



Abram Hindle

Department of Computing Science
University of Alberta

 **Design Patterns**



Originally by Ken Wong

Images reproduced in these slides have been included under section 29 of the Copyright Act, as fair dealing for research, private study, criticism, or review. Further distribution or uses may infringe copyright.

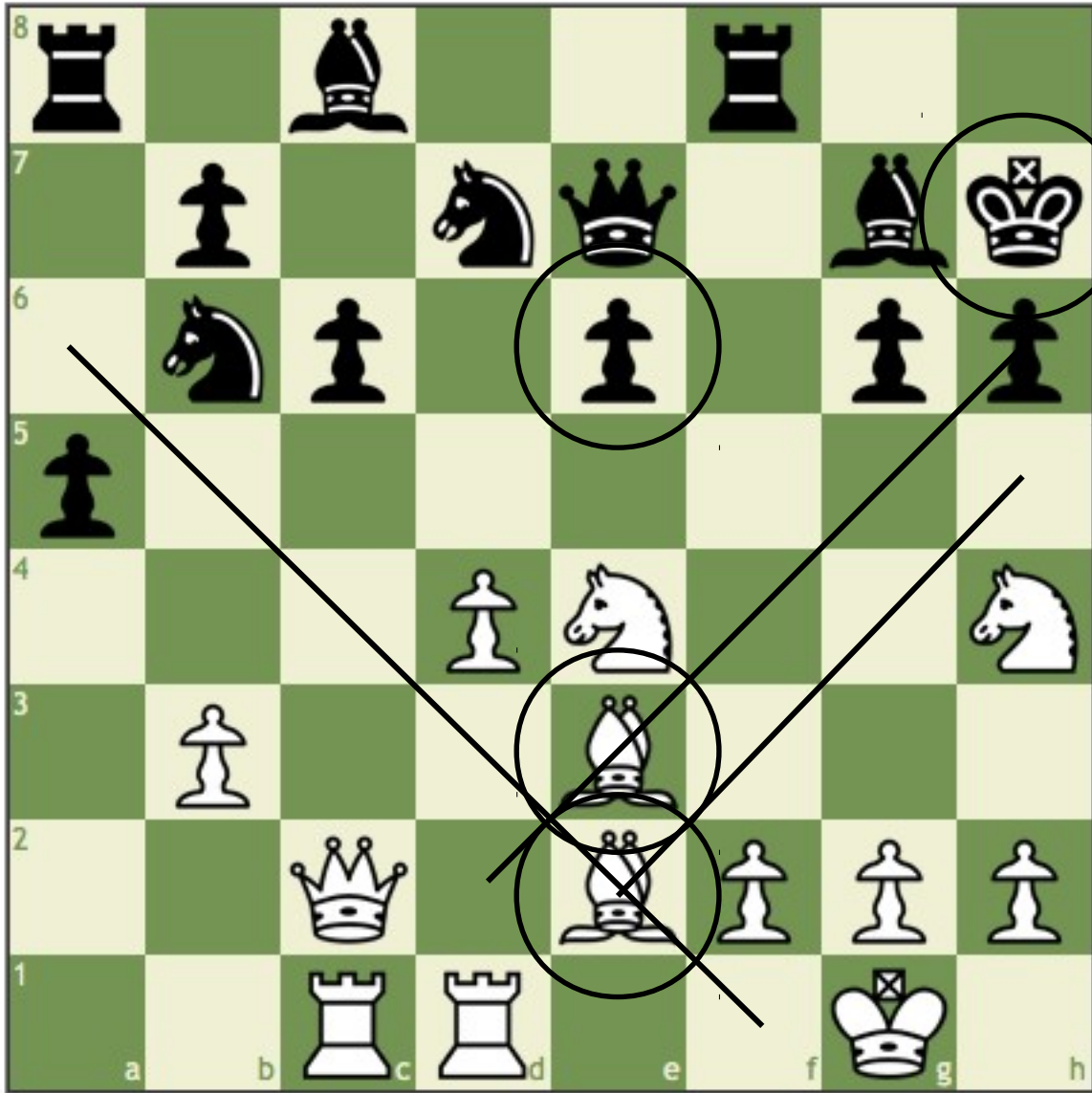


Patterns

Idea:

a pattern is a solution to a problem in some context

experts work with patterns where appropriate, rather than deriving everything from first principles



white to move

what pattern applies to win the game?

