Chain of Responsibility

Comp-303 : Programming Techniques

Lecture 21

Alexandre Denault

Computer Science

McGill University

Winter 2004

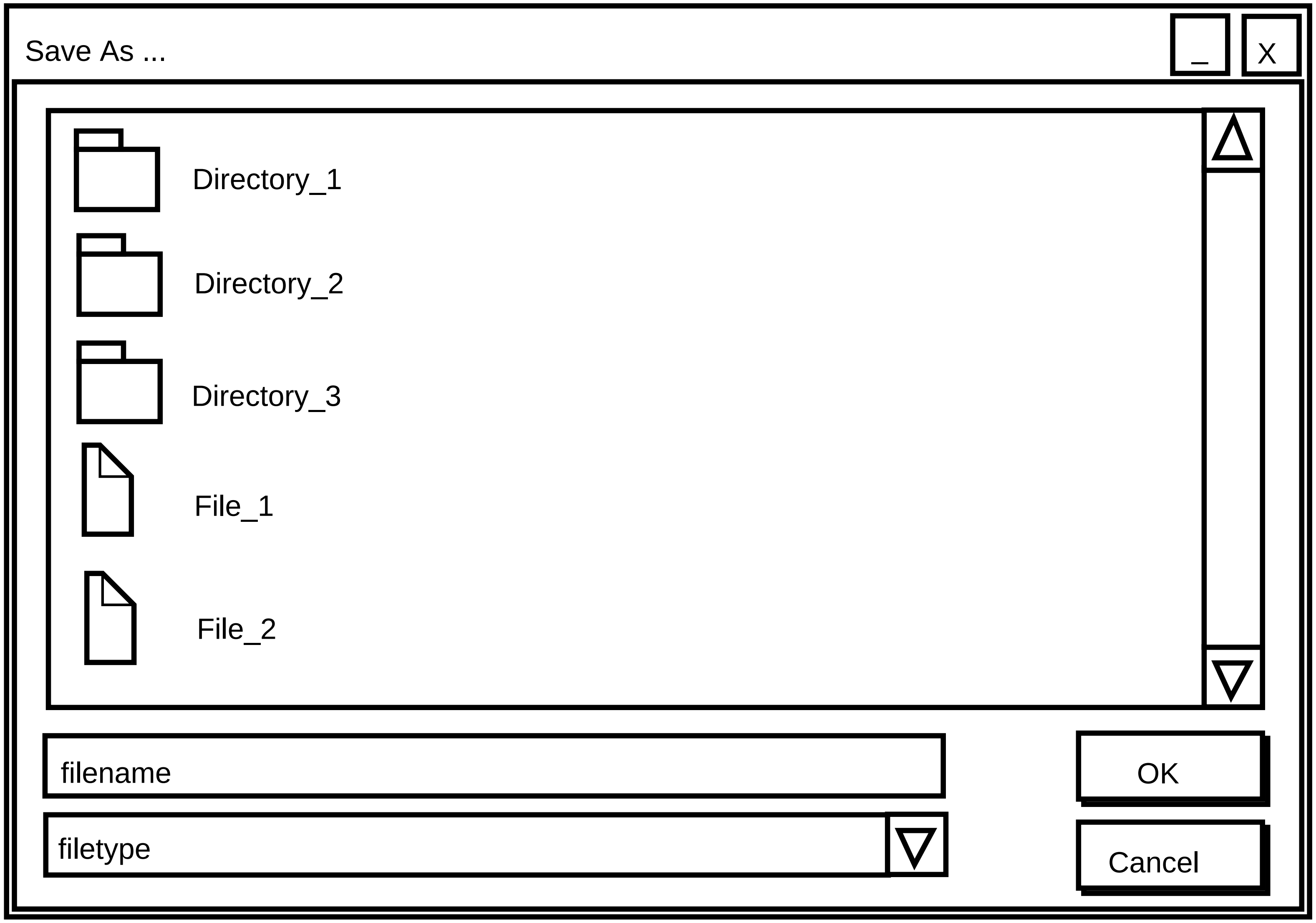
Last lecture . . .

- Flyweights allows us to deal with situation where a large number of objects could be required.
- There is still a lot of work to do on the project and not a lot of time to do it.

