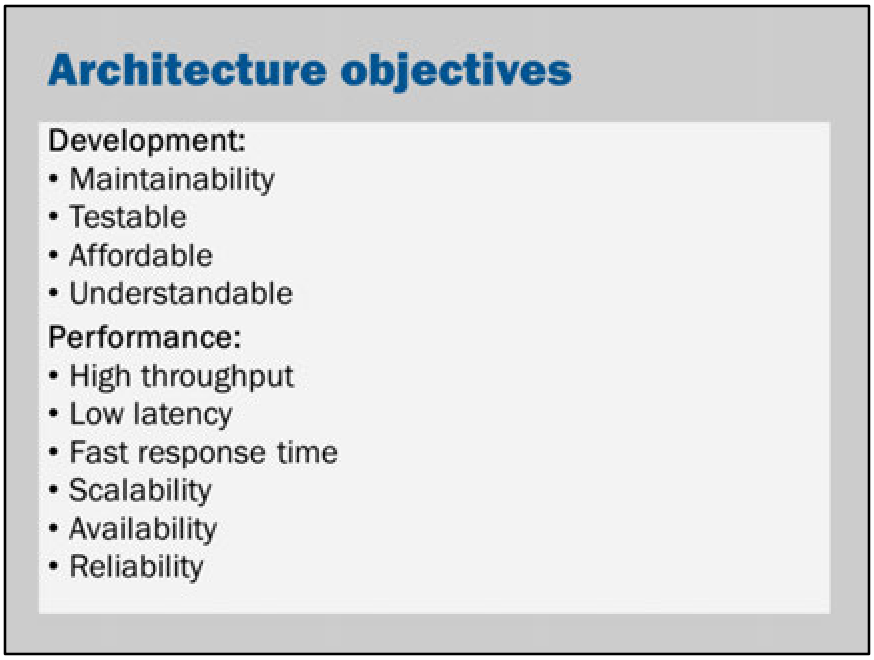
Architecture



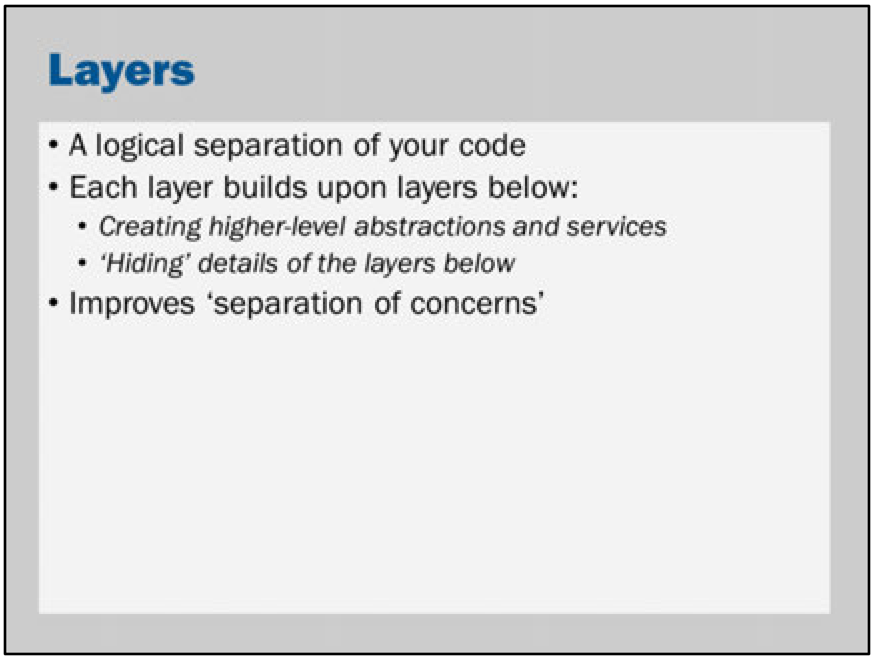
Architecture objectives

Development:
- Maintainability
- Testable
- Affordable
- Understandable

Performance:
- High throughput
- Low latency
- Fast response time
- Scalability
- Availability
- Reliability

Maintainability: How easy is it to make changes and fix bugs?
Testability: How easy is it to check for correctness and monitor and adapt the running system?
Affordability: How much does it cost to develop and to run?
Understandability: Can new programmers understand and improve the system?