open diagonals

isolated pawn

bishop pair

exposed king



Code Patterns

```
// not idiomatic
```

```
int x = -1;

while (9 > x) {
    ++x;
    table[x] = x;
}
```

```
// idiomatic
```

```
for (int i = 0; i < 10; i++) {
    table[i] = i;
}
```



Design Patterns

Idea:

a design pattern is a practical, proven solution to a recurring design problem

not as well-defined as an algorithm or code, but consists of a coherent set of abstractions

e.g., model-view-controller



Design Patterns

Builds a design vocabulary:

“So I have this data object that notifies all view objects depending on it whenever the data changes. The nice thing is that views can be added or removed dynamically, and the data object doesn’t need to know the details of each type of view ...”

“Observer ...”



GoF Pattern Catalog

Creational patterns (creating objects):

abstract factory, builder, factory method, prototype, singleton

Structural patterns (connecting objects):

adapter, bridge, composite, decorator, façade, flyweight, proxy

Behavioral patterns (distributing duties):

chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor



Singleton Pattern

Design intent:

“ensure a class only has *one* instance, and provide a global point of access to it”

e.g., just one preferences object for application-wide settings

how?

Singleton Example Code 1

```
public class ExampleSingleton { // traditional way
    ...
    private static final ExampleSingleton instance =
        new ExampleSingleton();

    // private constructor,
    // so only this class itself can call new,
    // and other classes cannot make another
    // instance
    private ExampleSingleton() {
        ...
    }

    // use ExampleSingleton.getInstance() to access
    public static ExampleSingleton getInstance() {
        return instance;
    }
    ...
}
```

Singleton Example Code 2

```
public class ExampleSingleton { // by Bill Pugh
    ...
    // nested class is loaded and singleton instance
    // created on first call to getInstance()
    private static class ExampleSingletonHolder {
        private static final
            ExampleSingleton instance =
                new ExampleSingleton();
    }

    private ExampleSingleton() {
        ...
    }

    public static ExampleSingleton getInstance() {
        return ExampleSingletonHolder.instance;
    }
    ...
}
```

Singleton Example Code 3

```
public class ExampleSingleton { // lazy construction
    ...
    private static ExampleSingleton instance = null;

    // protected constructor makes it possible to
    // create instances of subclasses
    protected ExampleSingleton() {
        ...
    }

    // lazy construction of the instance
    public static ExampleSingleton getInstance() {
        if (instance == null) {
            instance = new ExampleSingleton();
        }
        return instance;
    }
    ...
}
```



Composite Pattern

Design intent:

