Object-Oriented Software Engineering
Practical Software Development using UML and Java

Chapter 9:
Architecting and Designing Software

9.1 The Process of Design

Definition:

- *Design* is a problem-solving process whose objective is to find and describe a way:
  - —To implement the system's *functional requirements*...
  - —While respecting the constraints imposed by the *non-functional requirements*...
    - including the budget
  - —And while adhering to general principles of *good quality*

Design as a Series of Decisions

A designer is faced with a series of *design issues*

- These are sub-problems of the overall design problem.
- Each issue normally has several alternative solutions:
  - design *options*.
- The designer makes a *design decision* to resolve each issue.
  - This process involves choosing the best option from among the alternatives.

Making Decisions

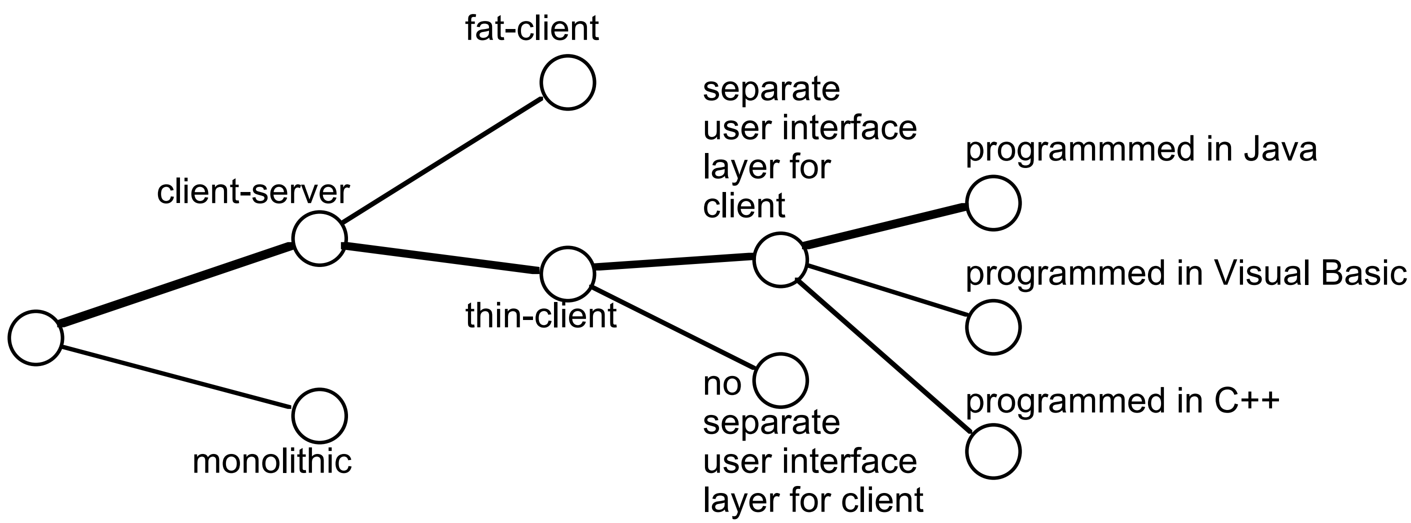
To make each design decision, the software engineer uses:

- Knowledge of
  - the requirements
  - the design as created so far
  - the technology available
  - software design principles and 'best practices'
  - what has worked well in the past

Design space

The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*

- For example:



Component

Any piece of software or hardware that has a clear role.

- A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
- Many components are designed to be reusable.
- Conversely, others perform special-purpose functions.

Module

A component that is defined at the programming language level

- For example, methods, classes and packages are modules in Java.

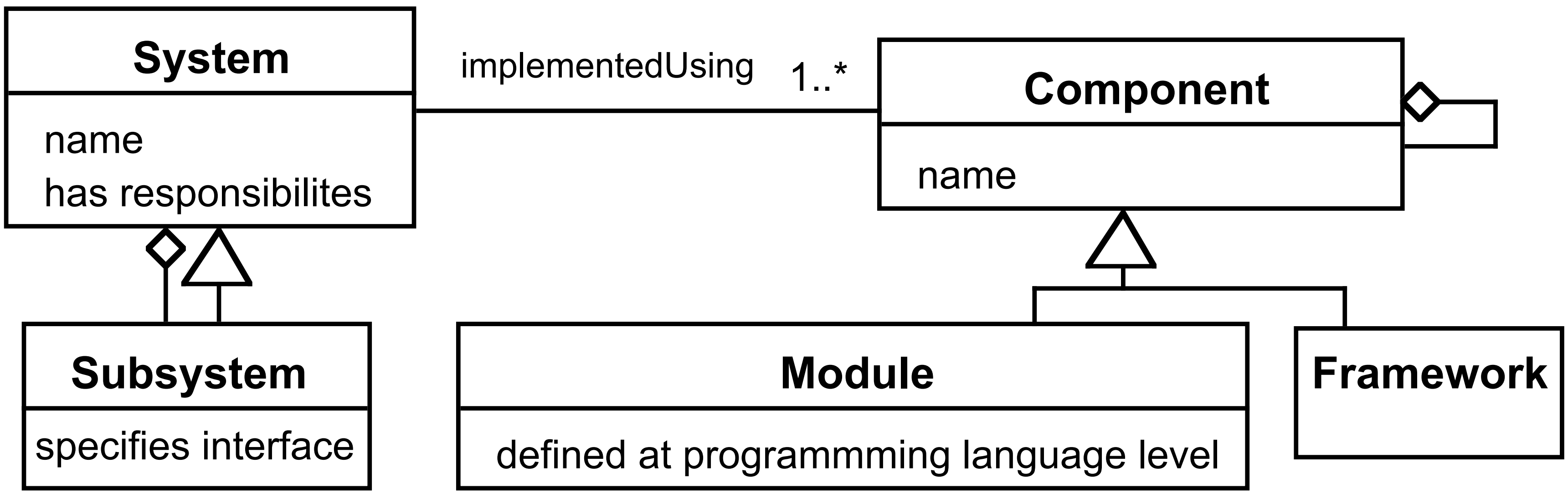
System

A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both.

- A system can have a specification which is then implemented by a collection of components.
- A system continues to exist, even if its components are changed or replaced.
- The goal of requirements analysis is to determine the responsibilities of a system.

- **Subsystem**:
  - A system that is part of a larger system, and which has a definite interface

UML Class Diagram of System Parts



Top-Down and Bottom-Up Design

Top-down design

- First design the very high level structure of the system.
- Then gradually work down to detailed decisions about low-level constructs.
- Finally arrive at detailed decisions such as:
  - the format of particular data items;
  - the individual algorithms that will be used.

Top-Down and Bottom-Up Design

Bottom-up design

- Make decisions about reusable low-level utilities.
- Then decide how these will be put together to create high-level constructs.

A mix of top-down and bottom-up approaches are normally used:

- Top-down design is almost always needed to give the system a good structure.
- Bottom-up design is normally useful so that reusable components can be created.

Different Aspects of Design

- *Architecture design*:
  - The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- *Class design*:
  - The various features of classes.
- *User interface design*
- *Algorithm design*:
  - The design of computational mechanisms.
- *Protocol design*:
  - The design of communications protocol.

9.2 Principles Leading to Good Design

Overall *goals* of good design:

- Increasing profit by reducing cost and increasing revenue
- Ensuring that we actually conform with the requirements
- Accelerating development
- Increasing qualities such as
  - Usability
  - Efficiency
  - Reliability
  - Maintainability
  - Reusability

Design Principle 1: Divide and Conquer

Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things

- Separate people can work on each part.
- An individual software engineer can specialize.
- Each individual component is smaller, and therefore easier to understand.
- Parts can be replaced or changed without having to replace or extensively change other parts.

