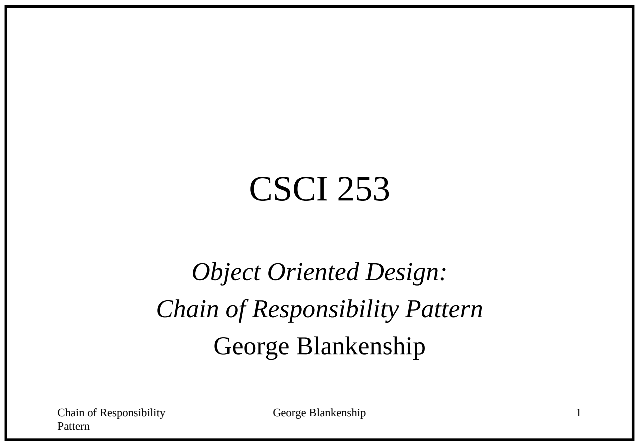
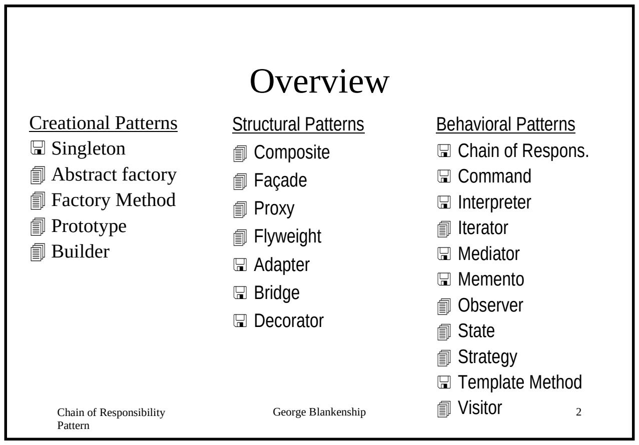


CSCI 253

Object Oriented Design:
Chain of Responsibility Pattern
George Blankenship

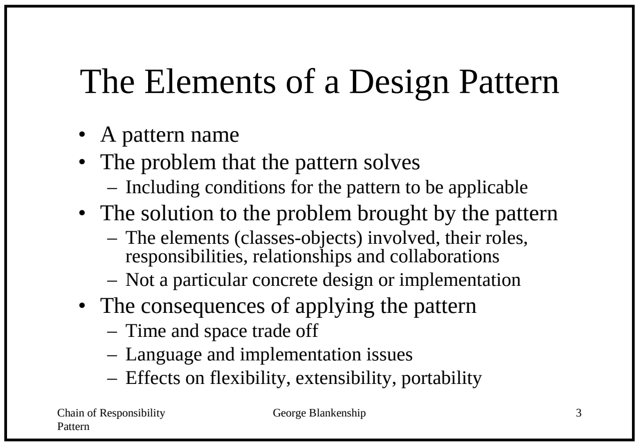


Overview

<u>Creational Patterns</u>
🖫 Singleton
🗐 Abstract factory
🗐 Factory Method
🗐 Prototype
🗐 Builder

<u>Structural Patterns</u>
🗐 Composite
🗐 Façade
🗐 Proxy
🗐 Flyweight
🖫 Adapter
🖫 Bridge
🖫 Decorator

<u>Behavioral Patterns</u>
🖫 Chain of Respons.
🖫 Command
🖫 Interpreter
🗐 Iterator
🖫 Mediator
🖫 Memento
🗐 Observer
🗐 State
🗐 Strategy
🖫 Template Method
🗐 Visitor



The Elements of a Design Pattern

- A pattern name
- The problem that the pattern solves
  - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
  - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
  - Not a particular concrete design or implementation
- The consequences of applying the pattern
  - Time and space trade off
  - Language and implementation issues
  - Effects on flexibility, extensibility, portability

Chain of Responsibility Pattern: The Problem

the **chain-of-responsibility pattern** is a design pattern consisting of a source of command objects and a series of **processing objects**. Each processing object contains a set of logic that describes the types of command objects that it can handle, and how to pass off those that it cannot to the next processing object in the chain.