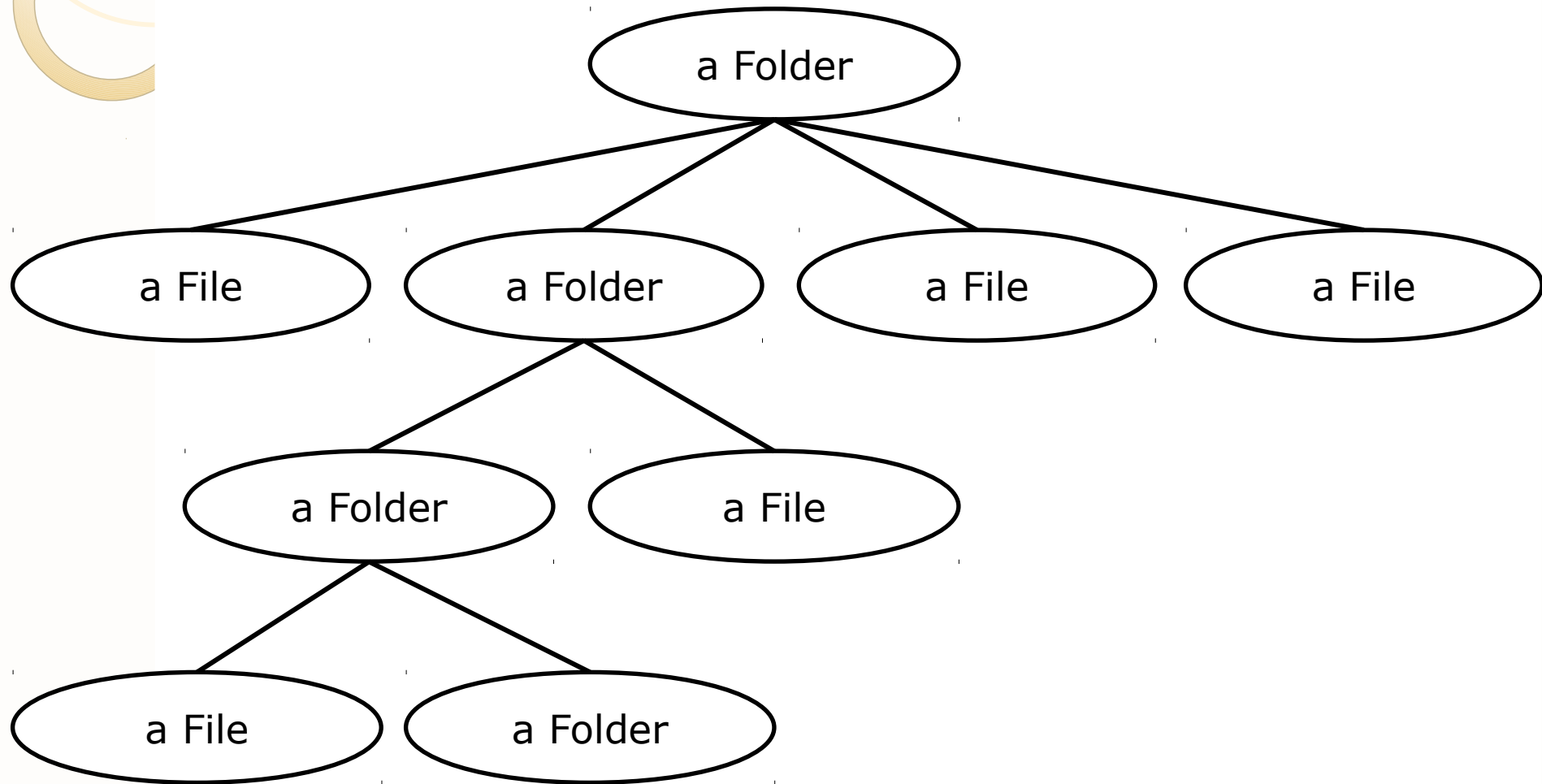
to compose individual objects to build up a tree structure

- e.g., a folder can contain files