

Classes 17-20

See document: [CS3733 SoftwareEngineering.mmap](#)

Classes 17-20	1
Class 17	1
Design	1
Solution domain	2
Is coding involved	2
Architecture	2
Definition	2
More than structure	2
Multiple views	2
Layers	3
Client-server	3
Three-tier	4
Thin-client	4
Subtopic	4
OSI 7-layers	4
Class 18	5
Design Patterns	5
Background	5
What are design patterns?	5
Describing patterns	5
Sample patterns	6
Adapter	6
Facade	6
Singleton	7
Class 19	7
Dependencies	7
Refactoring	8
Class 20	8
Subtopic	8

Class 17

Design

When you consider software design, you will normally start from the "big picture" and work down to the details. You have already developed a picture of the problem domain when you defined the requirements and built your analysis model. The analysis model gives you an understanding of what needs to be done, in enough detail to design a solution.

Design encompasses many activities and requires several skills. Many of these skills are fine-tuned through practice and apprenticeship with expert designers.

Solution domain

Once you have analyzed part of the problem, you can then design it. It is foolish to try to design a solution for a problem you don't understand. So we can consider that design is part of the solution domain, whereas analysis is in the problem domain.

Since you are working iteratively, you will be bouncing back and forth from analysis to design, but you must understand that the purpose of analysis is to understand the problem. The purpose of design is to create an acceptable solution to the problem.

Is coding involved

Design does not mean that you simply draw diagrams and make plans. When you design, you need to know that your design can actually be implemented. So, as a designer you will produce just enough code to make sure that your design works. Then you will refine that code during implementation.

Architecture

One of the major activities of design is creating an architecture that satisfies the requirements. Not only does the architecture have to satisfy the requirements, but it also must be implementable, and maintainable.

This is a tall order and requires a lot of knowledge of many technologies and practices. This is why software architects are in such demand. As we have built more complex systems, we learned the hard way that the system must be maintainable. It may exist for a long time. During its lifetime, we will need to add new features, change technology, fix defects, and so on. If we don't have a flexible, resilient architecture, the system can easily become brittle and come crashing down upon us.

Definition

There are many definitions of what we mean by software architecture. The following is taken from the RUP glossary:

The highest level concept of a system in its environment, according to *IEEE*. The architecture of a software system (at a given point in time) is its organization or structure of significant *components* interacting through *interfaces*, those components being composed of successively smaller components and interfaces.

More than structure

There is a misconception that the architecture of software is about structure only. The fact is that architecture is about much more. Architecture concerns itself with the whole system. A good architect considers the purpose for which the system is being built and then identifies the technology, structure, behavior, and other facets of the system that match the requirements.

Architecture also includes the decisions that are made while building the system. This is called the *design rationale*. Too often the design rationale is hidden from the developers and engineers who will be responsible for the on-going maintenance of the system.

Multiple views

How do you represent an architecture? Is it just a set of UML diagrams that show how a system is put together? No. You express the architecture at many levels. In a classic paper, Philippe Kruchten describes the "4+1" views approach to describing the system architecture. He suggests that there are four views of the system that each illustrates a specific facet of the system. These four views are:

- **Logical view**, which contains the most important design classes and their organization into packages and subsystems, and the organization of these packages and subsystems into layers.
- **Implementation View**, which contains an overview of the implementation model and its organization in terms of modules into packages and layers. The allocation of packages and classes (from the Logical View) to the packages and modules of the Implementation View is also described.
- **Process View**, which contains the description of the tasks (process and threads) involved, their interactions and configurations, and the allocation of design objects and classes to tasks. This view need only be used if the system has a significant degree of concurrency.
- **Deployment View**, which contains the description of the various physical nodes for the most typical platform configurations, and the allocation of tasks (from the Process View) to the physical nodes. This view need only be used if the system is distributed.

All of these views are tied together by the fifth (+1) view, which is the **use case view** that contains the use cases and scenarios that describe the architecturally significant behavior of the system.

Layers

One important aspect of an architecture is the "layering" of a system. In general, layers are used to encapsulate pieces of the system in such a way that the objects in one layer communicate only with those layers above or below the layer.

By using layering, you reduce the overall complexity of the system and make it easier to understand and maintain the system. The following sections describe several common types of layering schemes.