Ways of Dividing a Software System

- A distributed system is divided up into clients and servers
- A system is divided up into subsystems
- A subsystem can be divided up into one or more packages
- A package is divided up into classes
- A class is divided up into methods

Design Principle 2: Increase **Cohesion** Where Possible

A subsystem or module has high cohesion if it *keeps together* things that are related to each other, and keeps out other things

- Measures the organization of a system
- Makes the system as a whole easier to understand and change
- Types of cohesion:
  - —Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Functional Cohesion

This is achieved when all the code that *computes a particular result* is kept together - and everything else is kept out

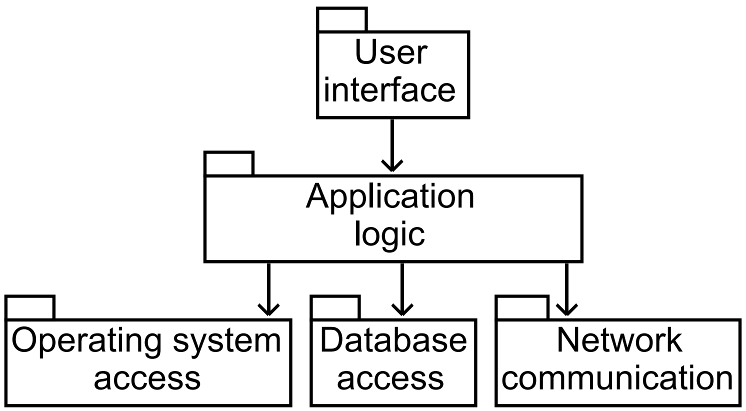
- i.e. when a module only performs a *single* computation, and returns a result, *without having side-effects*.
- Benefits to the system:
  - —Easier to understand
  - —More reusable
  - —Easier to replace
- Modules that update a database, create a new file or interact with the user are not functionally cohesive

Layer Cohesion

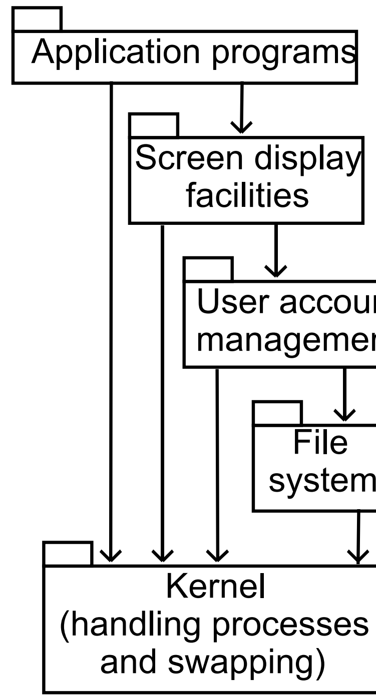
All the *facilities for providing or accessing a set of related services* are kept together, and everything else is kept out

- The layers should form a hierarchy
  - Higher layers can access services of lower layers,
  - Lower layers do not access higher layers
- The set of procedures through which a layer provides its services is the *application programming interface (API)*
- You can replace a layer without having any impact on the other layers
  - You just replicate the API

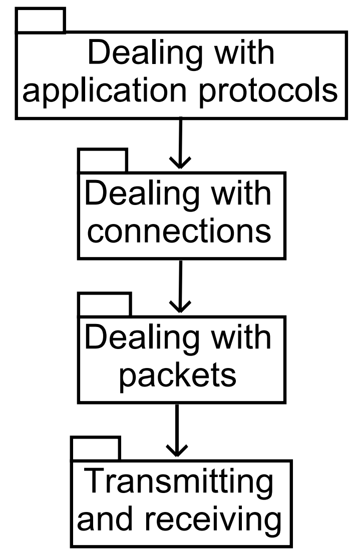
Examples of the Use of Layers



a) Typical layers in an application program



b) Typical layers in an operating system



c) Simplified view of layers in a communication system

Communicational Cohesion

All the modules that *access or manipulate certain data* are kept together (e.g. in the same class) - and everything else is kept out

- A class would have good communicational cohesion if
  - *all* the system's facilities for storing and manipulating its data are contained in this class.
  - the class does not do *anything other* than manage its data.
- Main advantage: When you need to make changes to the data, you find all the code in one place

Sequential Cohesion

Procedures, in which *one procedure provides input to the next*, are kept together – and everything else is kept out

- You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

Procedural Cohesion

Keep together several procedures that are *used one after another*

- Even if one does not necessarily provide input to the next.
- Weaker than sequential cohesion.

Temporal Cohesion

Operations that are performed during the *same phase of the execution* of the program are kept together, and everything else is kept out

- Used at a similar period of *time*
- For example, placing together the code used during system start-up or initialization.
- Weaker than procedural cohesion.

Utility Cohesion

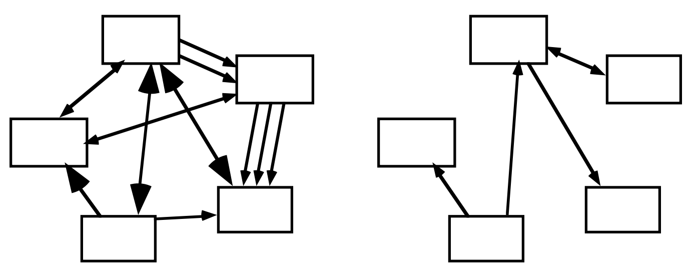
When related *utilities* which cannot be logically placed in other cohesive units are kept together

- A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be *reusable*.
- For example, the **`java.lang.Math`** class.

Design Principle 3: Reduce **Coupling** Where Possible

***Coupling* occurs when there are *interdependencies* between one module and another**

- When interdependencies exist, changes in one place will require changes somewhere else.
- A network of interdependencies makes it hard to see at a glance how some component works.
- Type of coupling:
  — Content, Common, Control, Stamp, Data, Routine Call, Type Use, Inclusion/Import, External



Content Coupling

Occurs when one component *surreptitiously* modifies data that is *internal* to another component

- To reduce content coupling you should therefore *encapsulate* all instance variables
  - declare them `private`
  - and provide get and set methods
- A worse form of content coupling occurs when you directly modify an instance variable *of* an instance variable

Example of Content Coupling

```java
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
}

public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
    Point theEnd = baseline.getEnd();
    theEnd.setLocation(theEnd.getX(),newY);
  }
}
```

Common Coupling

Occurs whenever you use a *global variable*

- All the components using the global variable become coupled to each other
- A weaker form of common coupling is when a variable can be accessed by a *subset* of the system's classes
    - —e.g. a Java package
- Can be acceptable for creating global variables that represent system-wide default values
- The Singleton pattern provides encapsulated global access to an object

Control Coupling

Occurs when one procedure calls another using a 'flag' or 'command' that explicitly controls what the second procedure does

- To make a change you have to change both the calling and called method
- The use of polymorphic operations is normally the best way to avoid control coupling
- One way to reduce the control coupling could be to have a *look-up table*
  - commands are then mapped to a method that should be called when that command is issued

Example of Control Coupling

```
public routineX(String command)
{
   if (command.equals("drawCircle")
   {
      drawCircle();
   }
   else
   {
      drawRectangle();
   }
}
```

Stamp Coupling

Occurs whenever one of your application classes is declared as the *type* of a method argument