and other folders

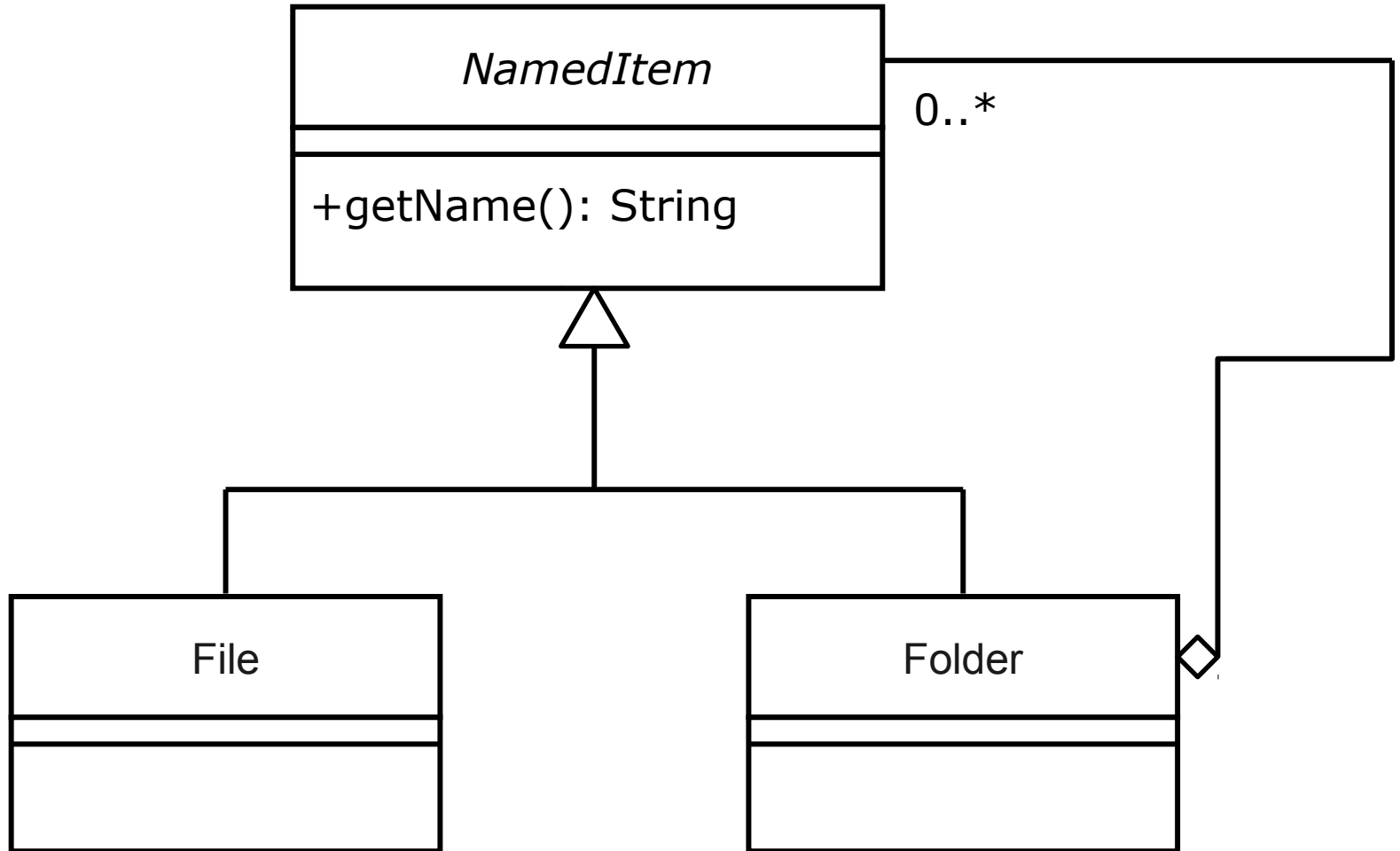
the individual objects and the composed objects are treated uniformly