Client-server

Taken from the [SEI web site](#):

The term client/server was first used in the 1980s in reference to personal computers (PCs) on a network. The actual client/server model started gaining acceptance in the late 1980s. The client/server software architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability, and scalability as compared to centralized, mainframe, time sharing computing.

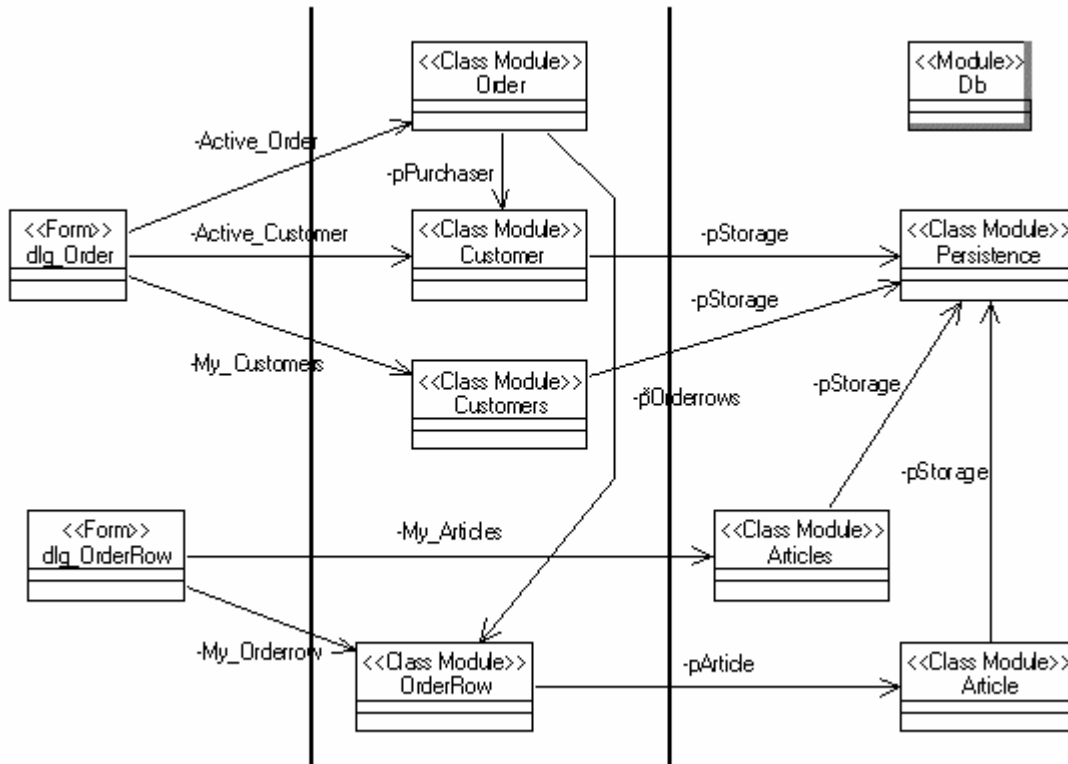
A client is defined as a requester of services and a server is defined as the provider of services. A single machine can be both a client and a server depending on the software configuration. For details on client/server software architectures see Schussel and Edelstein [Schussel 96, Edelstein 94].

This technology description provides a summary of some common client/server architectures and, for completeness, also summarizes mainframe and file sharing architectures. Detailed descriptions for many of the individual architectures are provided elsewhere in the document.

Three-tier

A three-tiered architecture has three layers, or tiers. These are the

- presentation layer that interfaces with the users
- application, or logical layer that contains the program logic
- data, or persistence layer that contains the required mechanisms for maintaining and persisting the data.



The three-tiered architecture is a type of client-server system.

Thin-client

When you place the application and data layers on the server and just place the presentation capability on the client system, it is known as a thin client.

This type of system can be used to easily distribute applications and update them. It is often used for web-hosted delivery. The downside of thin clients is the total dependence upon server availability and performance.

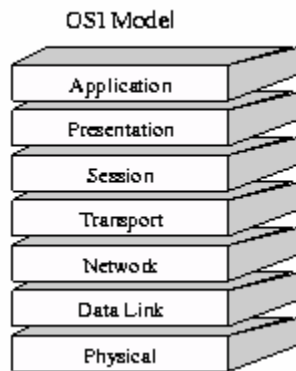
Subtopic

A thick-client is where you place the application layer as well as the presentation layer on the client system and have the data layer implemented on the server side of the system.