Typical Situation

- Today, our example starts with a help system.
- When a user presses F1 on his keyboard, he brings up a help screen for the object currently in focus.
- Consider the screen on the following screen on the next slides : *Given that not every object has a help screen, which help screen should be shown?*

Typical *Save As* Dialog Box



Flyweight

- *Pattern Name* : Chain of Responsibility
- *Classification* : Behavioral
- *Intent* : Avoid coupling the sender of a request to its receiver by giving more that one object the chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

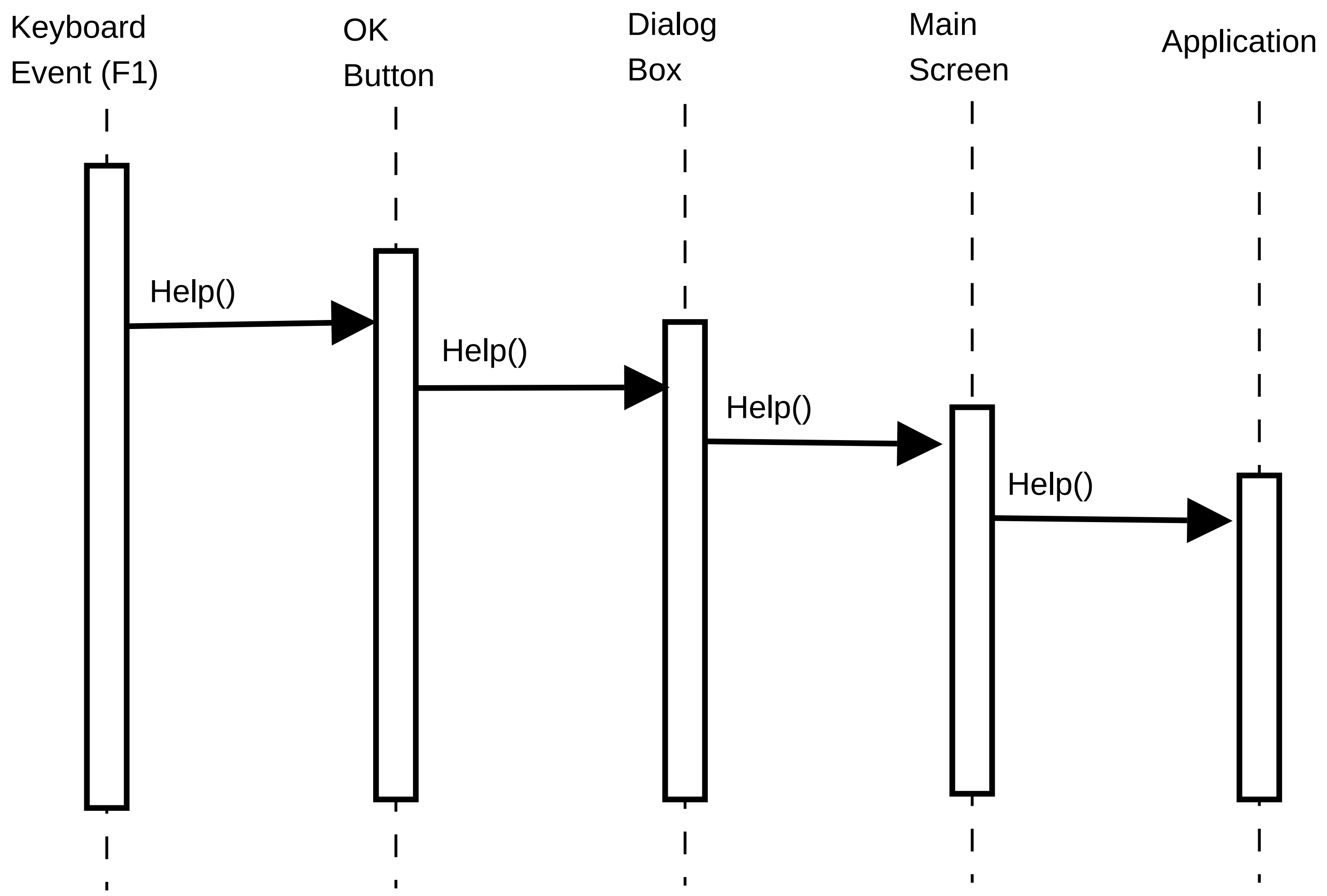
Motivation

- When an object doesn't have a help screen associated to it, we would like to see the help request forwarded to a lower level.
- For example, if the OK button doesn't have a help screen, then the request should be forwarded to the dialog box.
- If the dialog box doesn't have a help screen, then the request should be forwarded to the main screen.
- If the main screen doesn't have a help screen, then the request should be forwarded to the application.
- And so on ...

Motivation (cont.)

