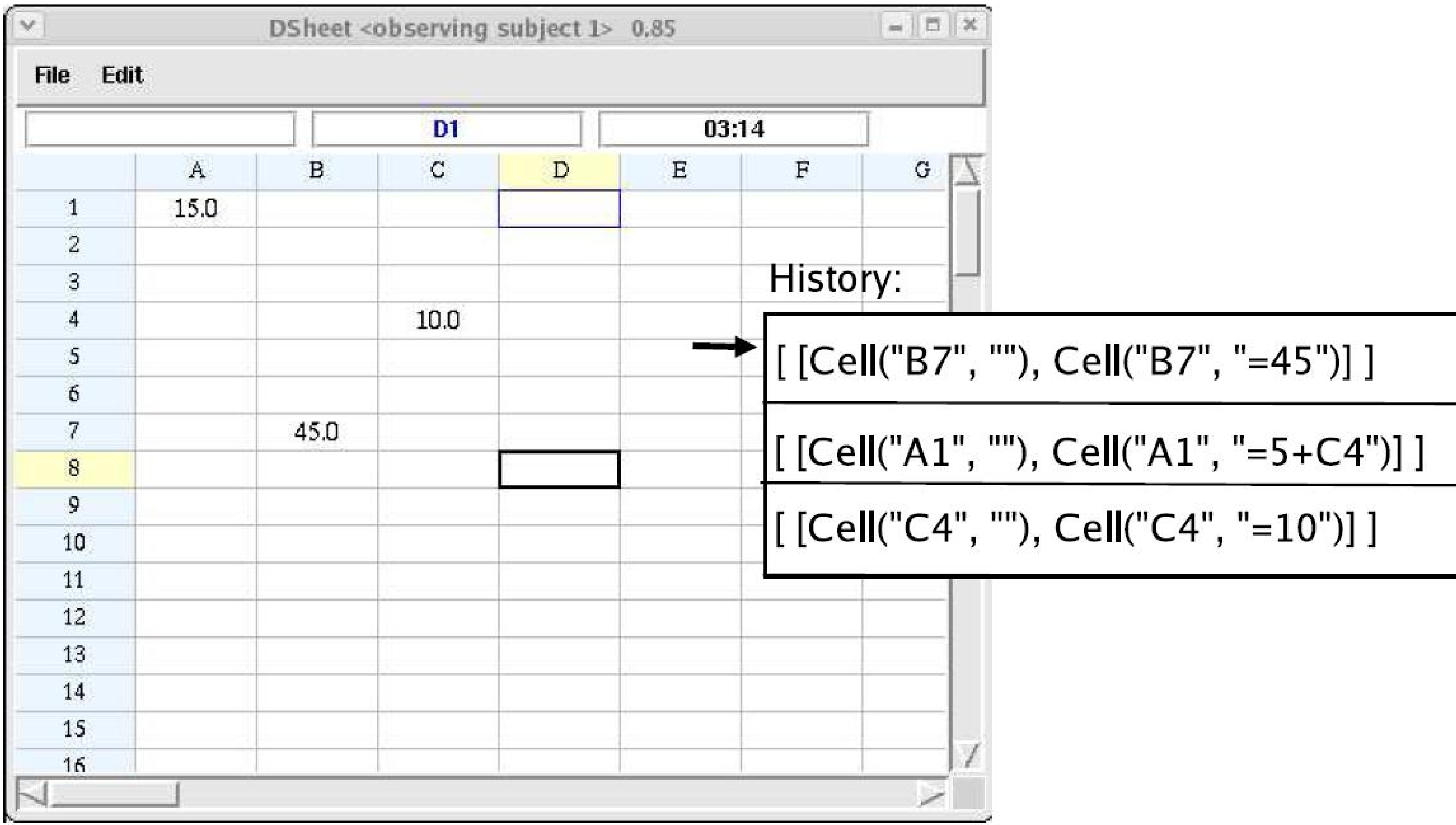
"A1"	Cell("A1", "=5+C4")
"B7"	Cell("B7", "=45")
"C4"	Cell("C4", "=10")