This may provide better response time at the expense of higher cost in client machine power, as well as the cost of updating the application logic.

OSI 7-layers

One concrete example of a layered architecture is the [OSI seven-layered model](#).



Class 18

Design Patterns

Design patterns are recurring solutions to software design problems you find again and again in real- world application development.

Background

Software design patterns were introduced first by Erich Gamma in his doctoral work at the University of Zurich. He based much of his work on the architectural (building) design patterns work of Christopher Alexander.

Gamma is the primary author of the seminal work, *Design Patterns*, which he co-authored with Ralph Johnson, John Vlissides, and Richard Helm.

What are design patterns?

People who are unfamiliar with design patterns often think of them as templates for implementing code. You just fill-in-the-blanks and you have a nice, well- designed code artifact. This is, however, poor understanding of what design patterns are all about.

Design patterns are an approach to solving a particular problem, *in a certain context*. Design patterns may be thought of as a *plan of attack* on solving the particular problem.

Describing patterns

We describe patterns using a format that is referred to as a *design pattern language*. It is not so much a language but a somewhat consistent way of describing the pattern. There are several different design pattern languages, but most have a lot in common with the others.

Some main sections of a pattern description that you are likely to encounter are:

- **Synopsis:** a description of the purpose of the pattern
- **Context:** when and where the pattern might be used. Sometimes this is a concrete example.
- **Forces:** what are the forces that would dictate your selecting this pattern over another
- **Solution:** how the pattern is applied in practice. Often this is shown with code and / or an accompanying UML diagram showing the participating objects and classes.
- **Consequences:** what are the benefits and disadvantages to using the pattern. Every choice you make when designing a solution to a problem will have post positive and negative consequences. You need to decide whether the benefits outweighs the disadvantages for your context.

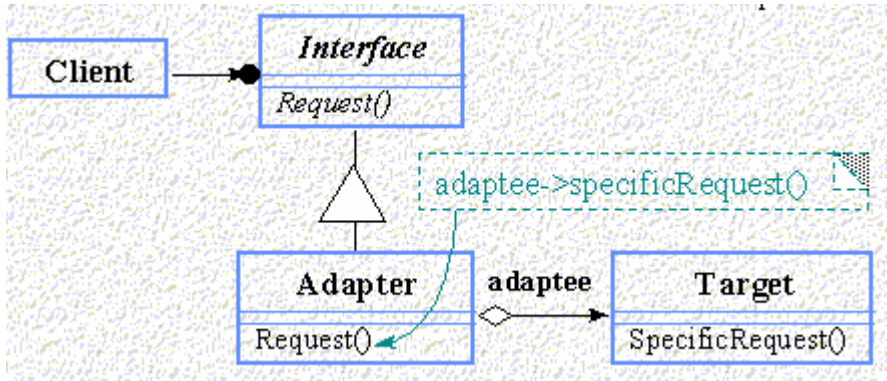
Sample patterns

The following are a couple of design patterns that you should be familiar with. For each, there is a brief description. You can use this link (<http://www.patterndigest.com/>) to find out more about the patterns here. You should also have read about some of the patterns in your textbook.

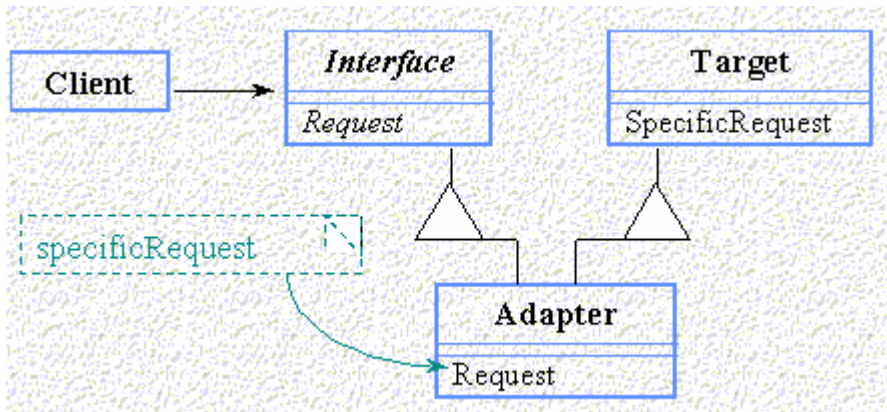
Adapter

The adapter pattern is designed to allow classes that were designed for different purposes, or with different interfaces, to be able to collaborate.