- e.g., files and folders both have a name

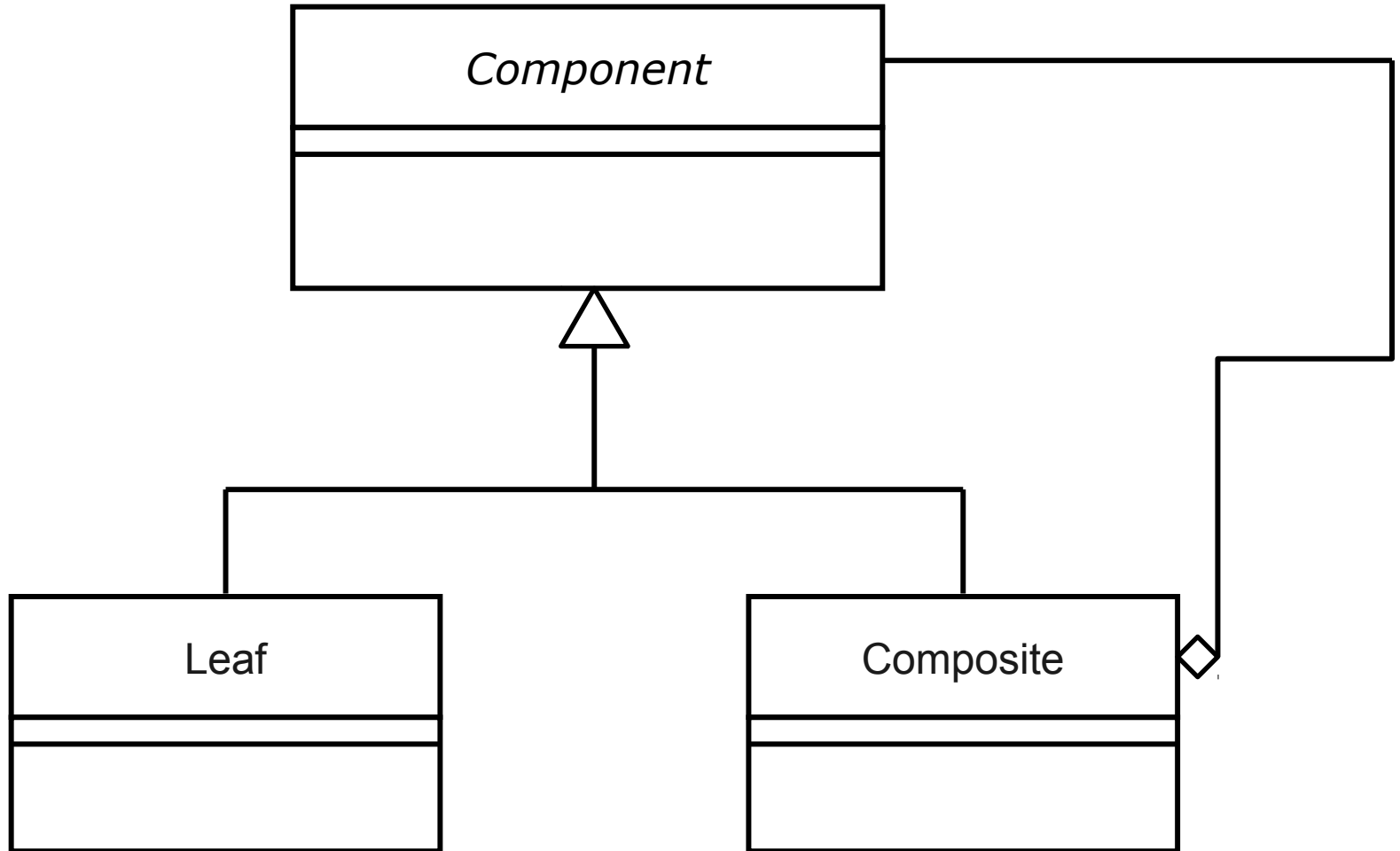
“Recursive (De)composition”