Throughput: How many requests can be handled in a given period of time?
Latency: How long it takes to contact the server?
Response time: How long it takes to respond to an individual request?
Scalability: How does the system cope with more servers, more data and more users?
Availability: What proportion of the time is the system online?
Reliability: How often does it fail?

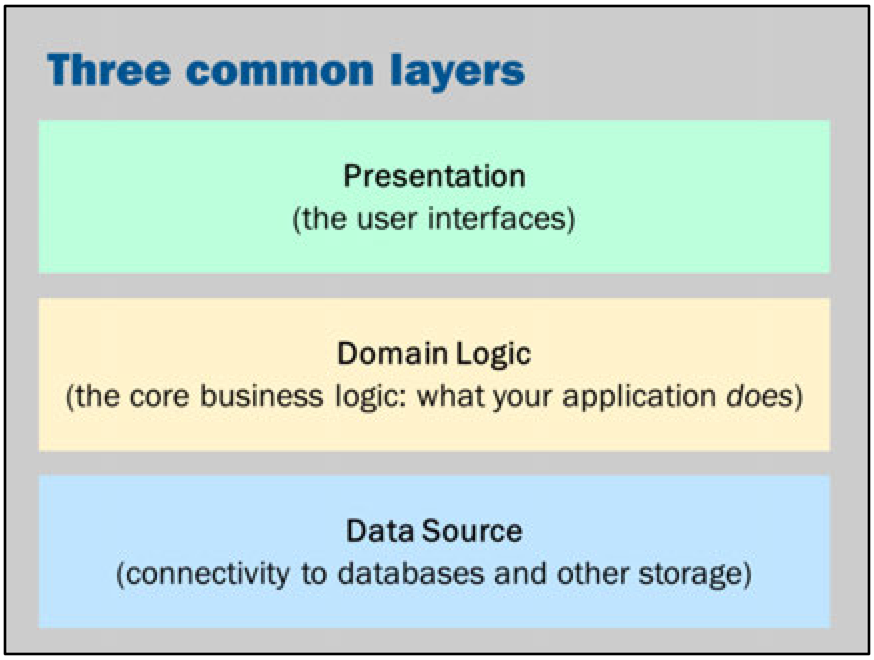


Layers

- A logical separation of your code
- Each layer builds upon layers below:
  - *Creating higher-level abstractions and services*
  - *'Hiding' details of the layers below*
- Improves 'separation of concerns'

We have been looking at layering throughout the entire subject.
We tried to separate domain logic from presentation when we were looking at JSP.
We tried to separate persistence logic from business logic when we were looking at JDBC and DAOs.

Layering is an outcome of good software design.
A layer is a collection of classes that work well together and that have minimal dependencies on other layers.
Typically, layers build upon one another.
Higher level layers depend on lower level layers.
Each layer creates higher-level abstractions and hides implementation details of the layer below.


Three common layers
Presentation
(the user interfaces)
Domain Logic
(the core business logic: what your application *does*)
Data Source
(connectivity to databases and other storage)

When developing software applications, we tend to think in terms of these three major layers:

Presentation: The code responsible for generating the visual user interface. In an expense management system, this would be the code that generates HTML, decides which fonts and layout positions to use and handles upload and download of receipts.

