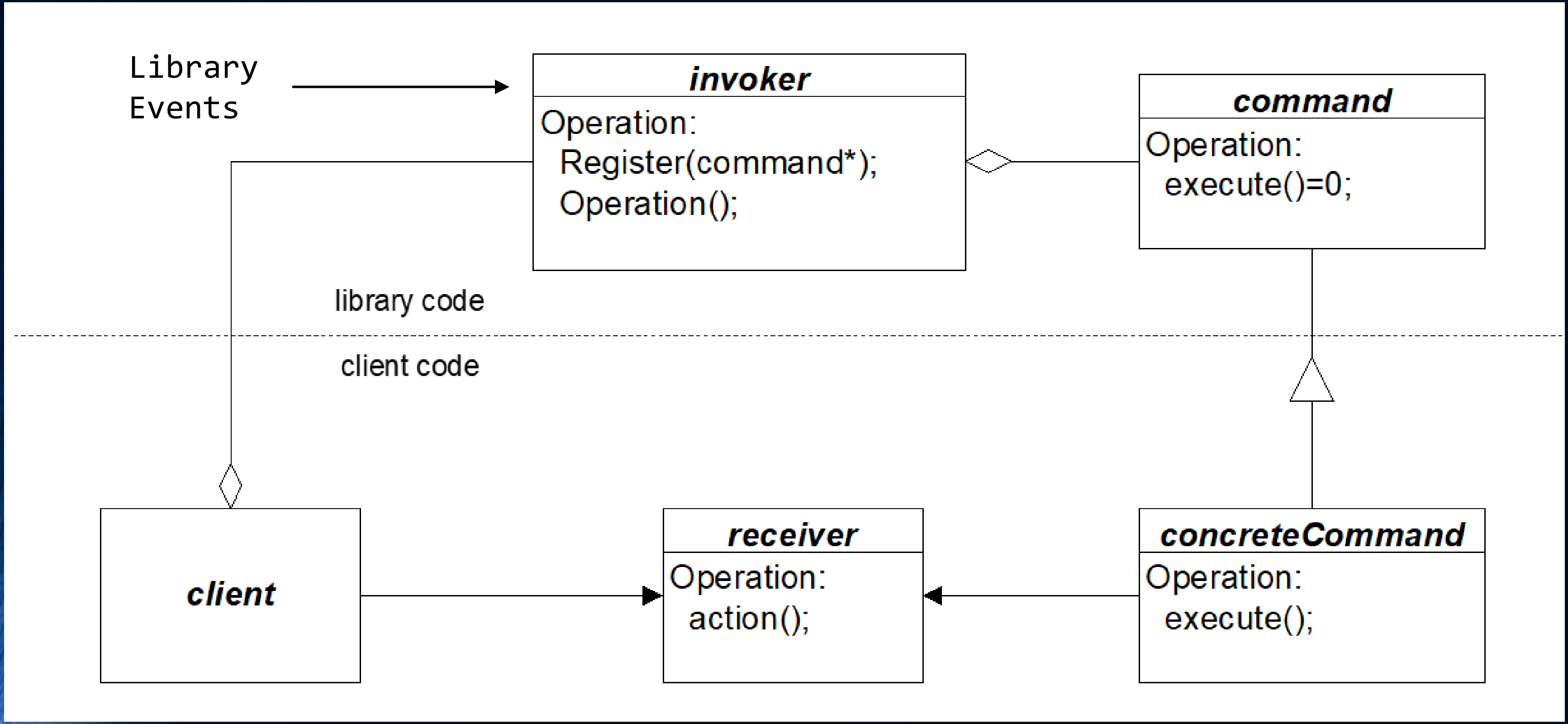
Command Pattern

Jim Fawcett
CSE776 – Design Patterns
Fall 2014


Library Events
invoker
Operation:
Register(command*);
Operation();
command
Operation:
execute()=0;
library code
client code
client
receiver
Operation:
action();
concreteCommand
Operation:
execute();