SetCells Command

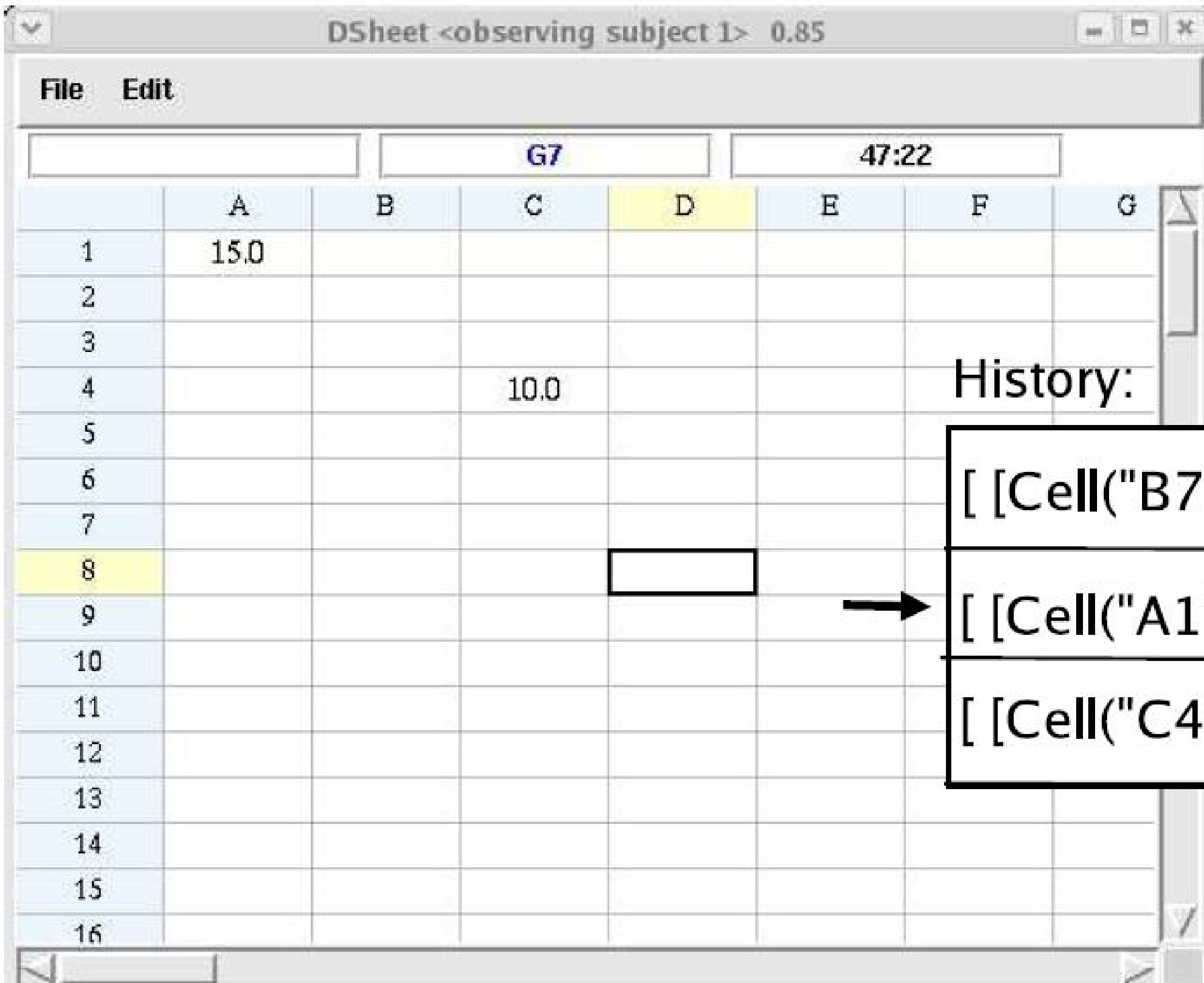
- 'SetCells' command, which acting on the previous hashtable is used to support undo/redo
- The history list is stored directly in the setcells command. (Unique command)
- Each time a set of cells is modified by the user, the difference between the previous state and the next state is added in the history list.



Example (cont.)



Undo



History:

[[Cell("B7", ""), Cell("B7", "=45")]]