- Since one class now uses the other, changing the system becomes harder
  - Reusing one class requires reusing the other

- Two ways to reduce stamp coupling,
  - using an interface as the argument type
  - passing simple variables

Example of Stamp Coupling

```
public class Emailer
{
  public void sendEmail(Employee e, String text)
  {...}
  ...
}
```

Using simple data types to avoid it:

```
public class Emailer
{
  public void sendEmail(
    String name, String email, String text)
  {...}
  ...
}
```

Example of Stamp Coupling

Using an interface to avoid it:

```
public interface Addressee
{
  public abstract String getName();
  public abstract String getEmail();
}

public class Employee implements Addressee {…}

public class Emailer
{
  public void sendEmail(
    Addressee e, String text)
  {...}
  ...
}
```

Data Coupling

Occurs whenever the types of method arguments are either primitive or else simple library classes

- The more arguments a method has, the higher the coupling
  - All methods that use the method must pass all the arguments
- You should reduce coupling by not giving methods unnecessary arguments

- There is a trade-off between data coupling and stamp coupling
  - Increasing one often decreases the other

Routine Call Coupling

Occurs when one routine (or method in an object oriented system) calls another

- The routines are coupled because they depend on each other's behaviour
- Routine call coupling is always present in any system.

- If you repetitively use a sequence of two or more methods to compute something
  - —then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.

Type Use Coupling

Occurs when a module uses a data type defined in another module

- It occurs any time a class declares an instance variable or a local variable as having another class for its type.
- The consequence of type use coupling is that if the type definition changes, then the users of the type may have to change
- Always declare the type of a variable to be the most general possible class or interface that contains the required operations

Inclusion or Import Coupling

Occurs when one component imports a package

- (as in Java)

or when one component includes another

- (as in C++).
- The including or importing component is now exposed to everything in the included or imported component.
- If the included/imported component changes something or adds something.
  - This may raises a conflict with something in the includer, forcing the includer to change.
- An item in an imported component might have the same name as something you have already defined.

External Coupling

When a module has a dependency on such things as the operating system, shared libraries or the hardware

- It is best to reduce the number of places in the code where such dependencies exist.
- The Façade design pattern can reduce external coupling

Design Principle 4: Keep the Level of Abstraction as High as Possible

Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity

- A good abstraction is said to provide *information hiding*
- Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

Abstraction and Classes

Classes are data abstractions that contain procedural abstractions

- Abstraction is increased by defining all variables as private.
- The fewer public methods in a class, the better the abstraction
- Superclasses and interfaces increase the level of abstraction
- Attributes and associations are also data abstractions.
- Methods are procedural abstractions
  — Better abstractions are achieved by giving methods fewer parameters

Design Principle 5: Increase Reusability Where Possible

Design the various aspects of your system so that they can be used again in other contexts

- Generalize your design as much as possible
- Follow the preceding three design principles
- Design your system to contain hooks
- Simplify your design as much as possible

Design Principle 6: Reuse Existing Designs and Code Where Possible

Design with reuse is complementary to design for reusability

- Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
    - —*Cloning* should not be seen as a form of reuse

Design Principle 7: Design for Flexibility

Actively anticipate changes that a design may have to undergo in the future, and prepare for them

- Reduce coupling and increase cohesion
- Create abstractions
- Do not hard-code anything
- Leave all options open
  - Do not restrict the options of people who have to modify the system later
- Use reusable code and make code reusable

Design Principle 8: Anticipate Obsolescence

Plan for changes in the technology or environment so the software will continue to run or can be easily changed

- Avoid using early releases of technology
- Avoid using software libraries that are specific to particular environments
- Avoid using undocumented features or little-used features of software libraries
- Avoid using software or special hardware from companies that are less likely to provide long-term support
- Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

Have the software run on as many platforms as possible

- Avoid the use of facilities that are specific to one particular environment
- E.g. a library only available in Microsoft Windows

Design Principle 10: Design for Testability

Take steps to make testing easier

- Design a program to automatically test the software
  - Discussed more in Chapter 10
  - Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
- In Java, you can create a main() method in each class in order to exercise the other methods

Design Principle 11: Design Defensively

Never trust how others will try to use a component you are designing

- Handle all cases where other code might attempt to use your component inappropriately
- Check that all of the inputs to your component are valid: the *preconditions*
  - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

Design by Contract

A technique that allows you to design defensively in an efficient and systematic way

- Key idea
  - each method has an explicit *contract* with its callers
- The contract has a set of assertions that state:
  - What *preconditions* the called method requires to be true when it starts executing
  - What *postconditions* the called method agrees to ensure are true when it finishes executing
  - What *invariants* the called method agrees will not change as it executes

9.3 Techniques for Making Good Design Decisions

Using priorities and objectives to decide among alternatives

- Step 1: List and describe the alternatives for the design decision.
- Step 2: List the advantages and disadvantages of each alternative with respect to your objectives and priorities.
- Step 3: Determine whether any of the alternatives prevents you from meeting one or more of the objectives.
- Step 4: Choose the alternative that helps you to best meet your objectives.
- Step 5: Adjust priorities for subsequent decision making.

Example Priorities and Objectives

Imagine a system has the following objectives, starting with top priority:

- **Security**: Encryption must not be breakable within 100 hours of computing time on a 400Mhz Intel processor, using known cryptanalysis techniques.
- **Maintainability**. No specific objective.
- **CPU efficiency**. Must respond to the user within one second when running on a 400MHz Intel processor.
- **Network bandwidth efficiency**: Must not require transmission of more than 8KB of data per transaction.
- **Memory efficiency**. Must not consume over 20MB of RAM.
- **Portability.** Must be able to run on Windows 98, NT 4, ME and XP as well as Linux

Example Evaluation of Alternatives

	Security	Maintain-ability	Memory efficiency	CPU efficiency	Bandwidth efficiency	Portability
Algorithm A	High	Medium	High	Medium; NO	Low	Low
Algorithm B	High	High	Low	Medium; NO	Medium	Low
Algorithm C	High	High	High	Low; NO	High	Low
Algorithm D				Medium; NO	NO	
Algorithm E	NO			Low; NO		

'NO' means that the objective is not met

Using Cost-Benefit Analysis to Choose Among Alternatives

- To estimate the *costs*, add up:
  - The incremental cost of doing the *software engineering* work, including ongoing maintenance
  - The incremental costs of any *development technology* required
  - The incremental costs that *end-users and product support personnel* will experience
- To estimate the *benefits*, add up:
  - The incremental software engineering time saved
  - The incremental benefits measured in terms of either increased sales or else financial benefit to users

9.4 Software Architecture

Software architecture **is process of designing the global organization of a software system, including:**

- Dividing software into subsystems.
- Deciding how these will interact.
- Determining their interfaces.
  - The architecture is the core of the design, so all software engineers need to understand it.
  - The architecture will often constrain the overall efficiency, reusability and maintainability of the system.

The Importance of Software Architecture

Why you need to develop an architectural model:

- To enable everyone to better understand the system
- To allow people to work on individual pieces of the system in isolation
- To prepare for extension of the system
- To facilitate reuse and reusability

Contents of a Good Architectural Model

A system's architecture will often be expressed in terms of several different *views*

- The logical breakdown into subsystems
- The interfaces among the subsystems
- The dynamics of the interaction among components at run time
- The data that will be shared among the subsystems
- The components that will exist at run time, and the machines or devices on which they will be located

Design Stable Architecture

To ensure the maintainability and reliability of a system, an architectural model must be designed to be *stable*.