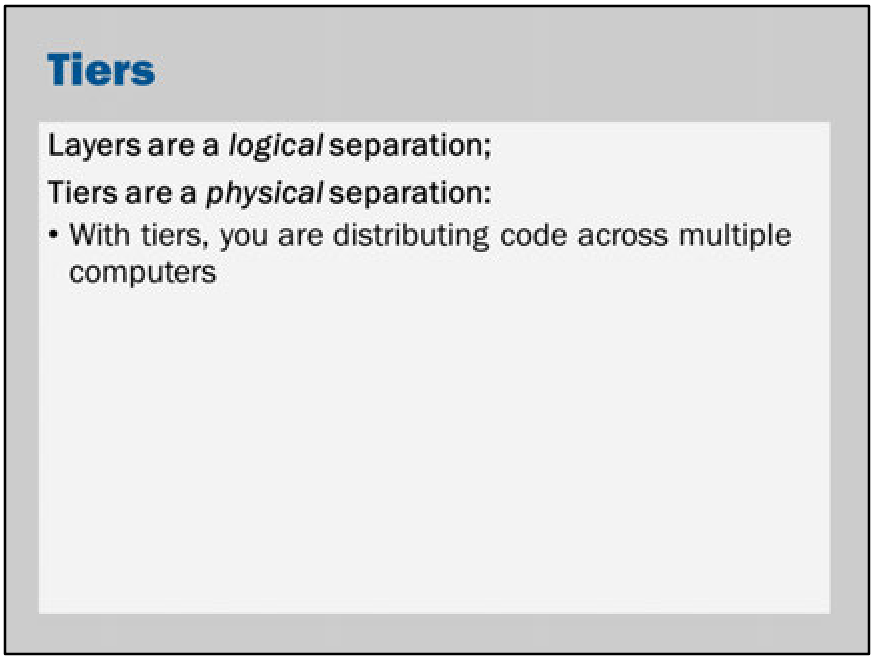
Domain logic: The code that implements the business logic of the system. In an expense management system, this would be code that determines an expense is valid, computes the expense totals, decides who to forward expenses to for approval and so on.

Data source: The code that stores the data. In an expense management system, this would be the database, the code that communicates with the database, and also the mechanism used for storing the uploaded files (perhaps uploaded files are stored in the database or in the filesystem).

Sometimes the layers are subdivided into a bit more detail and/or go by different

names. For example, the book "Core J2EE Patterns" uses these layers (they refer to it as tiers):
Client Layer: the application or browser that displays the GUI
Presentation Layer: handles session management, content creation and delivery
Business Layer: business logic, transactions and data services
Integration Layer: connections to legacy systems and services such as rule engines or workflow engines
Resource Layer: the underlying database

Even though there is no "official" layering or name for the layers, professional developers will understand the need for layering and the general principle that we should keep interchangeable parts of the system separate so that they can be interchanged.



Tiers

Layers are a *logical* separation;

Tiers are a *physical* separation:

- With tiers, you are distributing code across multiple computers

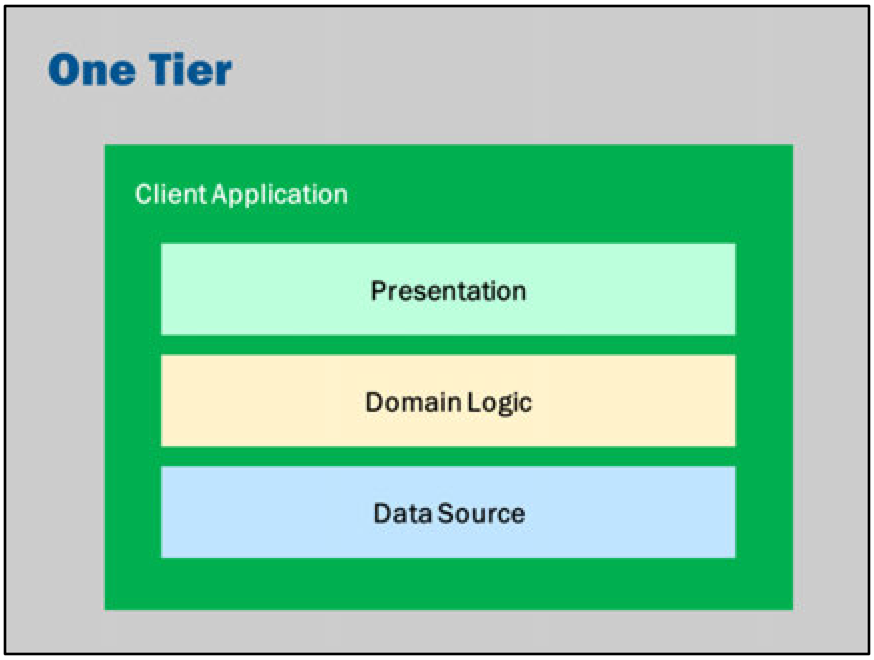
If we've got a well designed application split into layers, it may also be possible to distribute the application over multiple computers (e.g., run each layer on a different computer).
Doing so makes it possible to go from single-user to multi-user applications.
Doing so can also get us performance benefits. In particular, sharing workload over multiple computers can increase throughput and therefore the ability to serve more users.