- The main idea is that when F1 is pressed, the application doesn't know what item will answer the request.
- An item in the application knows one of two things:
  - How it should answer a request for help. OR
  - Who should it pass along the request for help.
- The items don't need to know anything about requests it can't handle. It only needs to know how to pass them along.

Motivation (cont.)

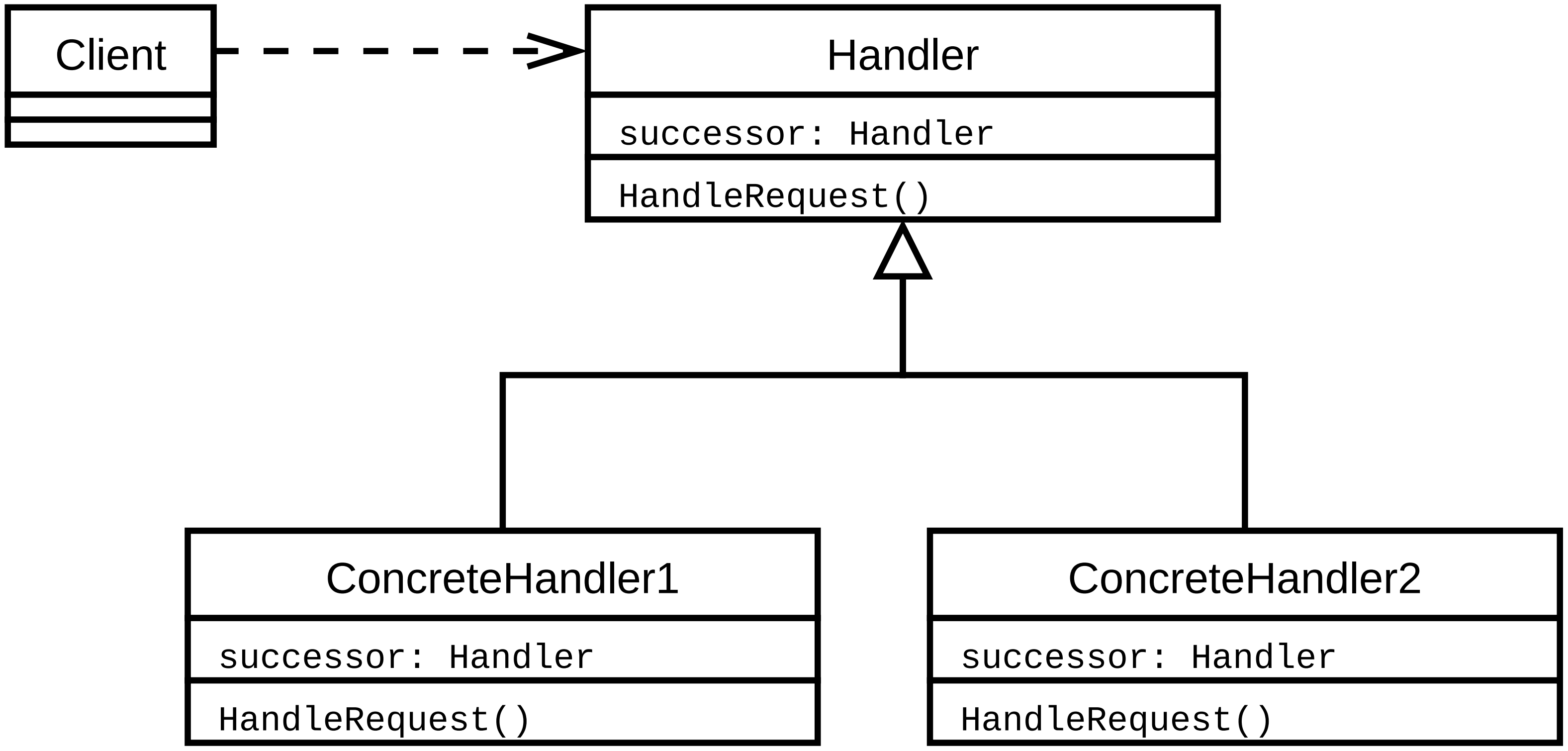


Applicability

Use Chain of Responsibility when ...

- ... more than one object may handle a request, and the handler isn't known ahead of time.
- ... you want to issue a request to one of several objects without specifying the receiver explicitly.
- ... the set of objects that can handle a request should be specified dynamically.