There are two kinds of adapters, the object adapter



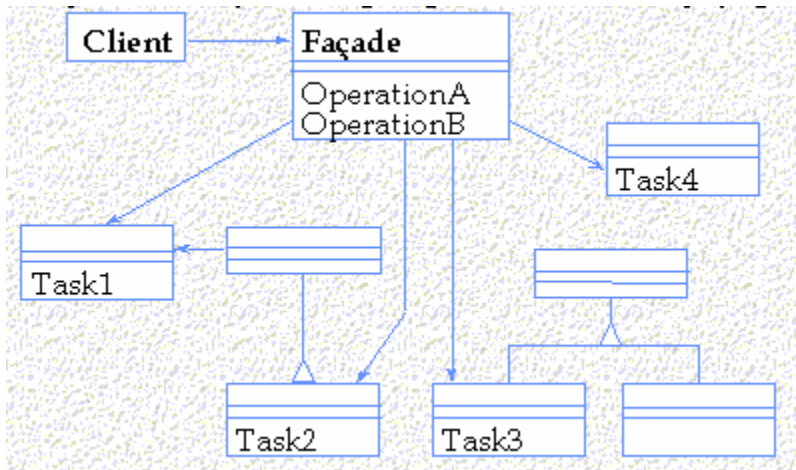
and the class adapter



The first performs adaptation by delegating the service request while the second uses multiple inheritance to do the same job.

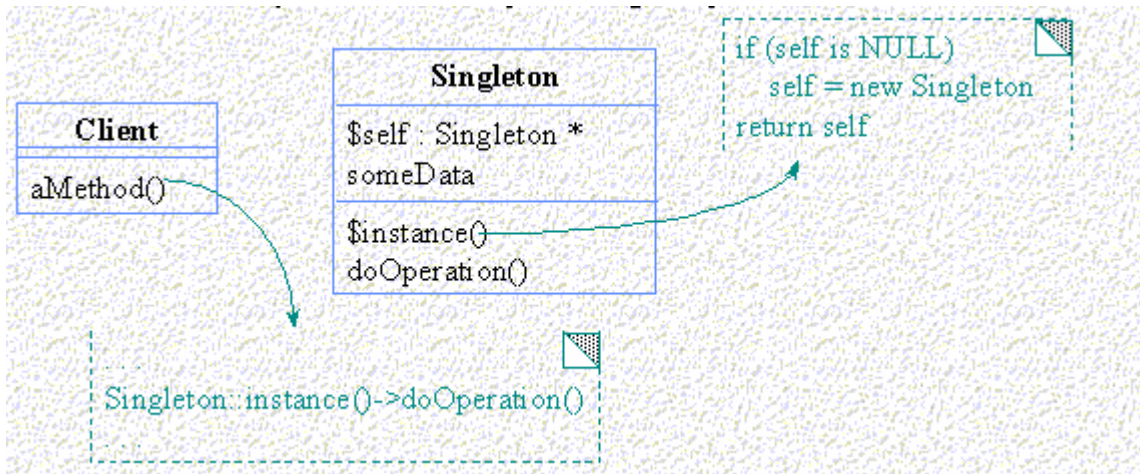
Facade

Facade a subsystem using a high-level interface, simplifying subsystem usage and hiding structural details. It is often used to define the interface to a layer of an architecture.



Singleton

The Singleton pattern is used to ensure that there is just one object of a given type during program execution.



Class 19

Dependencies

Dependency between classes, or packages, means that a change in one can cause a change in the other. In UML, dependency is represented as a dashed line with an open arrow on the independent element. So, if A depends upon B, there is a dashed line from A to B with the arrow on B.

A good system will limit the dependencies, and avoid cycles in dependencies. Cycles in dependencies is a good indication of a poorly designed system.

See the following links for information on dependencies and how to remove cycles from dependencies:

- [Wikipedia](#)
- [Compuware Java Central](#)

Refactoring

From [Wikipedia](#):

In software engineering, the term refactoring is often used to describe modifying source code without changing its external behavior, and is sometimes informally referred to as "cleaning it up". Refactoring is often practiced as part of the software development cycle: developers alternate between adding new tests and functionality and refactoring the code to improve its internal consistency and clarity. Testing ensures that refactoring does not change the behavior of the code.

Class 20

Subtopic

The [slides for this lecture](#) have been posted. Refer to them.