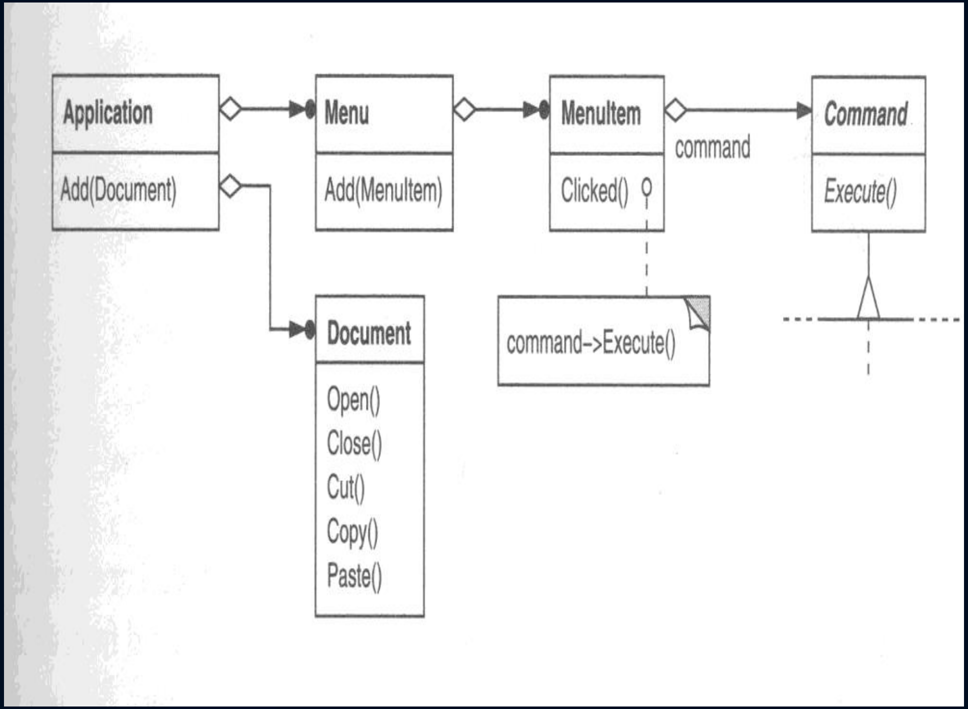
Command Pattern Intent

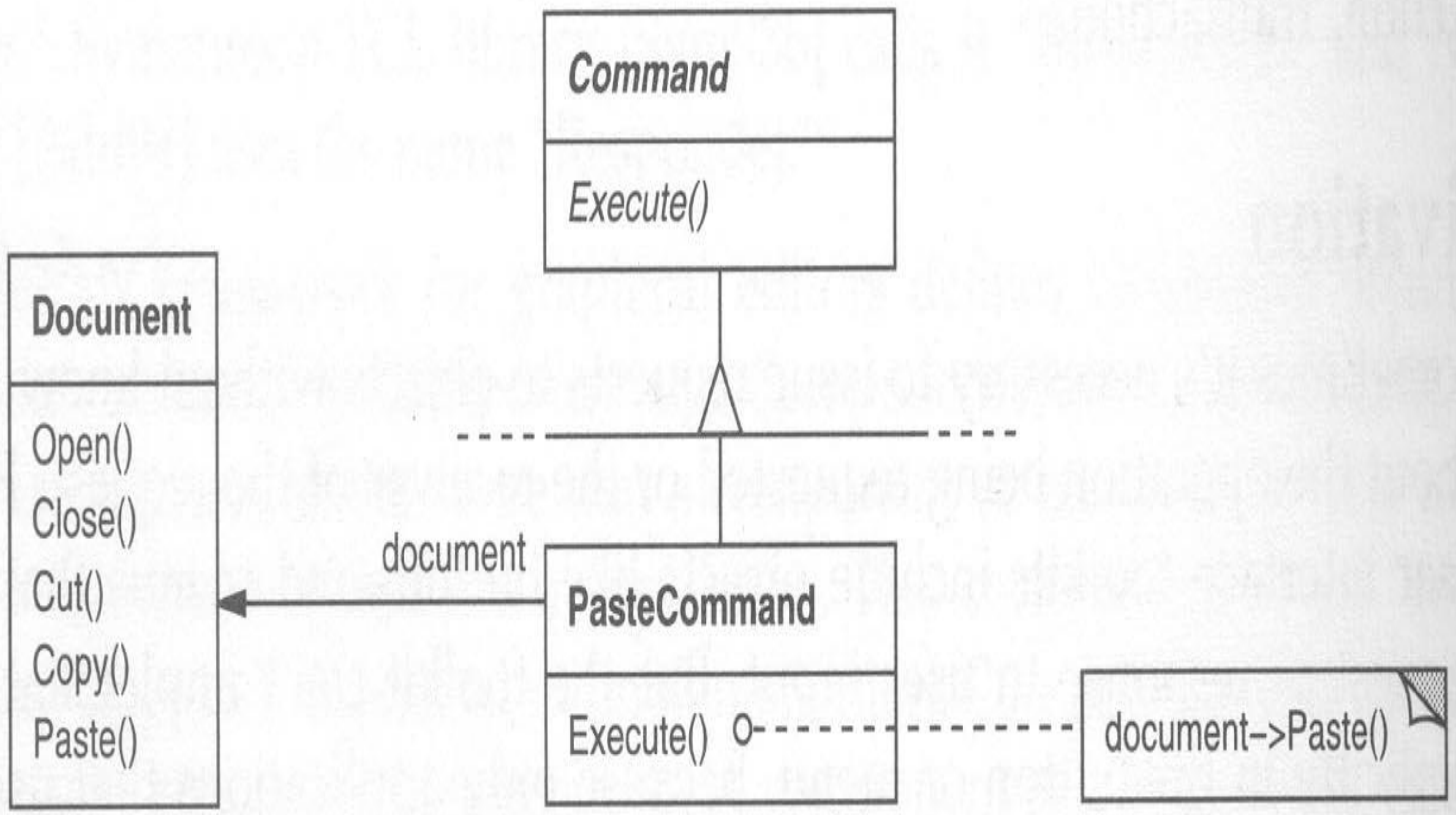
A command encapsulates a request as an object. A reference to the command is given to an invoker for later invocation.

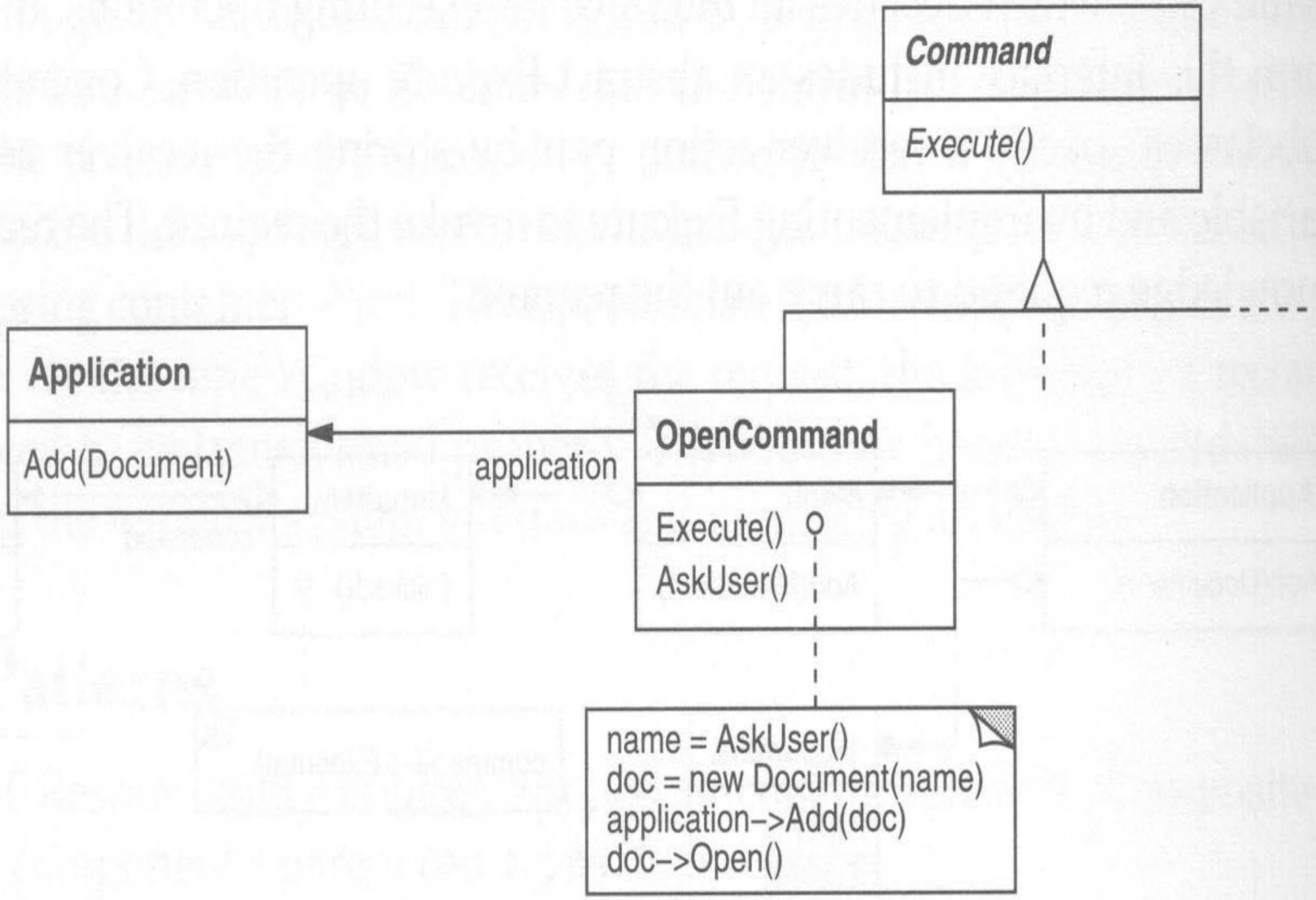
- Intent:
  - decouple the event triggering a command from the processing associated with the command.
  - The invoker of a command knows about the trigger event but does not need to know anything about command's processing. The creator of the command class knows about the processing but nothing about the invoker's event.
  - With commands, you can control their selection, sequencing, queue them, undo them, and otherwise manipulate them.
  - Commands are an object-oriented replacement for function pointers.