Layers refers to the design of your code and classes.
You can have a multi-layer application on a single computer or you can have one layer of an application running on many computers.

When we physically separate layers on separate computers, we refer to the "physical layers" as tiers.
A tier is essentially a layer of physical computers.
Tiers are a physical structure.

One Tier
Client Application

A one tier or single tiered application is an application that runs entirely on one computer.



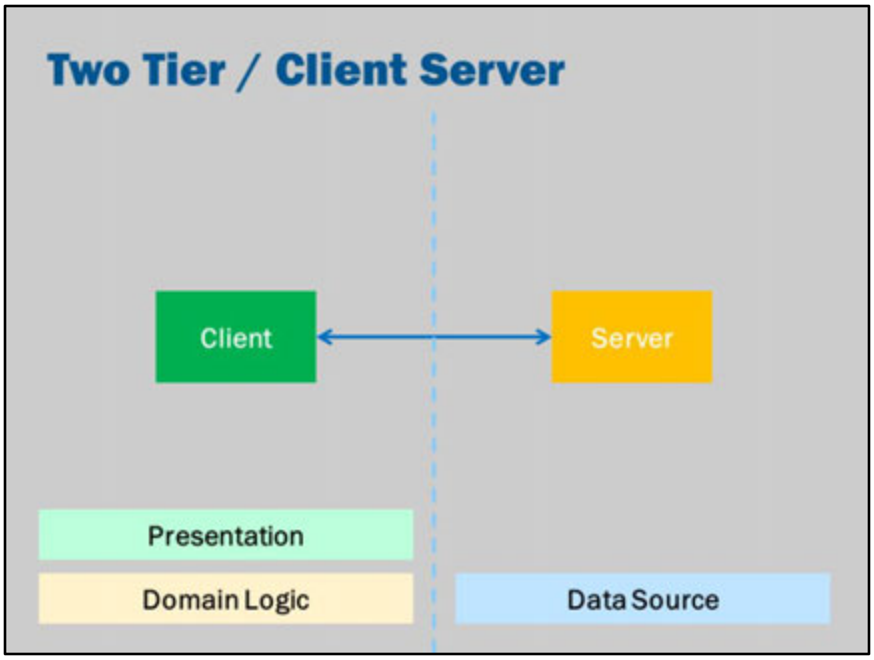
Within that one tier, we can have the entire application. All of the logical layers run on the single tier.

Example:
An example of this architecture would be a desktop application such as Word or PowerPoint.