- Being stable means that the new features can be easily added with only small changes to the architecture

Developing an Architectural Model

Start by sketching an outline of the architecture

- Based on the principal requirements and use cases
- Determine the main components that will be needed
- Choose among the various architectural patterns
  - —Discussed next
- *Suggestion*: have several different teams independently develop a first draft of the architecture and merge together the best ideas

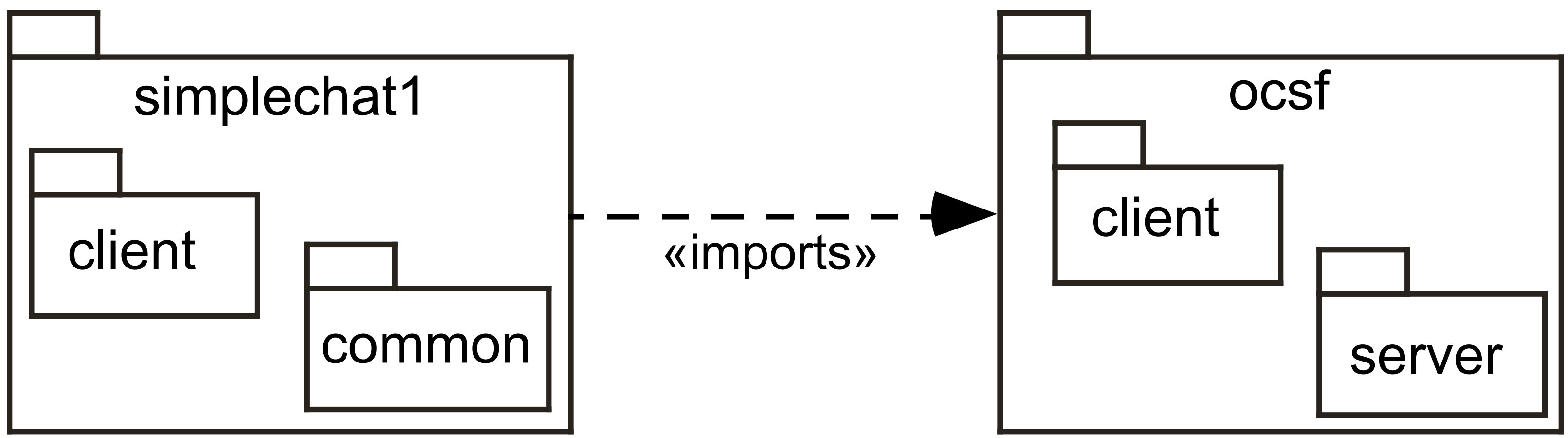
Developing an Architectural Model

- Refine the architecture
  - Identify the main ways in which the components will interact and the interfaces between them
  - Decide how each piece of data and functionality will be distributed among the various components
  - Determine if you can re-use an existing framework, if you can build a framework
- Consider each use case and adjust the architecture to make it realizable
- Mature the architecture

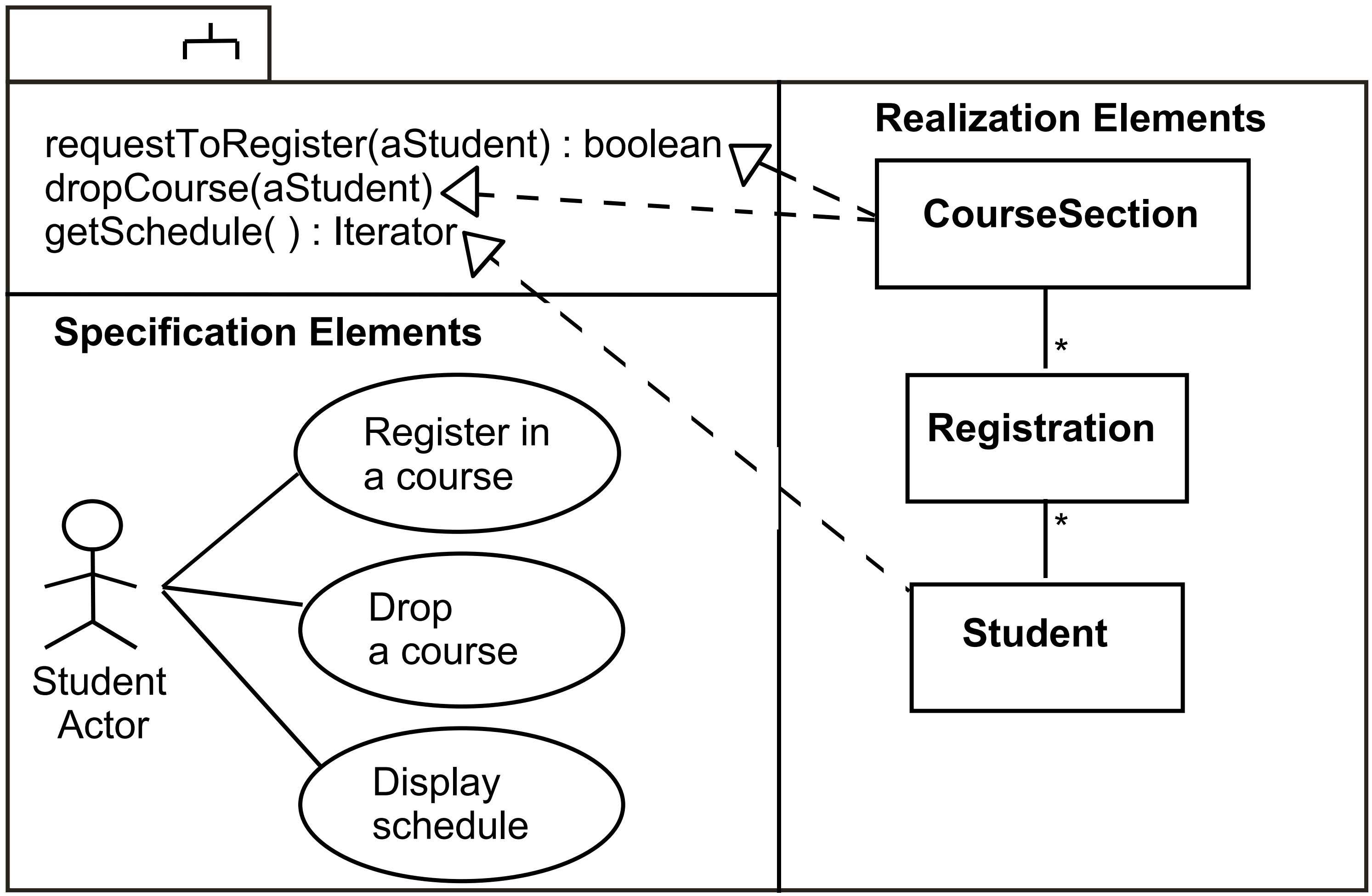
Describing an Architecture Using UML

- All UML diagrams can be useful to describe aspects of the architectural model
- Four UML diagrams are particularly suitable for architecture modelling:
  - Package diagrams
  - Subsystem diagrams
  - Component diagrams
  - Deployment diagrams