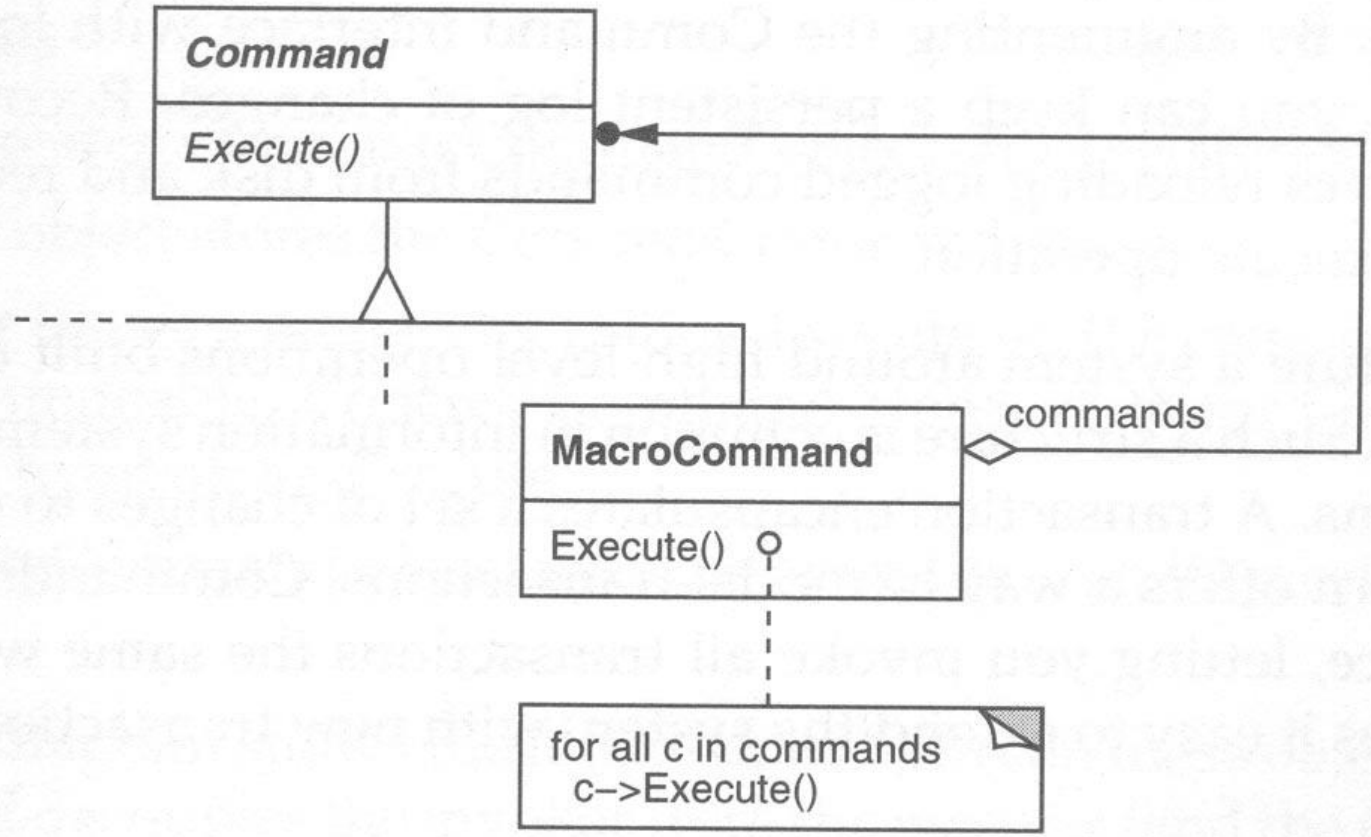
Motivation

- Command addresses the need to issue requests to objects without knowing anything about the objects themselves.
- At instantiation, the Command is given any information it will need to later carry out its task.
- The actual order to carry out that request is given at a later time.
- The key to this pattern is an abstract Command Class.