Composite Example Structure



Composite Pattern Structure



could have other leafs and composites



- **Command Pattern**



Command Pattern

Design intent:

“encapsulate a request as an object”, so you can run, queue, log, undo/redo these requests

also known as Action or Transaction



Motivation

Idea:

a class may want to issue a request without knowing anything about the operation being requested or the receiver object for the request

make request itself as a *command* object, so we can store it, run it, and pass it around



Motivation

Example uses:

pull logic out of the user interface classes into these command objects

- easier to change the user interface or to move to another user interface toolkit
- user interface classes call upon these command objects to initiate requests for services

devise a set of command primitives for your application back-end

- command subclasses encapsulate the right receivers for services



Applicability

Situations to use this pattern:

to specify, queue, and run commands

- these activities can happen at different times

to support undo

- a command object can store the state needed for reversing its effects
- executed commands can be stored in a history list

to implement a callback function

- object-oriented version of “function pointers”



Applicability

Situations to use this pattern:

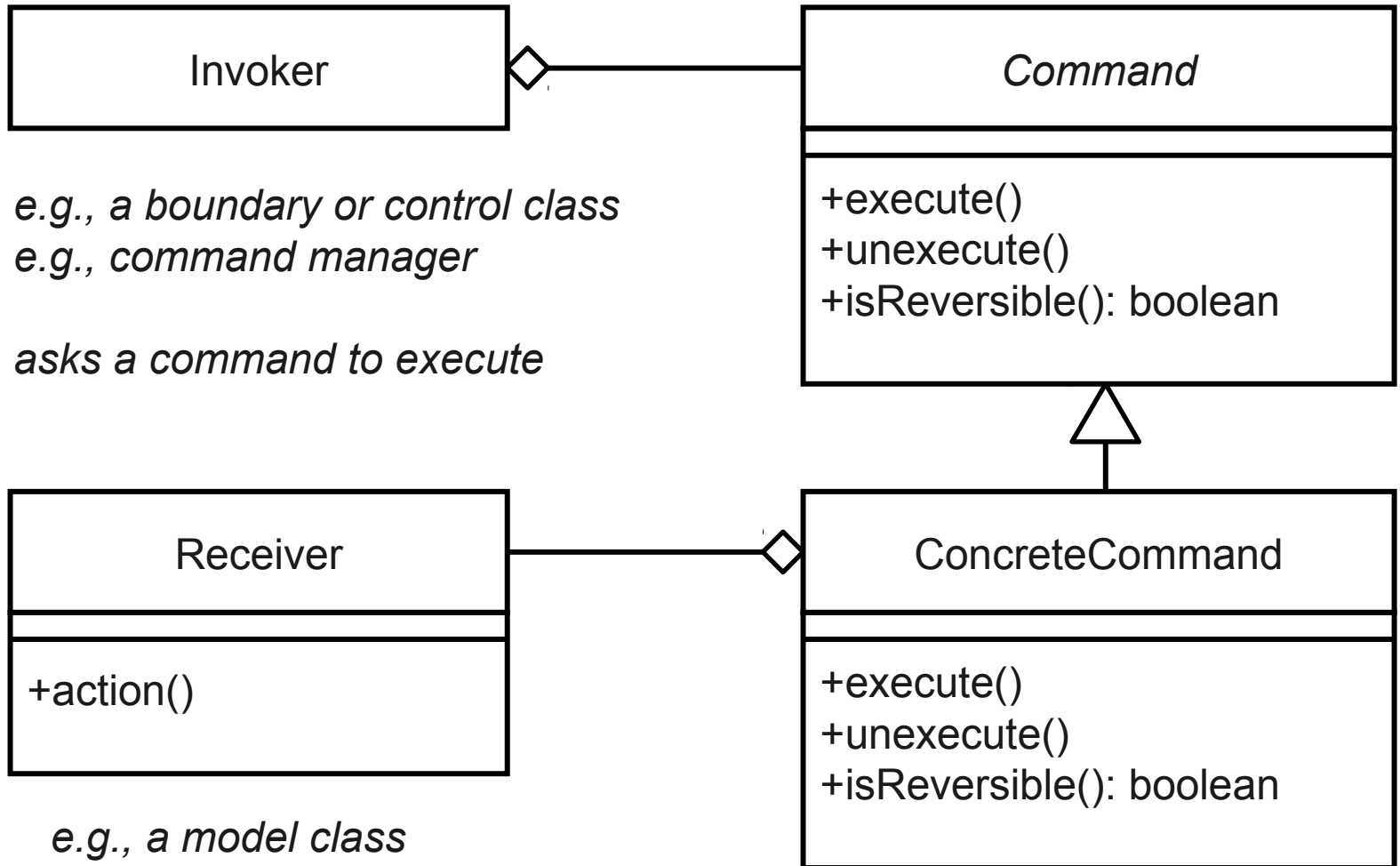
to store a log of commands executed

- can re-apply the commands if the system crashes

to structure a system around a set of primitive commands

- have “transactions”, each encapsulating a coherent set of changes to the model data

Command Pattern Structure



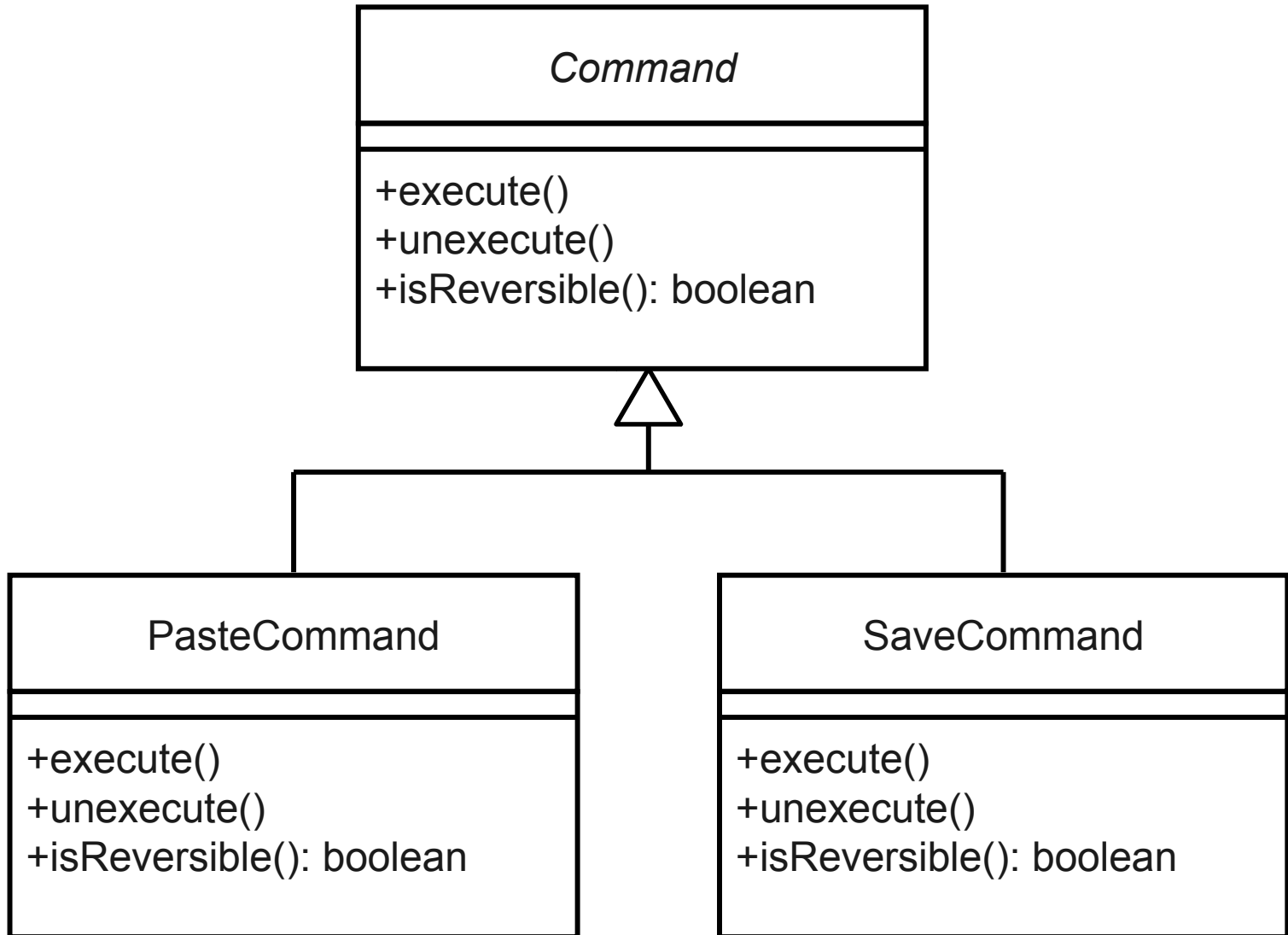
e.g., a boundary or control class
e.g., command manager