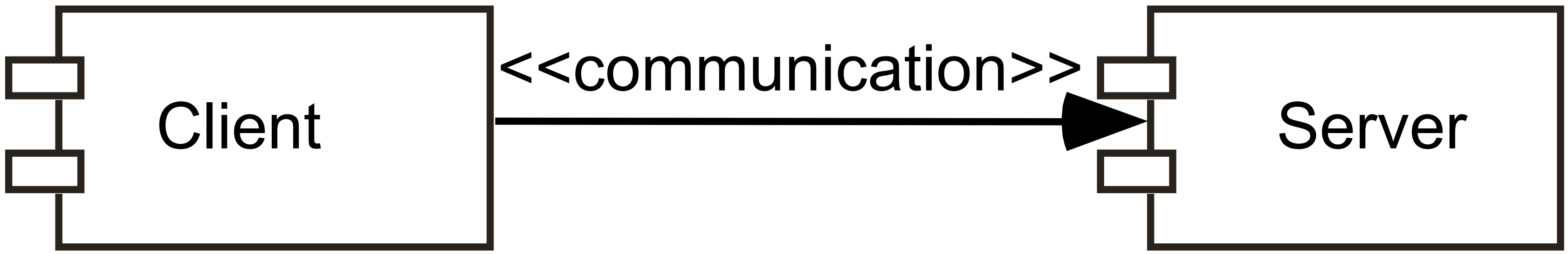
Package Diagrams



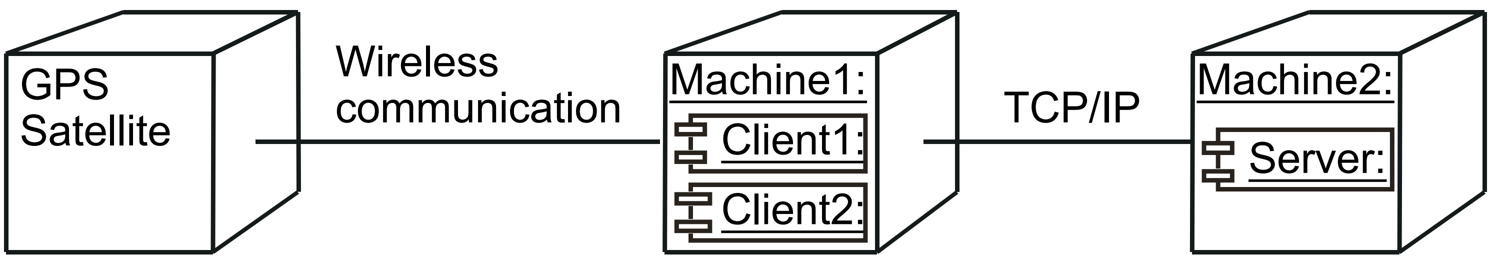
Subsystem Diagrams



Component Diagrams



Deployment Diagrams



9.5 Architectural Patterns

The notion of patterns can be applied to software architecture.

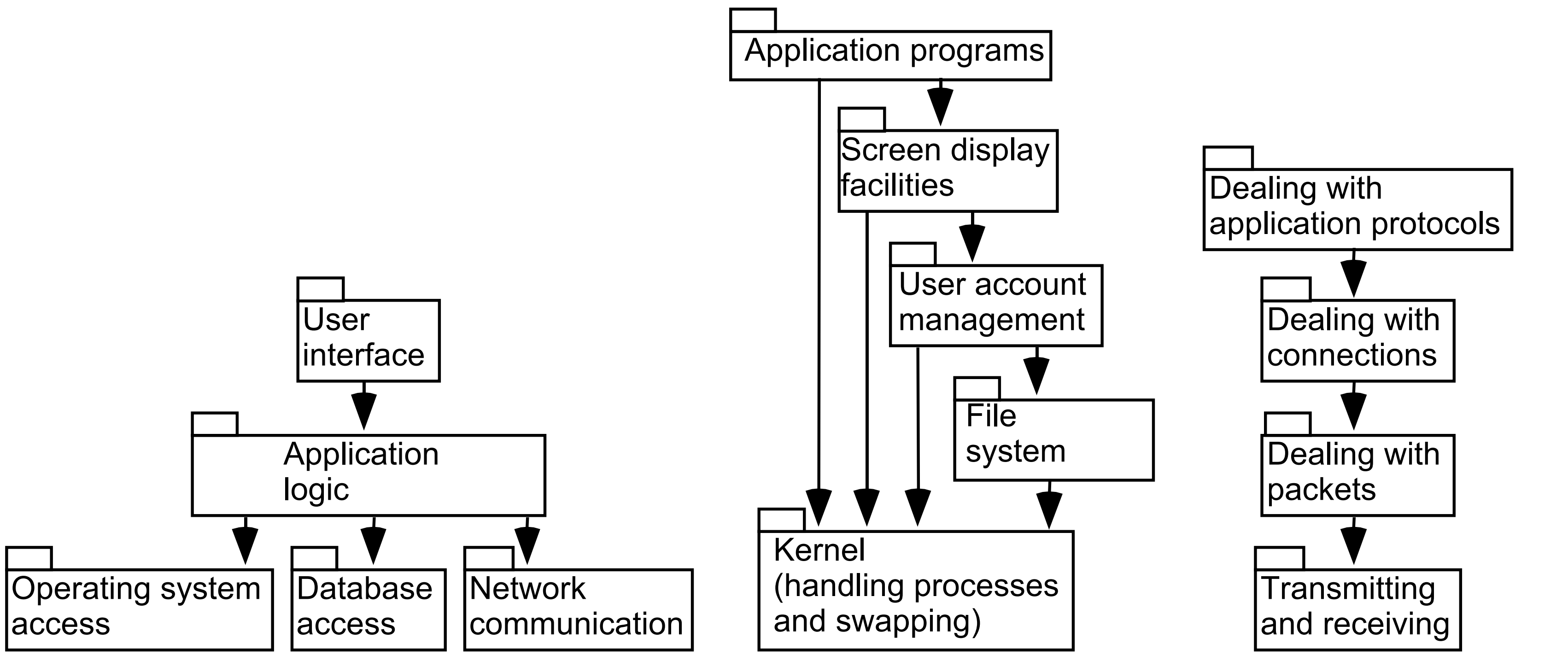
- These are called *architectural patterns* or *architectural styles*.
- Each allows you to design flexible systems using components
  - The components are as independent of each other as possible.

The Multi-Layer Architectural Pattern

In a layered system, each layer communicates only with the layer immediately below it.

- Each layer has a well-defined interface used by the layer immediately above.
  - —The higher layer sees the lower layer as a set of *services*.
- A complex system can be built by superposing layers at increasing levels of abstraction.
  - —It is important to have a separate layer for the UI.
  - —Layers immediately below the UI layer provide the application functions determined by the use-cases.
  - —Bottom layers provide general services.
    - e.g. network communication, database access

Example of Multi-Layer Systems - Layer Cohesion Revisited



a) Typical layers in an application program

b) Typical layers in an operating system

c) Simplified view of layers in a communication system

The Multi-Layer Architecture - Design Principles

1. *Divide and conquer*: The layers can be independently designed.
2. *Increase cohesion*: Well-designed layers have layer cohesion.
3. *Reduce coupling*: Well-designed lower layers do not know about the higher layers and the only connection between layers is through the API.
4. *Increase abstraction*: you do not need to know the details of how the lower layers are implemented.
5. *Increase reusability*: The lower layers can often be designed generically.

The Multi-Layer Architecture - Design Principles Continued

6. *Increase reuse*: You can often reuse layers built by others that provide the services you need.
7. Increase flexibility: you can add new facilities built on lower-level services, or replace higher-level layers.
8. *Anticipate obsolescence*: By isolating components in separate layers, the system becomes more resistant to obsolescence.
9. *Design for portability*: All the dependent facilities can be isolated in one of the lower layers.
10. *Design for testability*: Layers can be tested independently.
11. *Design defensively*: The APIs of layers are natural places to build in rigorous assertion-checking.