Advantages:
- Simple
- Self-contained
- No need to run a server
- No need for network

Disadvantages:
- No resource sharing
- Monolithic applications
    - Needs powerful computer
    - Difficult to maintain
    - Difficult to integrate

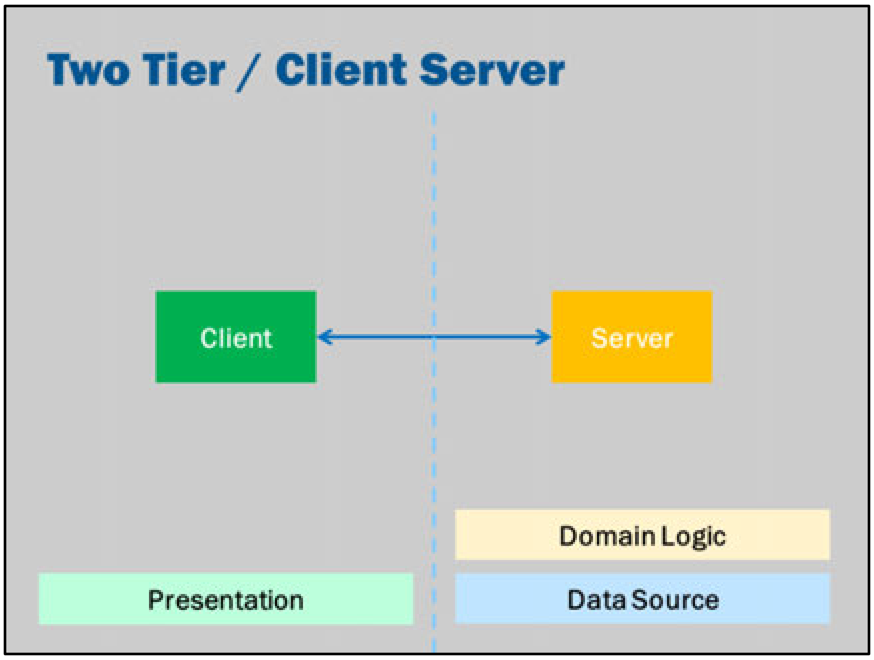
- Not scalable



Example:
- Custom application inside a company (e.g., order processing, inventory) that accesses the database directly

Advantages:
- Shared access to data, resources
- Typically robust, reliable (over private network)
- Less complex

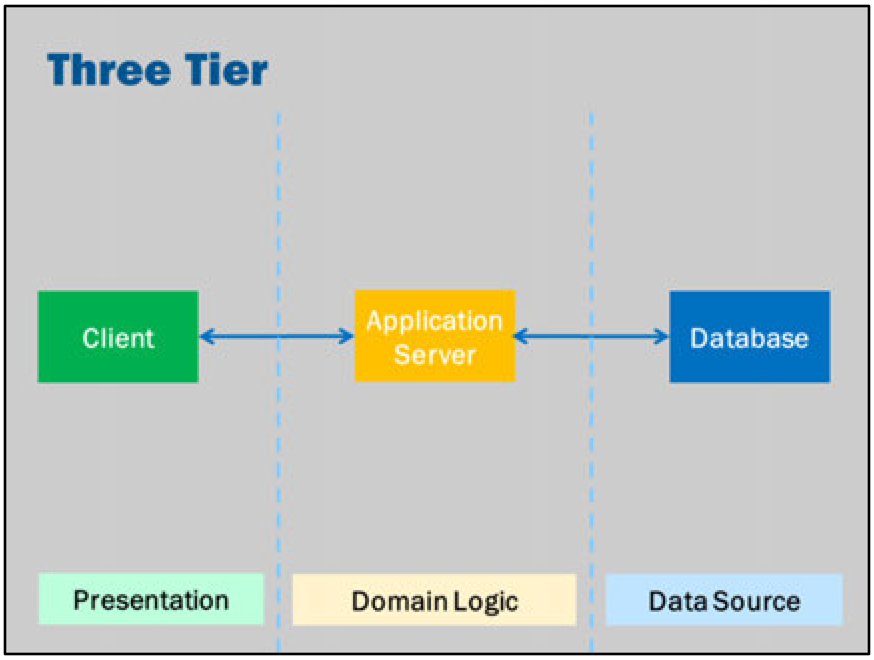
Disadvantages:
- Multiple requests for data can tax the network
- Difficult to maintain or extend functionality
- Exposure to security violations (e.g., clients sending their own SQL queries)
- Not scalable (especially to web audiences)
- Ties to one presentation type



The layers may be distributed over two tiers in a different way.