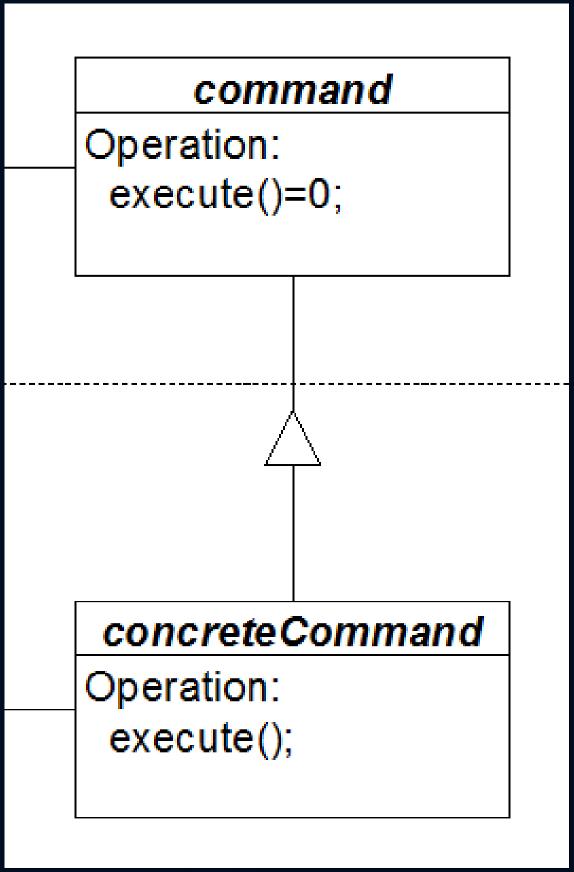

Command
Execute()
Document
Open()
Close()
Cut()
Copy()
Paste()
document
PasteCommand
Execute()
document->Paste()


Command
Execute()
Application
Add(Document)
application
OpenCommand
Execute()
AskUser()
name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()

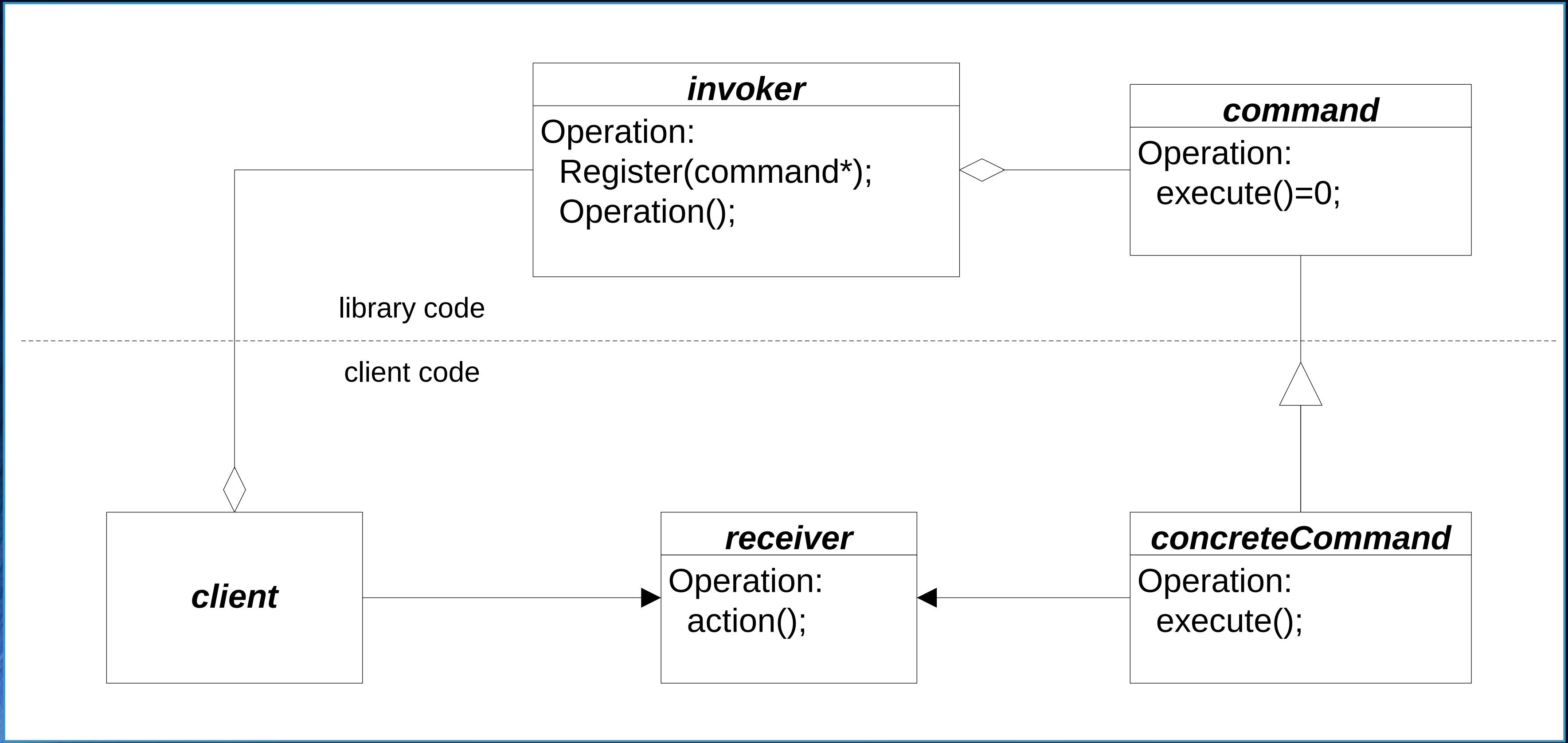

Command
Execute()
commands
MacroCommand
Execute()
for all c in commands
c->Execute()

Method

- A client encapsulates a request, along with needed state, by deriving a specialized command from the invoker's abstract base command class.
- Client associates command with a receiver (by the processing it encapsulates) and sends it to the invoker.
- The invoker simply uses the command interface to cause execution in the receiver