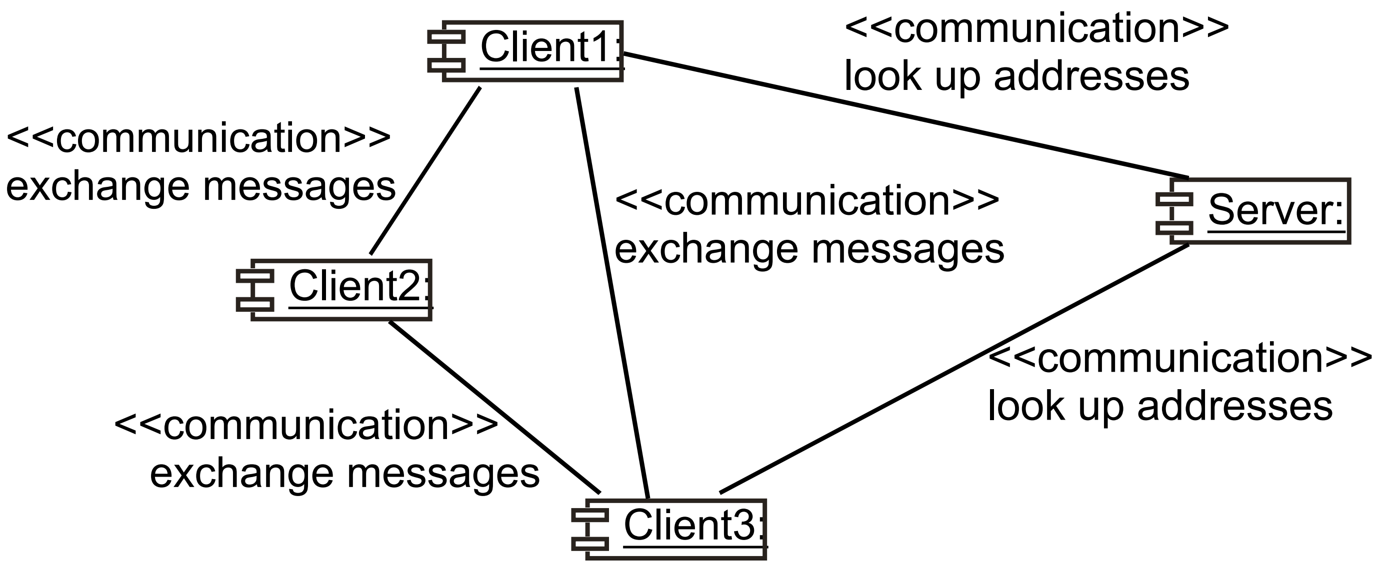
The Client-Server and Other Distributed Architectural Patterns

- There is at least one component that has the role of *server*, waiting for and then handling connections.

- There is at least one component that has the role of *client*, initiating connections in order to obtain some service.

- A further extension is the Peer-to-Peer pattern.
  - A system composed of various software components that are distributed over several hosts.

An Example of a Distributed System



The Distributed Architecture - Design Principles

1. *Divide and conquer*: Dividing the system into client and server processes is a strong way to divide the system.
   — Each can be separately developed.
2. *Increase cohesion*: The server can provide a cohesive service to clients.
3. *Reduce coupling*: There is usually only one communication channel exchanging simple messages.
4. *Increase abstraction*: Separate distributed components are often good abstractions.
6. *Increase reuse*: It is often possible to find suitable frameworks on which to build good distributed systems
   — However, client-server systems are often very application specific.

The Distributed Architecture - Design Principles Continued

7. *Design for flexibility*: Distributed systems can often be easily reconfigured by adding extra servers or clients.

9. *Design for portability*: You can write clients for new platforms without having to port the server.

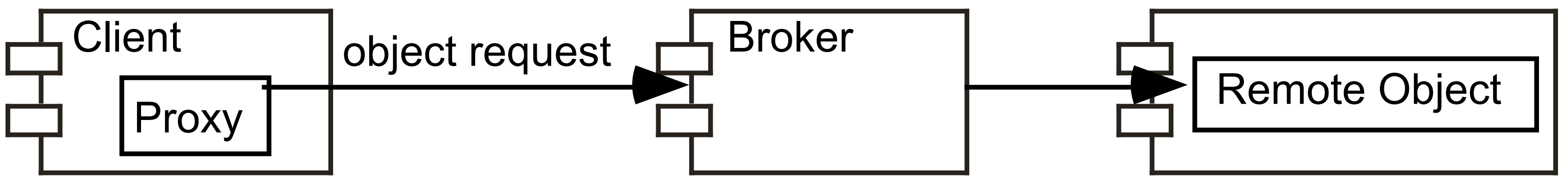
10 *Design for testability*: You can test clients and servers independently.

11. *Design defensively*: You can put rigorous checks in the message handling code.

The Broker Architectural Pattern

- Transparently distribute aspects of the software system to different nodes
  - An object can call methods of another object without knowing that this object is remotely located.
  - CORBA is a well-known open standard that allows you to build this kind of architecture.

Example of a Broker System



The Broker Architecture - Design Principles

1. *Divide and conquer*: The remote objects can be independently designed.

5. *Increase reusability*: It is often possible to design the remote objects so that other systems can use them too.

6. *Increase reuse*: You may be able to reuse remote objects that others have created.

7. *Design for flexibility*: The brokers can be updated as required, or the proxy can communicate with a different remote object.

9. *Design for portability*: You can write clients for new platforms while still accessing brokers and remote objects on other platforms.

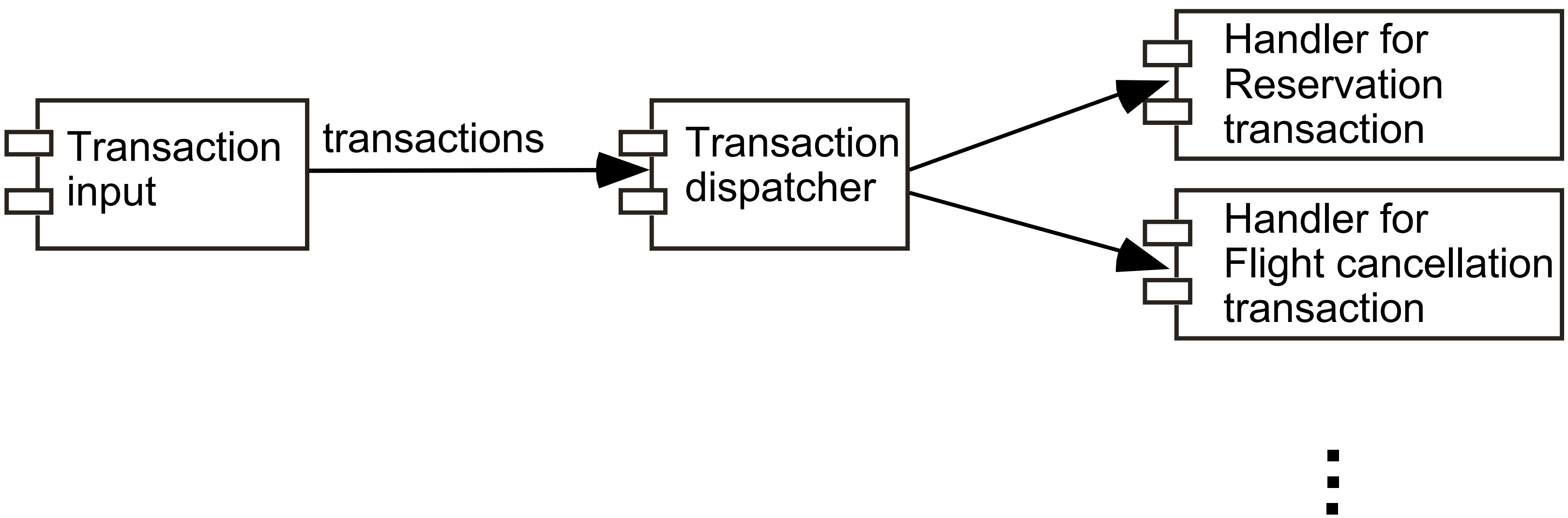
11. *Design defensively*: You can provide careful assertion checking in the remote objects.