Structure



Client
Handler
successor: Handler
HandleRequest()
ConcreteHandler1
successor: Handler
HandleRequest()
ConcreteHandler2
successor: Handler
HandleRequest()

Participants

- Handler : defines an interface for handling requests and sometimes implements the successor link
- ConcreteHandler : handles the request or forwards it to the successor
- Client : initiates the request to one of the ConcreteHandler in the chain

Collaborations

- The client issues the request to the ConcreteHandler.
- The request is then propagated along the chain until a ConcreteHandler takes responsibility.

Consequences

- Reduce coupling : The pattern frees the receiving object of knowing who should handle the request. Objects in the chain don't need to know the structure of the chain.
- Added flexibility: You can add or change responsibilities for handling a request by changing the chain at run time.
- Reception is not guaranteed: Nothing in the pattern guarantees that the request will be handled. A request can fall of the chain if the chain is not properly configured.

Implementation

- The are two ways to implement the successor chain:
  - Use existing links : some structures (such has trees) already have the necessary links to implement this pattern.
  - Create new links : if the handlers have no links (or the existing ones are unsuitable), you will need to implement your own links.
- Which brings us to the next point: the chain doesn't have to be linear, it can have the shape of a tree.
- Finally, you can make the handler more flexible by representing the request as on object and passing it along as a parameter.

Does this look familiar?

```
public abstract class Server {

  public void receive(Message msg) {

   switch(msg->getType()) {
     case PlayerMove:
       //do something
     case GraphicUpdate:
       //do something
     case ChatMessage:
       //do something
     case GameMessage:
       //do something
     default:
       //We've got trouble
   }
  }
}
```

Sample Code

```
public abstract class Handler {

  private Handler successor = null;

  public abstract boolean receive(Message msg);
  // Returns true if message is handled.

  public void setSuccessor(Handler su) {
   this.successor = su;
  }

  public boolean nextSuccessor(Message msg){
   if (this.successor == null) return false;
   return this.successor.receive(msg);
  }
}
```

Building our handlers

```
public class GraphicHandler extends Handler {

  public boolean receive(Message msg) {

    if (msg is related to graphics) {
     // handle request
     return true;
    } else {
     return nextSuccessor(msg);
    }

  }
}
```

Using our handlers

```
public class Server {

  Handler graphicMsgHandler;
  Handler playerMsgHandler;
  Handler chatMsgHandler;
  Handler gameMsgHandler;

  public Server() {
   graphicMsgHandler = new GraphicHandler();
   playerMsgHandler = new PlayerHandler();
   chatMsgHandler = new ChatHandler();
   gameMsgHandler = new GameHandler();

   graphicMsgHandler.setSuccessor(playerMsgHandler);
   playerMsgHandler.setSuccessor(chatMsgHandler);
   chatMsgHandler.setSuccessor(gameMsgHandler);
  }
```

Using our handlers

```
  public void receive(Message msg) {

    boolean msgHandle = graphicMsgHandler(msg);

    if (msgHandle == false} {
      //We’ve got trouble
    }
  }
}
```

Notes

- This has a lot more overhead than using switch statements.
- However, this is dynamic and modular. The server doesn't need to know anything about message types.
- Message types that are received most often should be at the front of the chain.
- As mentioned previously, the chain can be dynamic.

Dynamic Chain

```
public disableChat() {
 playerMsgHandler.setSuccessor(gameMsgHandler);
}

public enableChat() {
 playerMsgHandler.setSuccessor(chatMsgHandler);
}
```

Known Uses

- Event Handlers in Guis

Related Patterns

- Composite

Summary

- Chain of Responsibility is a useful pattern when you want to decouple sender and receiver.
- It allows you to use objects to handle events dynamically (i.e. dynamic switch statements).

Tool of the day: JProfiler

- A profiler is a computer program that can track the performance of another program, thereby finding bottle necks.
- If your having performance problems with your project, a profiler can help you pinpoint the problem spots.
- JProfiler is an example of such a tool. It is very easy to use. Unfortunately it's commercial.
- There are many other Java profiler out there (JMP, Extensible Java Profiler (EJP), YourKit Java Profiler, etc).
- IBM's and Sun's JVM even have a built-in Java profiler (but it's kinda unreliable and hard to use).

```
java -Xrunhprof:cpu=times className
java -Xrunhprof:heap=sites className
```

References

- These slides are inspired (i.e. copied) from these three books.
  - Design Patterns, Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; Addison Wesley; 1995
  - Java Design Patterns, a Tutorial; James W. Cooper Addison Wesly; 2000
  - Design Patterns Explained, A new Perspective on Object Oriented Design; Alan Shalloway, James R. Trott; Addison Wesley; 2002