Command Structure



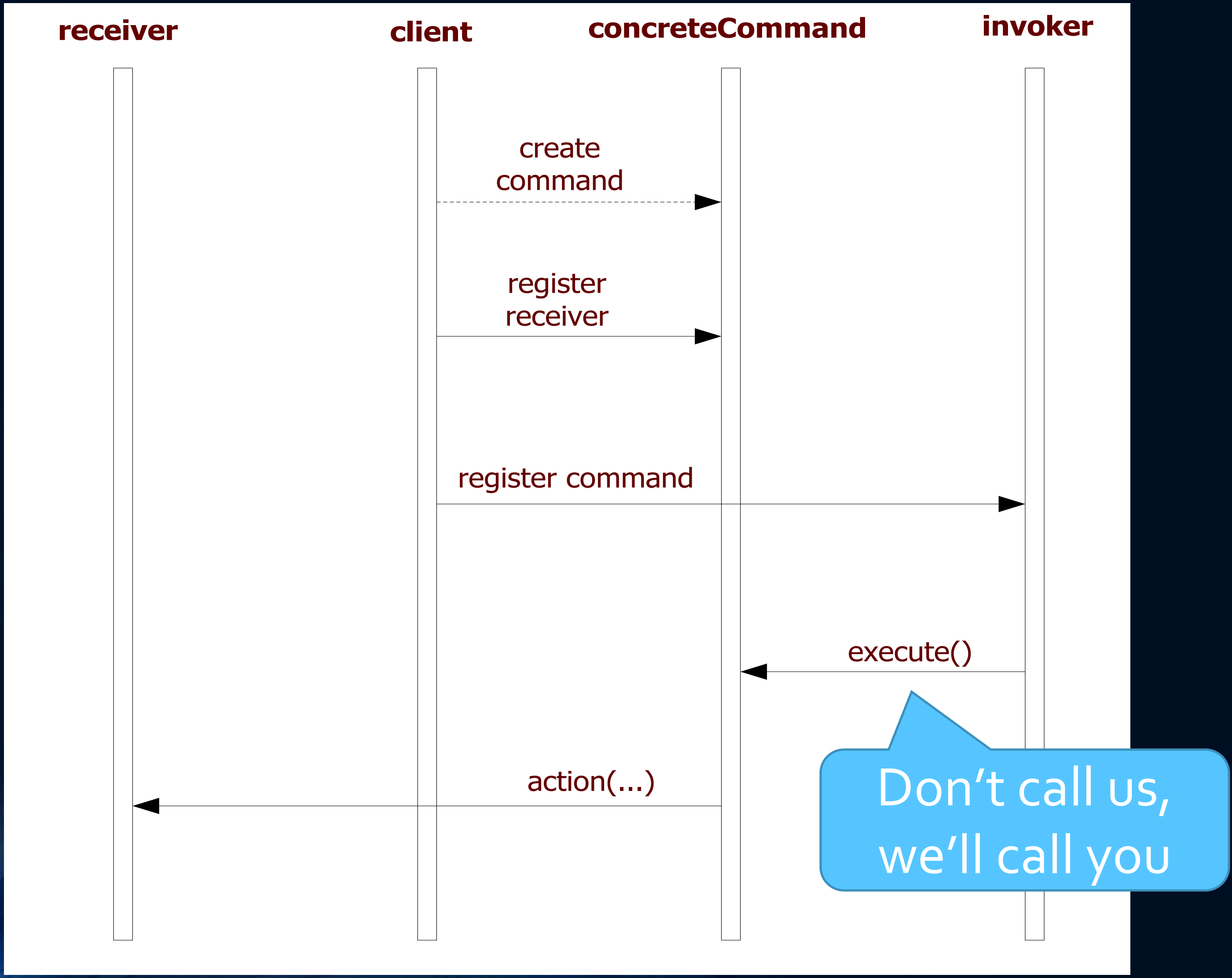
Command Participants

- Command provides an interface for executing commands
- ConcreteCommand provides binding between a receiver and command. It implements execution by calling receiver methods.
- Client creates a concreteCommand object, sets its receiver, and registers command with invoker
- Invoker issues command when an invoker event occurs
- Receiver actually performs the command processing

Collaborators

- Client creates concreteCommand and specifies receiver
- Invoker stores concreteCommand for later use
- Invoker issues request by calling execute() on command
- ConcreteCommand object invokes operations on its receiver to carry out the request

Command Event Trace



Applicability

- Use the command when you want to:
  - parameterize objects by an action to perform (menu items)
  - specify, queue, and execute requests at different times
    (a command object can have lifetime independent of the original request)
  - respond to library events in client code
    (library calls client functions even though the library knows nothing of client code)

Applicability

- As an object-oriented replacement for callback functions. Such functions are typically useful when designing menus and other user interfaces.
- Specify, queue, and then execute requests at different times. Command objects have lifetimes independent of their original request.
- Respond to library events in client code (library calls client functions even though the library knows nothing of client code).
- Supporting undo-able transactions. If the Command stores the relevant state of the receiver, it can reverse its own effects upon the receiver.
- Defining the structure of a system such that a broad class of high-level operations are built out of primitives. The Command allows various types of ***transactions*** to be invoked in the same way.