Chain of Responsibility: Applicability

- More than one handler that can handle a request and there is no way to know which handler to use. The handler must be determined automatically by the chain.
- A request to one of several objects without specifying which one explicitly.
- Modify the set of objects dynamically that can handle requests.

Message Receipt

```
public void onMessage(int transportId, String message) {
        HL7Connection connection = HL7Connection.findConnectionByTransport(transportId);
        if(connection!=null) {
                traceMessage(message," client received from "+
                    connection.getAddressA()+" ("+
                    String.valueOf(port)+")"+" on "+portType);
                connection.resetMessageSpaceIn();
                connection.resetMessageSpaceError();
                connection.resetMessageSpaceOut();
                connection.updateDataOut();
                try {
                        messageEngine.parseMessage(message,null,connection);
                } catch (ChameleonException e) {
                        Trace.exception(CODE_FILE,e,"(onMessage) exception while parsing message");
                        messageEngine.messageError(207,"Message processing internal error");
                        engine.sendAcceptACK(connection);
                        return;
                }
                if(connection.hasActiveFSM()) {
                        connection.getFSM().execute(Constants.FSM_MESSAGE_RECEIVED);
                } else if(HomeTelehealth.isWaitingMessage()) {
                        connection.setFSM(HomeTelehealth.getWaitingMessageFSM());
                        connection.getFSM().execute(connection.getAddressA(),connection.getIdNumber()); // tell FSM that there is a new
connection
                        HomeTelehealth.clearWaitingConnectionFSM();            // no FSM waiting for a connection
                } else {
                        engine.sendApplicationACK(connection);
                }
        } else {
                traceMessage(message," client received (no connection) transport ("+String.valueOf(transportId)+")");
        }
}
```

Message Receipt (Chain)

```
public void onMessage(int transportId, String message) {
        if(connection!=null) {
                try {
                        messageEngine.parseMessage(message,null,connection);
                } catch (ChameleonException e) {
                        engine.sendAcceptACK(connection);
                        return;
                }
                if(connection.hasActiveFSM()) {

        connection.getFSM().execute(Constants.FSM_MESSAGE_RECEIVED);
                } else if(HomeTelehealth.isWaitingMessage()) {
                } else {
                        engine.sendApplicationACK(connection);
                }
        }
}
```

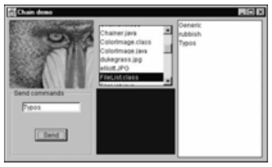
Chain of Responsibility Pattern   George Blankenship   7

Message Processing

```
public void onCensusMessage(CensusMessage abstractMessage, Transport transport) throws ChameleonException {
        PatientData patientDataIn = connection.getPatientDataIn();
        try {
                int field;
                trace.write("receive MDMT02 (Census) message");
                mainGUI.append("receive MDMT02 (Census) message");
                MSHTable mshTable = abstractMessage.MSH();
                if(mshTable.countOfRow()!=1) {
                        throw(new ChameleonException("incorrect count of MSH segments ("+mshTable.countOfRow()+")"));
                }
                engine.getMSHdata(mshTable,connection);
                field = connection.isMSHvalid();
                if(field!=0 && parameters.isVendor()==false) {
                        messageErrorWithException("MSH",field,0,400,"invalid field");
                }
        } catch (ChameleonException e) {        // this will generate an accept reject message
                trace.exception(e," error in MDMT02 (Census) message");
                messageError(201,e.toString());
                throw e;
        }
        try {
                engine.sendAcceptACK(connection);
                EventTable evnTable = abstractMessage.Event();
                if(evnTable.countOfRow()!=1) {
                        throw(new ChameleonException("incorrect count of EVN segments ("+evnTable.countOfRow()+")"));
                }
                engine.getEVNdata(evnTable,connection);
                DocumentLineTable documentLineTable = abstractMessage.DocumentLine();
                if(documentLineTable.countOfRow()!=1) {
                        throw(new ChameleonException("incorrect count of OBX segments ("+documentLineTable.countOfRow()+")"));
                }
                engine.getCensusReportData(documentLineTable,connection);
                connection.setMessageType(Constants.FSM_CENSUS_MESSAGE);
        } catch (ChameleonException e) {        // this will generate an application reject message
                trace.exception(e," error in MDMT02 (Census) message");
                applicationError(201,e.toString());
        }
}
```