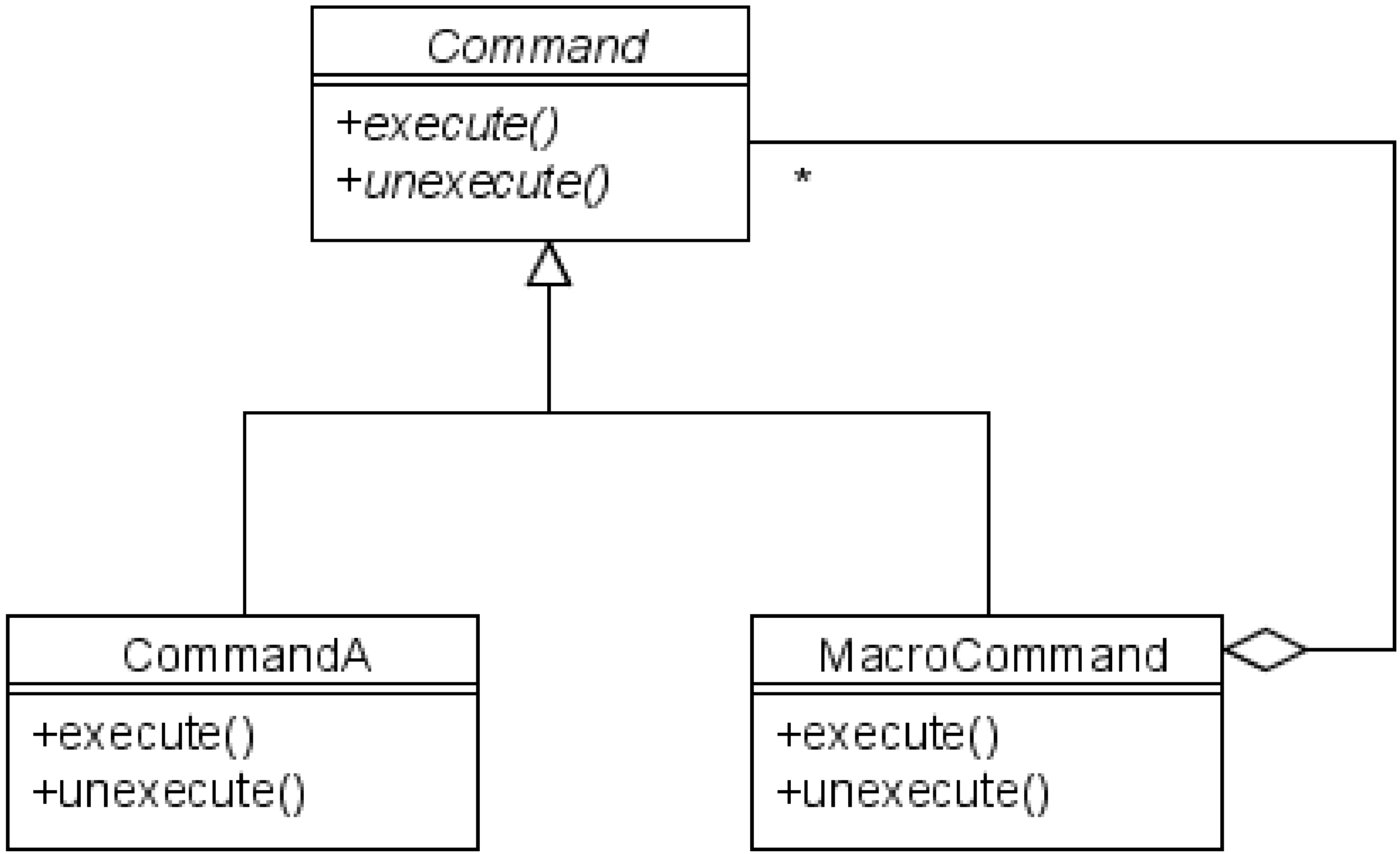
→ [[Cell("A1", ""), Cell("A1", "=5+C4")]]

[[Cell("C4", ""), Cell("C4", "=10")]]

Consequences

- Decoupling of the command and the invoker.
- Commands are objects. They can be manipulated and sub-classed like any other object.
- You can assemble commands into composites of commands.
- New commands is easy and does not require the modification of existing code.

Hierarchy in Commands (Macro)



How else can it be used?

- Transactional Behavior
- Action Queuing / Progress Monitoring (bar)
- Thread pools
- Macro Recording
- Networking / Dsitributed Actions

Other concerns

- Error accumulation in Undo/Redo.
- How smart should a command be?

Another Example: Path Finding

- The Path Finding system in Mammoth uses a variation of the command pattern.
- Imagine a game server that houses hundreds of NPC artificial intelligence agents.
  - Each of these agents are moving independently
  - Each of these agents are sending requests to the path finding engine.

Path Finding Engine?

- The Path Finding Engine is the components that allows a player to go from point A to B, avoiding obstacles.
- This is very CPU expensive.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

So what is so special?

- On a Mammoth server, you have multiple Path Finding coming in at the same time.
- The server cannot stop to process each of them.
  - The regular operations of the server is time critical.
  - Spawning a new thread for each requests would flood the system.

Flooded with Threads?

- This is not a new problem.
- Web servers and application servers typically have this problem.
  - They received a large number of simultaneous requests.
- The solve this problem by using a thread pool.

Thread Pool Pattern

- Although not an official pattern, a thread pool is a commonly used pattern to solve problems dealing will multiple simultaneous incoming requests.
- A thread pool is a collection of threads.
  - Requests to the thread pool are queued.
  - When a thread is available, a request is send to it.
  - The request will then run on that thread.
  - The response to the request is sent asynchronously.

Benefits

- Requests are run asynchronously at a controlled rate.
  - You never have more than N requests processed at one time.
  - You don't lose the requests you can't deal with.
  - The system is not adversely affected by a number of requests.
  - Since N is a fix number, you can play with the number of threads.

Disavantage

- Requests are run asynchronously at a controlled rate.
  - Your system is now asynchronous.
  - In cases of high demands, it might take a while until you get a response.

Back to Path Finding

- Path Finding requests are queued, then sent to the Thread Pool.
- Since the requests are objects, this is easy to do.
  - Objects can be queued.
  - Objects can be passed as parameters.
- When a path finding request is sent, a Path object is sent back as a response.
- When the request is executed, the path is slowly inserted inside the Path object.
- Path Finding requests also have a cancel() method.
  - If a player decides to go elsewhere, we should stop the request.