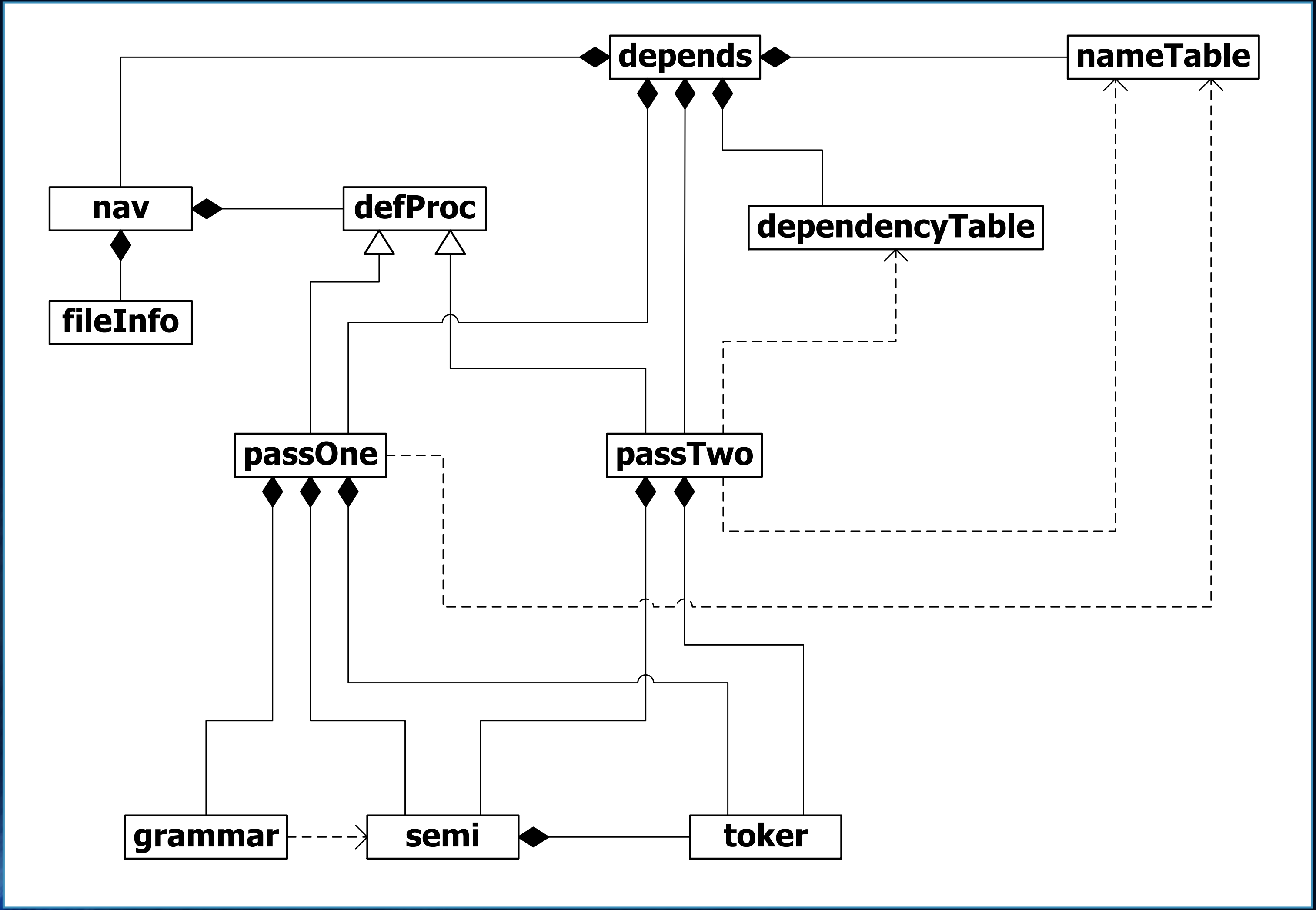
Consequences

- Command decouples the object that invokes an 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.
- It’s easy to add new commands, because you don’t have to change existing classes.

Known Uses

- Office – Word, Excel, …
- Virtually every graphical user interface known to mankind uses either callbacks, delegates, or commands. The object oriented ones use commands. MFC uses callbacks. .Net uses delegates.
- .Net delegates are a limited form of command