Chain of Responsibility Pattern   George Blankenship   8

Dynamic Chain



- A simple system for display the results of typed in requests. These requests can be
  - Image filenames
  - General filenames
  - Colors
  - Other commands
- In three cases, we can display a concrete result of the request, and in the last case, we can only display the request text itself. In the above example system, we type in "Mandrill" and see a display of the image Mandrill.jpg. Then, we type in "FileList" and that filename is highlighted in the center list box. Next, we type in "blue" and that color is displayed in the lower center panel. Finally, if we type in anything that is neither a filename nor a color, that text is displayed in the final, right-hand list box.

Chain of Responsibility Pattern   George Blankenship   9

Chain Creation

```
public interface Chain {
   public abstract void addChain(Chain c);
   public abstract void sendToChain(String mesg);
   public Chain getChain();
}
```

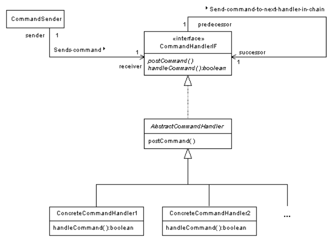
Message Processing (Chain)

```
public void onCensusMessage(CensusMessage abstractMessage, Transport transport) throws
ChameleonException {
        try {
                engine.getMSHdata(mshTable,connection);
                }
        } catch (ChameleonException e) {        // this will generate an accept reject message
        }
        try {
                engine.sendAcceptACK(connection);
                }
                engine.getEVNdata(evnTable,connection);
                }
                engine.getCensusReportData(documentLineTable,connection);
        } catch (ChameleonException e) {        // this will generate an application reject message
        }
}
```

Message Generation

```
public String generateMessage(Engine AEngine) throws ChameleonException {
        MSHsegment mshDataOut = connection.getMSHsegmentOut();
        CensusMessage abstractMessage = new CensusMessage(AEngine);
        abstractMessage.MSH().addRow();
        MSHTable mshTable = abstractMessage.MSH();
        mshDataOut.setReceiveApplication(parameters.getCensusApplication());
        mshDataOut.setReceiveFacility(parameters.getVistAstation(),
                        parameters.getVistAdomainName());
        mshDataOut.setSendApplication(connection.getLocalApplication());
        mshDataOut.setSendFacility(connection.getLocalStation(),
                        connection.getLocalDomainName());
        mshDataOut.setType("MDM");
        mshDataOut.setAcceptACK("AL");      // want an ACK for this message
        mshDataOut.setApplicationACK("AL");
        engine.buildMSH(mshTable,connection);
        EVNsegment evnDataOut = connection.getEVNsegmentOut();
        evnDataOut.setType("T02");
        abstractMessage.Event().addRow();
        EventTable eventTable = abstractMessage.Event();
        engine.buildEVN(eventTable,connection);
        abstractMessage.DocumentLine().addRow();
        DocumentLineTable documentLineTable = abstractMessage.DocumentLine();
        engine.buildCensusReport(documentLineTable,connection);
        return abstractMessage.generateMessage();
}
```