asks a command to execute

e.g., a model class

does the actual work

Command Examples





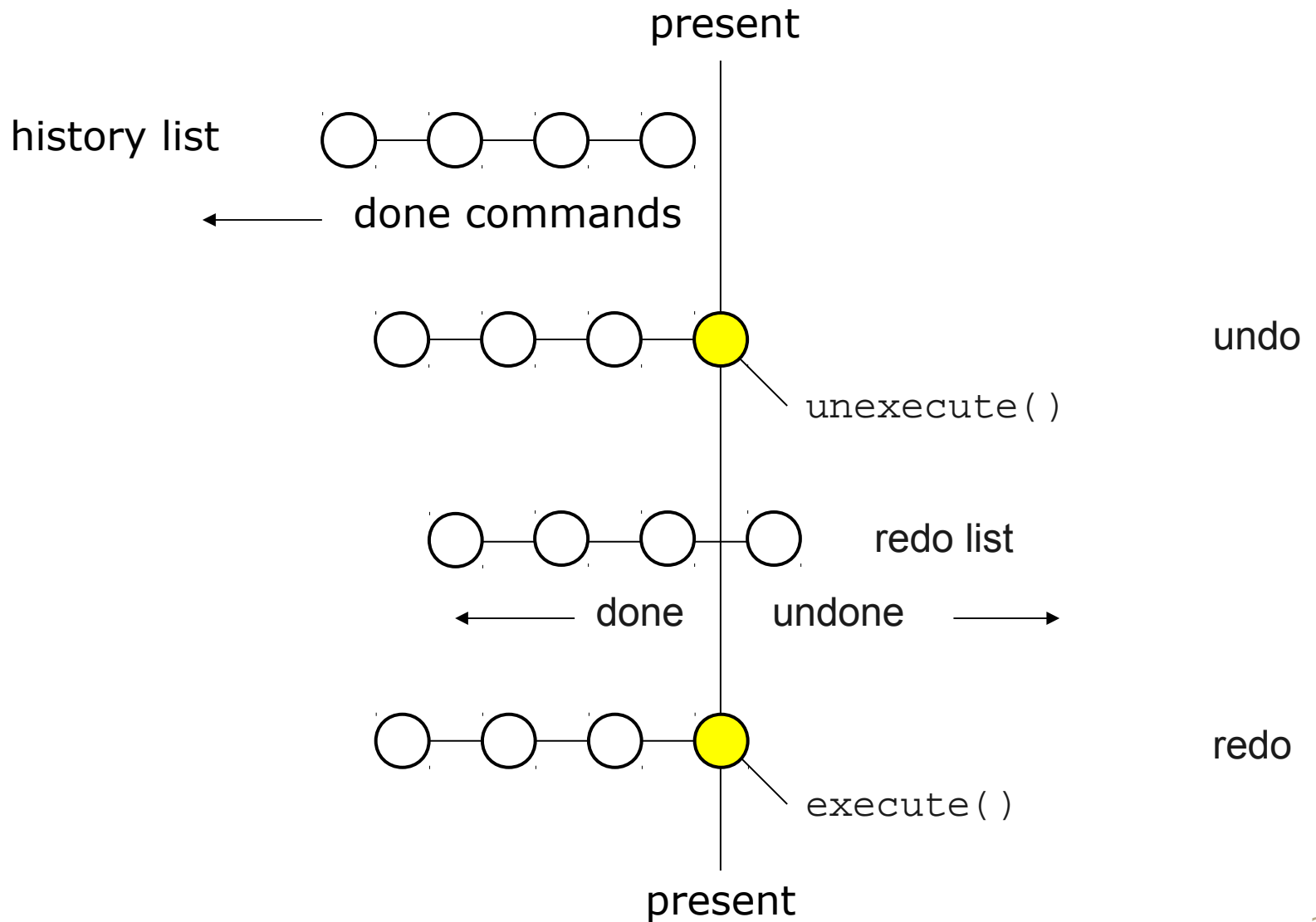
Consequences

Results:

decouples the object that invokes the operation from the one that knows how to perform it

easy to add new commands or manipulate them because they are first-class objects

Undo and Redo






Example Code


```
public abstract class Command {  
    public abstract void execute();  
    public abstract void unexecute();  
    public abstract boolean isReversible();  
}
```

```
// or use an interface
```



```
public class PasteCommand extends Command {
    private Document document; // a receiver
    private int position;
    private String text;
    ...
    public PasteCommand( Document document,
        int position, String text ) {

        this.document = document;
        this.position = position;
        this.text = text;
    }
    public void execute() {
        document.insertText( position, text );
    }
    public void unexecute() {
        document.deleteText( position,
            text.length() );
    }
    public boolean isReversible() {
        return true;
    }
}
```



```
public class CommandManager {
    private LinkedList<Command> historyList;
    private LinkedList<Command> redoList;

    private CommandManager() {
        historyList = new LinkedList<Command>();
        redoList = new LinkedList<Command>();
    }

    // invoke a command and add it to history list
    public void invokeCommand( Command command ) {

        command.execute();
```

```
        if (command.isReversible()) {
            historyList.addFirst( command );
        } else {
            historyList.clear();
        }

        if (redoList.size() > 0) {
            redoList.clear();
        }
    }
}
```

```
public void undo() {
    if (historyList.size() > 0) {
        Command command =
            historyList.removeFirst();
        command.unexecute();
        redoList.addFirst( command );
    }
}
public void redo() {
    if (redoList.size() > 0) {
        Command command =
            redoList.removeFirst();
        command.execute();
        historyList.addFirst( command );
    }
}
// CommandManager is a singleton
private static final CommandManager instance =
    new CommandManager();

public static CommandManager getInstance() {
    return instance;
}
}
```



```
// somewhere in an invoker
```

```
CommandManager commandManager =  
    CommandManager.getInstance();
```

```
Command command = new PasteCommand(  
    aDocument, aPosition, aText );
```

```
commandManager.invokeCommand( command );
```



Implementation Issues

Supporting multi-level undo and redo:

command state may include receiver(s), arguments used, and complete original values to be restored

- e.g., a delete command needs to remember what was deleted, so it can be undone

some requests cannot be undone since they may require too much state to restore

- e.g., saving the document, global search and replace



Implementation Issues

Macro commands:

can assemble commands into a command sequence to be run

how?

Macro Command

```
public class MacroCommand extends Command {
    ...
    private ArrayList<Command> commands;

    public MacroCommand() {
        commands = new ArrayList<Command>;
    }
    public addCommand( Command command ) {
        commands.add( command );
    }
    ...
    public void execute() {
        for (Command command : commands) {
            command.execute();
        }
    }
    ...
}
```

Use the Composite Pattern