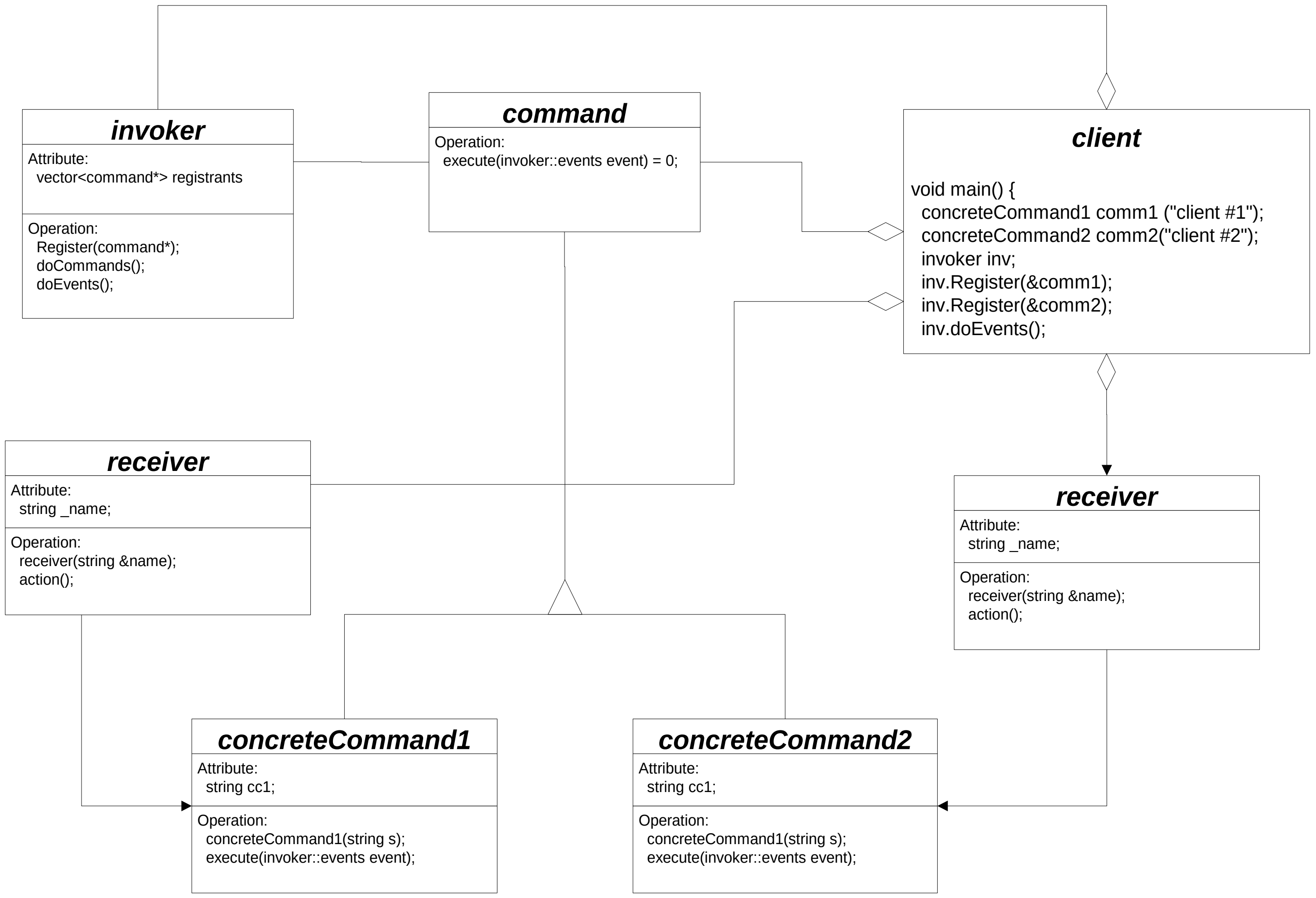
Navigator - Part of Code Analyzer



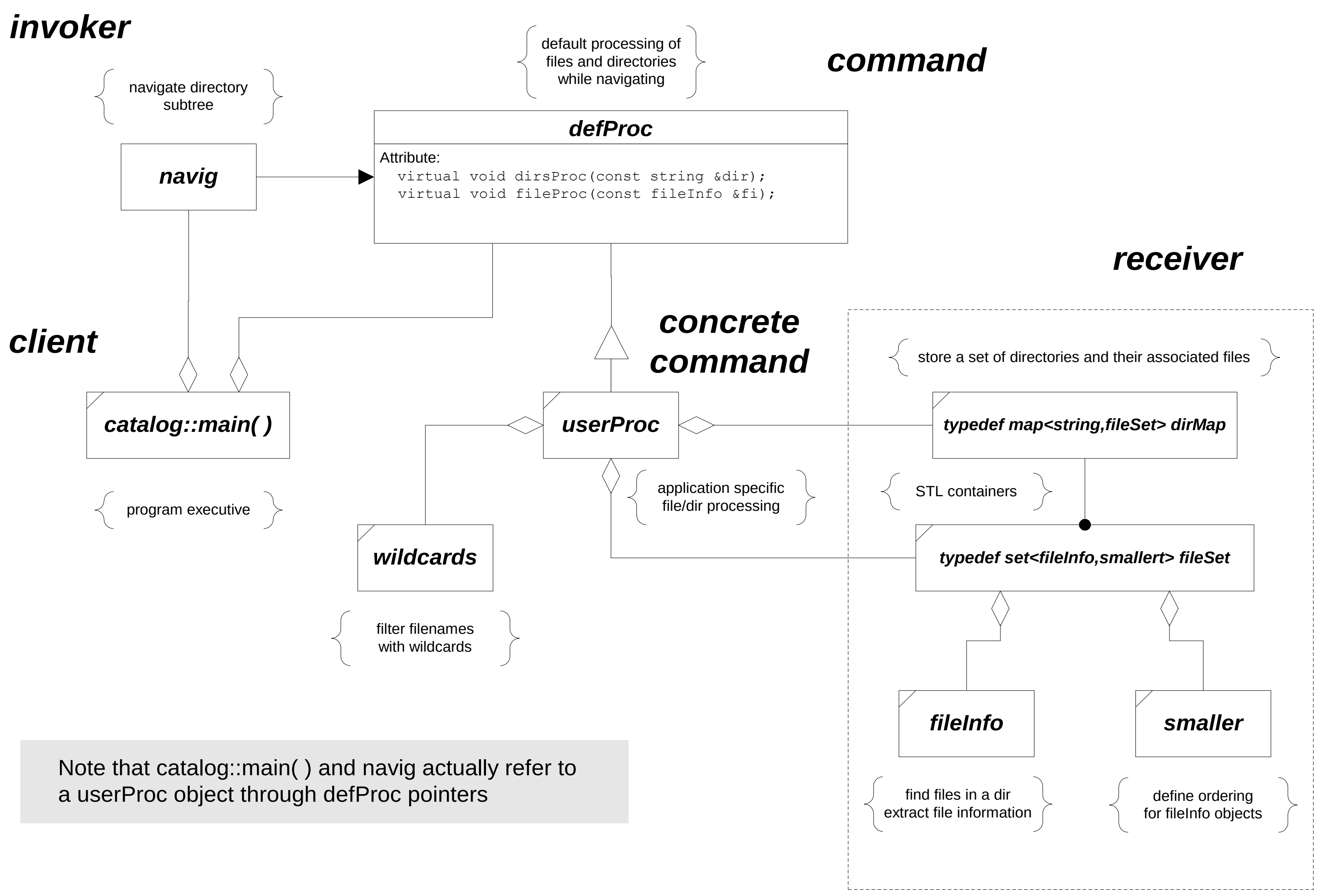
Related Patterns

- The relationships to other patterns as mentioned in the class text are rather tenuous.
- The command pattern is similar to the observer pattern. In both patterns an interested party can register to be notified of an event.