The Chain of Responsibility Pattern: Structure

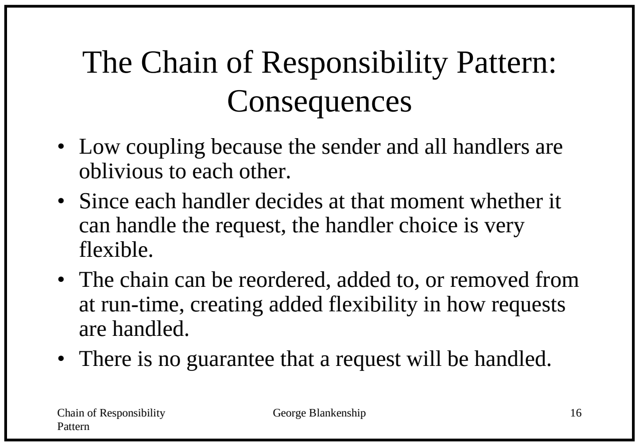


The Chain of Responsibility Pattern: Participants

- **CommandSender -** Instances of a class send commands to the first object in the chain. It sends a command by calling the **CommandHandlerIF** object's **postCommand** method.
- **CommandHandlerIF -** All objects in the chain of objects implement the interface.
  - **handleCommand** method to consume the commands. The **handleCommand** method returns true if it consumed the command or false if it did not.
  - **postCommand** method the **handleCommand** method. If the **handleCommand** method returns false and invokes the **postCommand** method fo next object in the chain. If the **handleCommand** method returns true, the processing is complete.
- **AbstractCommandHandler -** Classes in this role are abstract classes that implement the **postCommand** method to provide a common implementation of **postCommand**.
- **ConcreteCommandHandler1**, **ConcreteCommandHandler2 -** Instances of classes are objects in a chain of objects that can handle commands.

The Chain of Responsibility Pattern: Collaboration

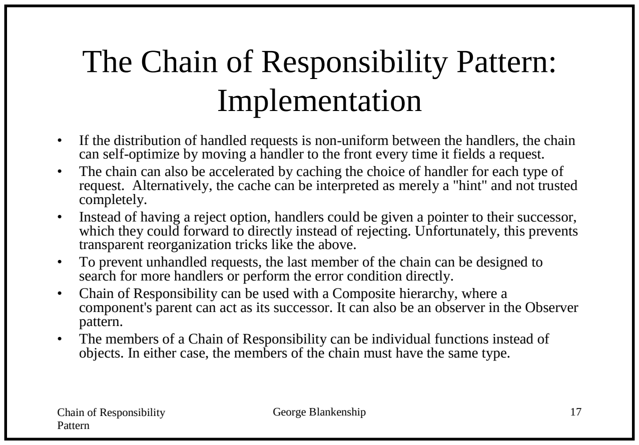
- The processing moves down the chain, item by item
- A chain using a linked list can perform this function automatically
- A logically equivalent method moves through the chain by directly invoking the objects



The Chain of Responsibility Pattern: Consequences

- Low coupling because the sender and all handlers are oblivious to each other.
- Since each handler decides at that moment whether it can handle the request, the handler choice is very flexible.
- The chain can be reordered, added to, or removed from at run-time, creating added flexibility in how requests are handled.
- There is no guarantee that a request will be handled.

Chain of Responsibility Pattern George Blankenship 16



The Chain of Responsibility Pattern: Implementation

- If the distribution of handled requests is non-uniform between the handlers, the chain can self-optimize by moving a handler to the front every time it fields a request.
- The chain can also be accelerated by caching the choice of handler for each type of request. Alternatively, the cache can be interpreted as merely a "hint" and not trusted completely.
- Instead of having a reject option, handlers could be given a pointer to their successor, which they could forward to directly instead of rejecting. Unfortunately, this prevents transparent reorganization tricks like the above.
- To prevent unhandled requests, the last member of the chain can be designed to search for more handlers or perform the error condition directly.
- Chain of Responsibility can be used with a Composite hierarchy, where a component's parent can act as its successor. It can also be an observer in the Observer pattern.
- The members of a Chain of Responsibility can be individual functions instead of objects. In either case, the members of the chain must have the same type.

Chain of Responsibility Pattern George Blankenship 17