The Transaction-Processing Architectural Pattern

A process reads a series of inputs one by one.

- Each input describes a *transaction* – a command that typically some change to the data stored by the system
- There is a transaction *dispatcher* component that decides what to do with each transaction
- This dispatches a procedure call or message to one of a series of component that will *handle* the transaction

Example of a Transaction-Processing System - Airline Reservations



The Transaction-Processing Architecture - Design Principles

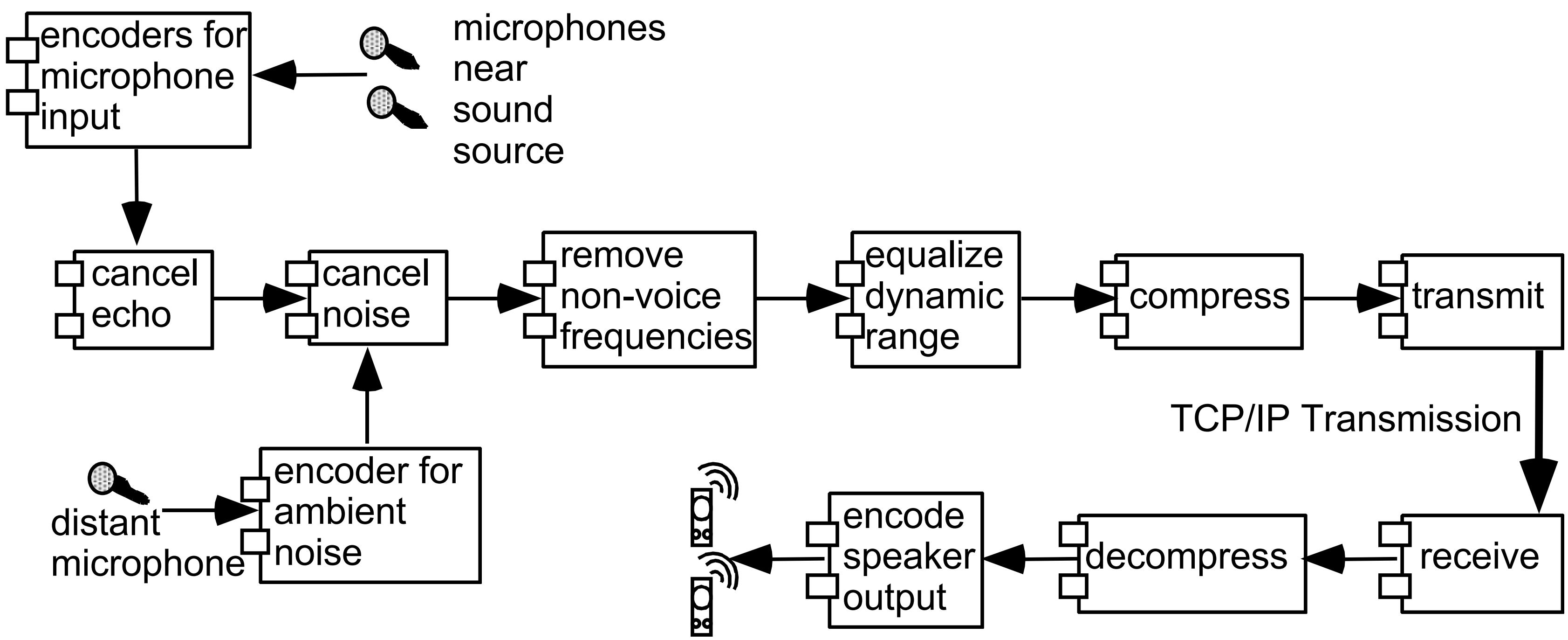
1. *Divide and conquer*: The transaction handlers are suitable system divisions that you can give to separate software engineers.
2. *Increase cohesion*: Transaction handlers are naturally cohesive units.
3. *Reduce coupling*: Separating the dispatcher from the handlers tends to reduce coupling.
7. *Design for flexibility*: You can readily add new transaction handlers.
11. *Design defensively*: You can add assertion checking in each transaction handler and/or in the dispatcher.

The Pipe-and-Filter Architectural Pattern

A stream of data, in a relatively simple format, is passed through a series of processes

- Each of which transforms it in some way.
- Data is constantly fed into the pipeline.
- The processes work concurrently.
- The architecture is very flexible.
  - Almost all the components could be removed.
  - Components could be replaced.
  - New components could be inserted.
  - Certain components could be reordered.

Example of a Pipe-and-Filter System - Sound Processing



The Pipe-and-Filter Architecture - Design Principles

1. *Divide and conquer*: The separate processes can be independently designed.
2. *Increase cohesion*: The processes have functional cohesion.
3. *Reduce coupling*: The processes have only one input and one output.
4. *Increase abstraction*: The pipeline components are often good abstractions, hiding their internal details.
5. *Increase reusability*: The processes can often be used in many different contexts.
6. *Increase reuse*: It is often possible to find reusable components to insert into a pipeline.

The Pipe-and-Filter Architecture - Design Principles Continued

7. *Design for flexibility*: There are several ways in which the system is flexible.

10. *Design for testability*: It is normally easy to test the individual processes.