Command Pattern Skeleton Code



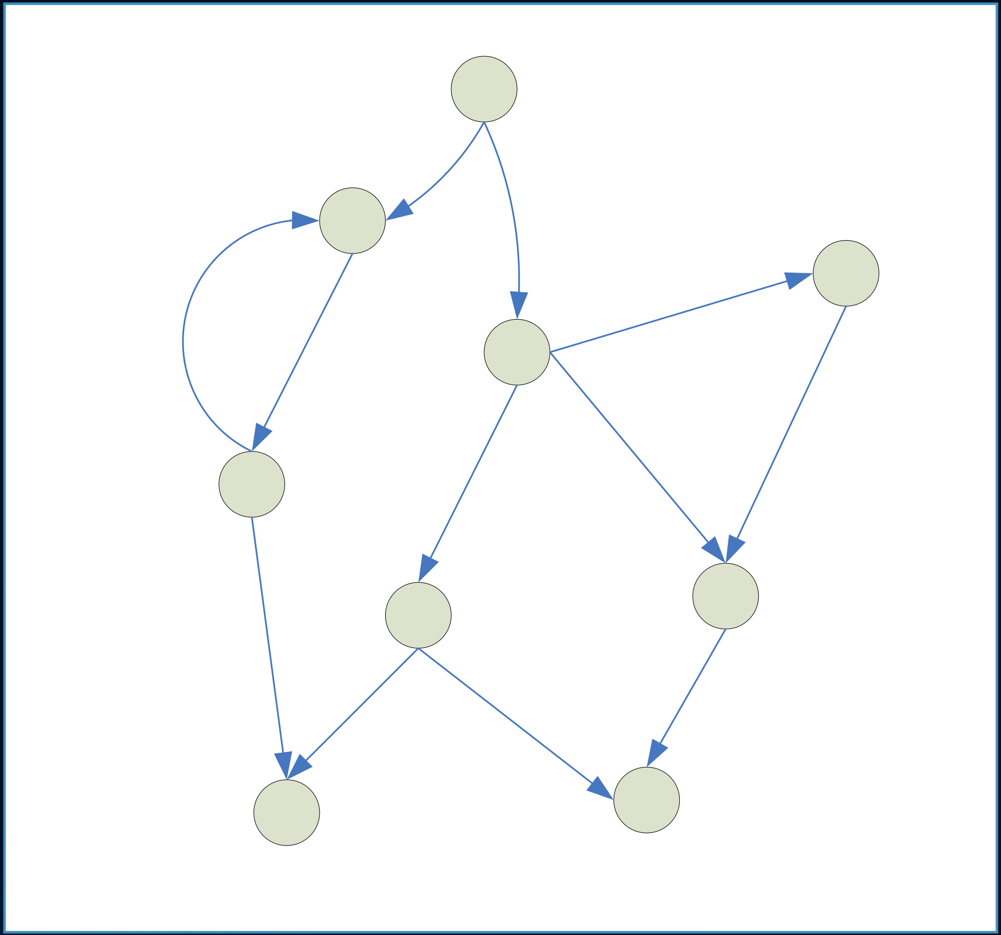
Class Diagram - Catalog Program


invoker
navigate directory subtree
navig
default processing of files and directories while navigating
command
defProc
Attribute:
virtual void dirsProc(const string &dir);
virtual void fileProc(const fileInfo &fi);
receiver
client
concrete command
catalog::main( )
program executive
userProc
application specific file/dir processing
wildcards
filter filenames with wildcards
store a set of directories and their associated files
typedef map<string,fileSet> dirMap
STL containers
typedef set<fileInfo,smallert> fileSet
fileInfo
find files in a dir extract file information
smaller
define ordering for fileInfo objects

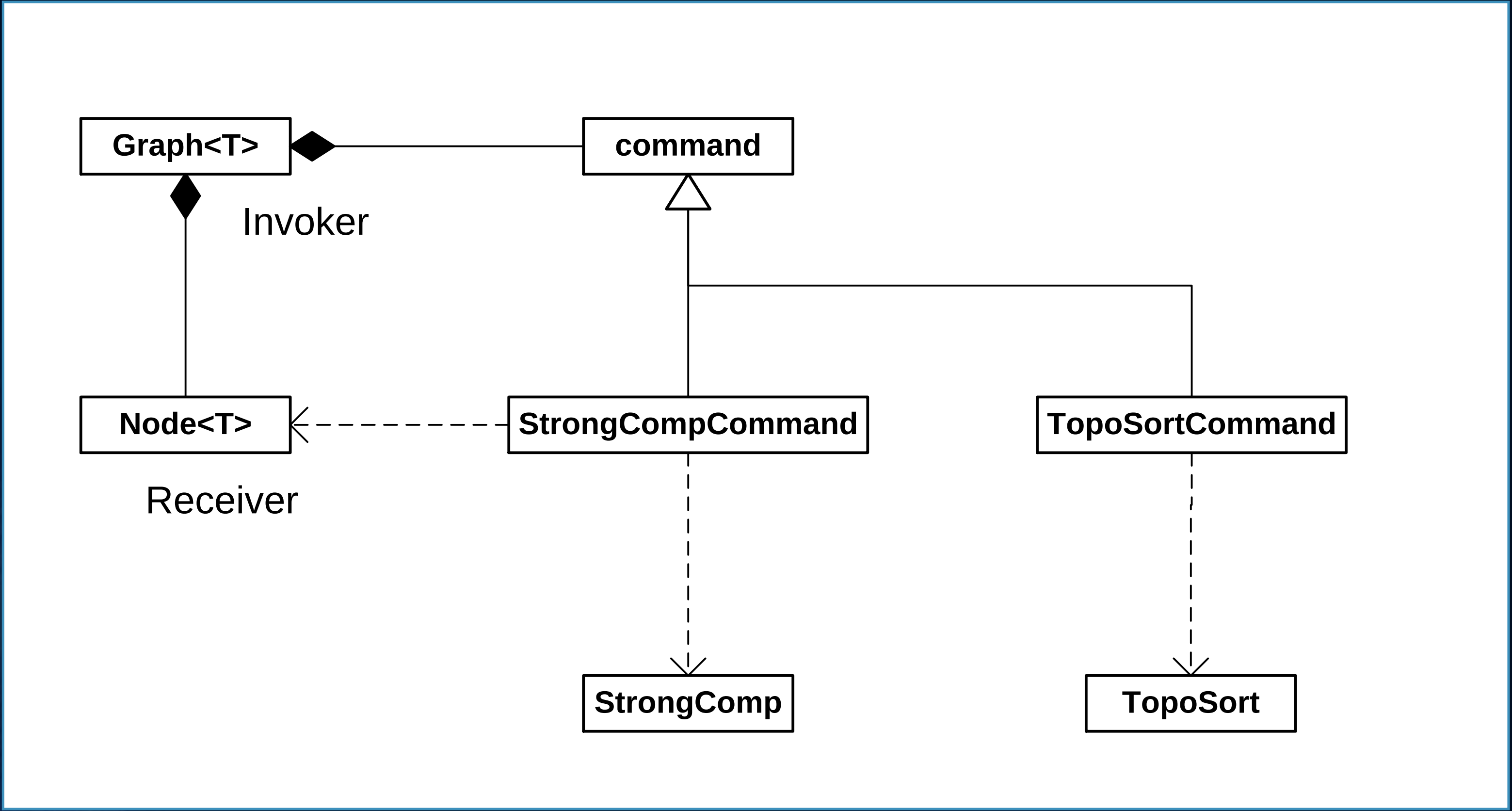
Note that catalog::main( ) and navig actually refer to a userProc object through defProc pointers

Application to Graph Algorithms

- Many graph algorithms are based on a traversal process
  - Breadth First Search
    - Shortest paths
    - Diameter
  - Depth First Search
    - Strong components
    - Topological sorting
- All of the above may be evaluated by executing functions on the graph nodes during search.



Command Pattern Applied to Graphs



End of Presentation