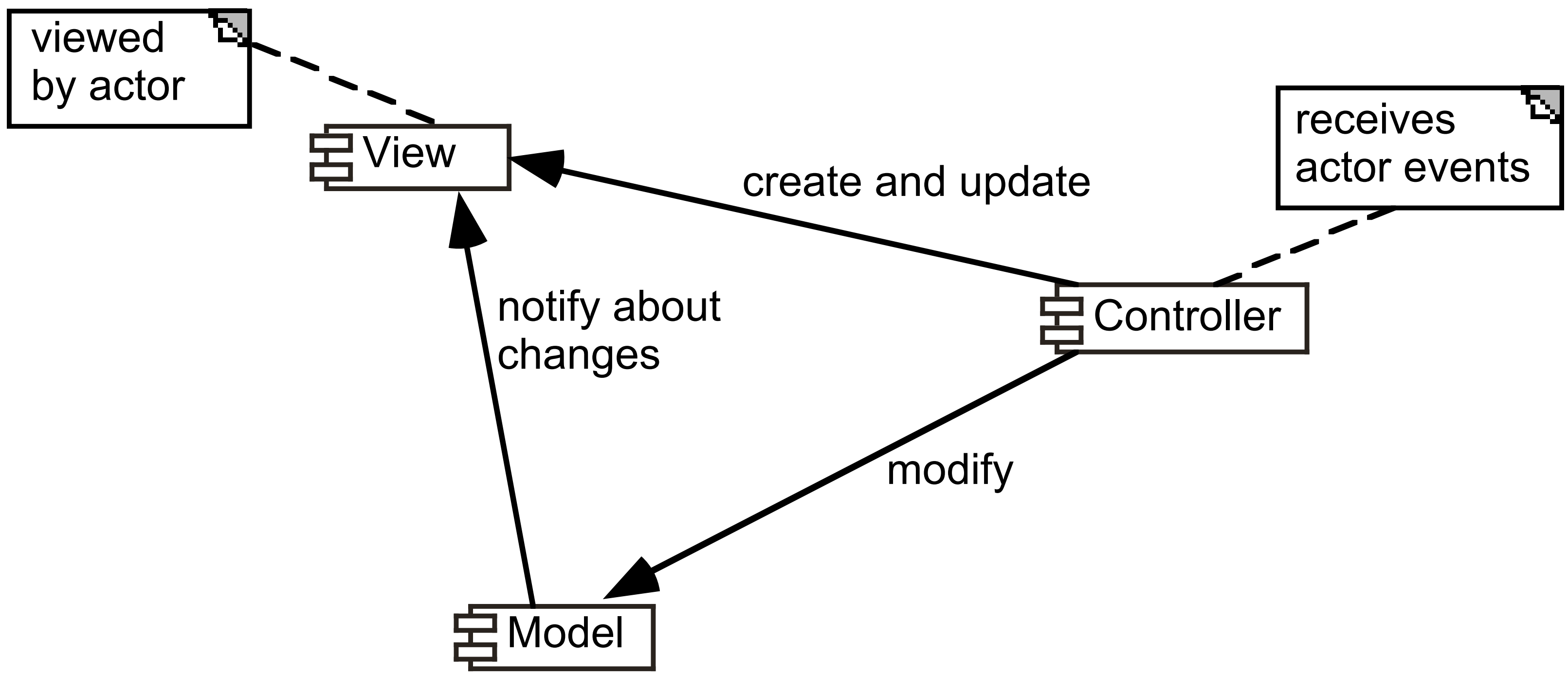
11. *Design defensively*: You rigorously check the inputs of each component, or else you can use design by contract.

The Model-View-Controller (MVC) Architectural Pattern

An architectural pattern used to help separate the user interface layer from other parts of the system

- The *model* contains the underlying classes whose instances are to be viewed and manipulated
- The *view* contains objects used to render the appearance of the data from the model in the user interface
- The *controller* contains the objects that control and handle the user's interaction with the view and the model
- The Observable design pattern is normally used to separate the model from the view

Example of the MVC Architecture for a User Interface



The MVC Architecture - Design Principles

1. *Divide and conquer*: The three components can be somewhat independently designed.
2. *Increase cohesion*: The components have stronger layer cohesion than if the view and controller were together in a single UI layer.
3. *Reduce coupling*: The communication channels between the three components are minimal.
6. *Increase reuse*: The view and controller normally make extensive use of reusable components for various kinds of UI controls.
7. *Design for flexibility*: It is usually quite easy to change the UI by changing the view, the controller, or both.
10. *Design for testability*: You can test the application separately from the UI.

9.6 Writing a Good Design Document

Design documents as an aid to making better designs

- They force you to be explicit and consider the important issues before starting implementation.
- They allow a group of people to review the design and therefore to improve it.
- Design documents as a means of communication.
  - To those who will be *implementing* the design.
  - To those who will need, in the future, to *modify* the design.
  - To those who need to create systems or subsystems that *interface* with the system being designed.

Structure of a Design Document

A. Purpose:

- What system or part of the system this design document describes.
- Make reference to the requirements that are being implemented by this design (*traceability)* .

B. General priorities:

- Describe the priorities used to guide the design process. □

C. Outline of the design:

- Give a high-level description of the design that allows the reader to quickly get a general feeling for it. □

D. Major design issues:

- Discuss the important issues that had to be resolved.
- Give the possible alternatives that were considered, the final decision and the rationale for the decision.

E. Other details of the design:

- Give any other details the reader may want to know that have not yet been mentioned.

When Writing the Document: Some General Rules about What to Exclude

- Avoid documenting information that would be *readily obvious* to a skilled programmer or designer.
- Avoid writing details in a design document that would be better placed as *comments* in the code.
- Avoid writing details that can be *extracted automatically* from the code, such as the list of public methods.

9.7 Design of a Feature of the SimpleChat System

A. Purpose

This document describes important aspects of the implementation of the **`#block`**, **`#unblock`**, **`#whoiblock`** and **`#whoblocksme`** commands of the SimpleChat system.

B. General Priorities

Decisions in this document are made based on the following priorities (most important first): Maintainability, Usability, Portability, Efficiency

C. Outline of the design

Blocking information will be maintained in the **`ConnectionToClient`** objects. The various commands will update and query the data using **`setValue`** and **`getValue`**.

Design Example - Issues

D. Major design issue

Issue 1: Where should we store information regarding the establishment of blocking?

- **Option 1.1**: Store the information in the **`ConnectionToClient`** object associated with the client requesting the block.
- **Option 1.2**: Store the information in the **`ConnectionToClient`** object associated with the client that is being blocked.
- **Decision**: Point 2.2 of the specification requires that we be able to block a client even if that client is not logged on. This means that we must choose option 1.1 since no **`ConnectionToClient`** will exist for clients that are logged off.

Design Example - Details - Client Side

E. Details of the design:

Client side:

□

- The four new commands will be accepted by **`handleMessageFromClientUI`** and passed unchanged to the server.

- Responses from the server will be displayed on the UI. There will be no need for **`handleMessageFromServer`** to understand that the responses are replies to the commands.

□

Design Example - Details - Server Side

Server side:

- Method **`handleMessageFromClient`** will interpret **`#block`** commands by adding a record of the block in the data associated with the originating client.

  This method will modify the data in response to **`#unblock`**.

- The information will be stored by calling **`setValue("blockedUsers", arg)`**

  where **`arg`** is a **`Vector`** containing the names of the blocked users.

- Method **`handleMessageFromServerUI`** will also have to have an implementation of **`#block`** and **`#unblock`**.

  These will have to save the blocked users as elements of a new instance variable declared thus: **`Vector blockedUsers;`**

Design Example - Continued

- The implementations of **`#whoiblock`** in **`handleMessageFromClient`** and **`handleMessageFromServerUI`** will straightforwardly process the contents of the vectors.
- For **`#whoblocksme`**, a new method will be created in the server class that will be called by both **`handleMessageFromClient`** and **`handleMessageFromServerUI`**.

  This will take a single argument (the name of the initiating client, or else 'SERVER').

  It will check all the **`blockedUsers`** vectors of the connected clients and also the **`blockedUsers`** instance variable for matching clients.

Design Example - Conclusion

- The **`#forward`**, **`#msg`** and **`#private`** commands will be modified as needed to reflect the specifications.

    Each of these will each examine the relevant **`blockedUsers`** vectors and take appropriate action.

9.8 Difficulties and Risks in Design

Like modelling, design is a skill that requires considerable experience

— *Individual software engineers should not attempt the design of large systems*

— *Aspiring software architects should actively study designs of other systems*

Poor designs can lead to expensive maintenance

— *Ensure you follow the principles discussed in this chapter*

Difficulties and Risks in Design

It requires constant effort to ensure a software system's design remains good throughout its life

— *Make the original design as flexible as possible so as to anticipate changes and extensions.*

— *Ensure that the design documentation is usable and at the correct level of detail*

— *Ensure that change is carefully managed*