Example:

- Web browser (client) and simple web server (server)



The three tier architecture is the prototypical conceptual model when thinking about web development. Most database-driven web applications will start out in a three tier architecture. When it scales and grows, additional tiers may be added.

The three tier model is the prototypical model for distribution.
The three layer model is the prototypical model for designing code in layers.
Perhaps this is why many people tend to use the word layer and tier interchangeably.
In practice, when somebody talks about the presentation tier, they may be referring to the presentation layer or the physical computers that the presentation layer runs on.
The context will make this clear.

Examples:

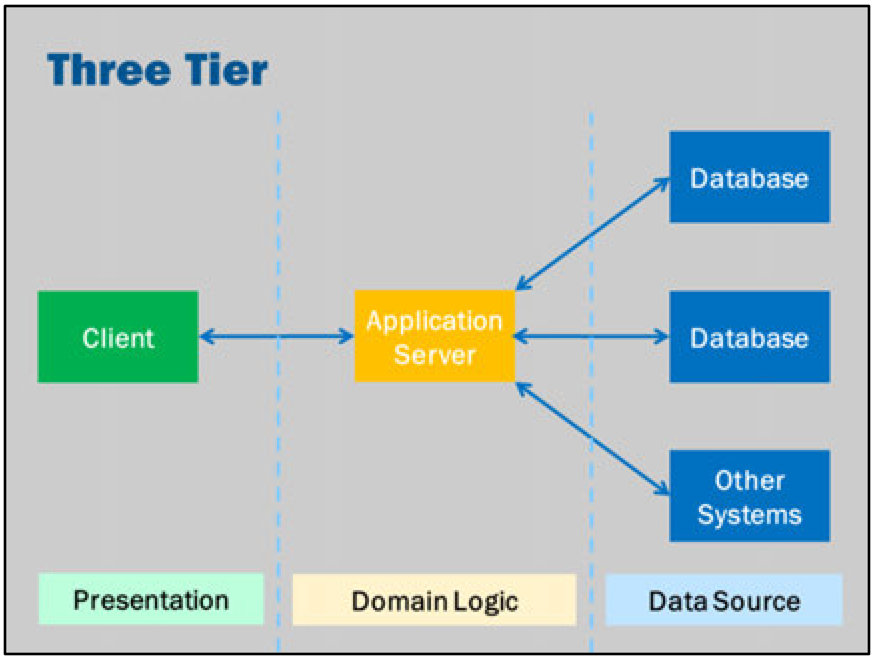
- Simple database-driven web applications
- Network-driven application (e.g., ERP system that has a desktop client, that connects to a remote web service, which in turn uses a database)

Advantages:

- Separation of business logic as distinct tier makes maintenance easier

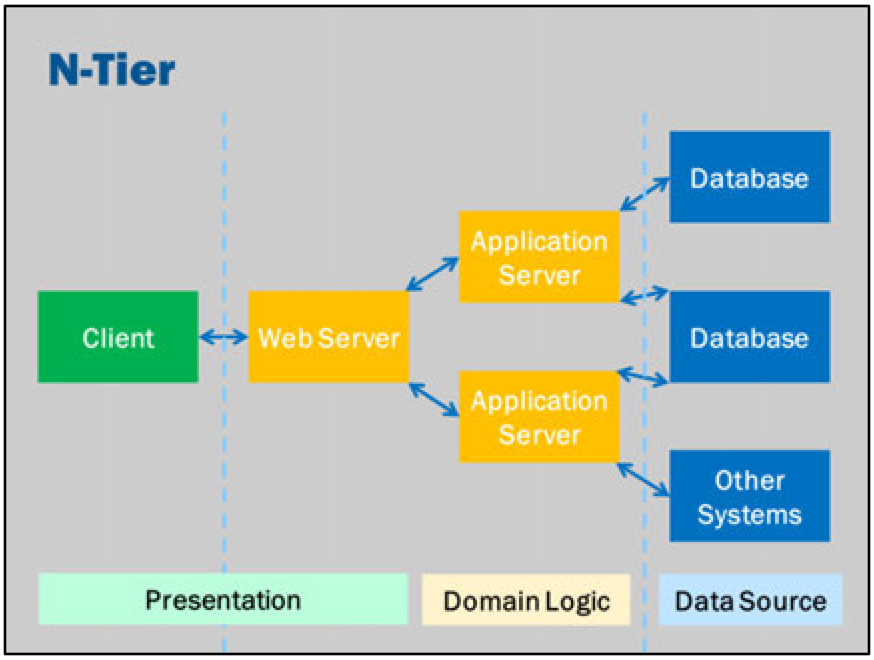
- Multiple user interfaces can be built and deployed
- Supports applications that use multiple data sources: enterprise database, XML documents, directory service
- Encourages applications to reuse data sources
- Separation into physical tiers helps encourage design that has good separation into layers

Disadvantages:
- Complexity
- Vendor incompatibility
- Single point of failure



We can have multiple computers or systems in a tier. For example, in a three tiered architecture, the application server might communicate to multiple databases.

This diagram highlights another benefit.
Consider a situation where we have 100 users and three separate databases.
In a two tier or client-server architecture we would need 100 x 3 = 300 separate connections (each user would need to connect to each of the three databases).
In a three tiered architecture, the application server can manage the connections to the database. In this way, we would only need 100 connections to the application server plus three connections from the application server to the database. In total this is only 103 connections!



Examples:

- Sophisticated component-based applications implemented using a web application server
- Java EE applications

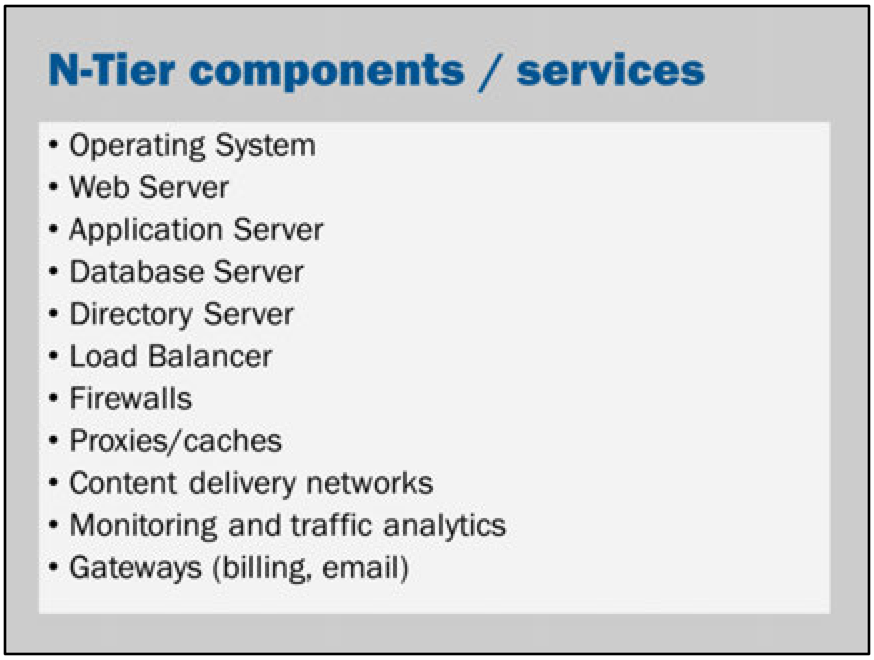
We will be building multi-tiered applications in the remainder of the subject

Advantages:

- Supports distributed applications
- Applications are built from reusable components
- Highly scalable

Disadvantages:

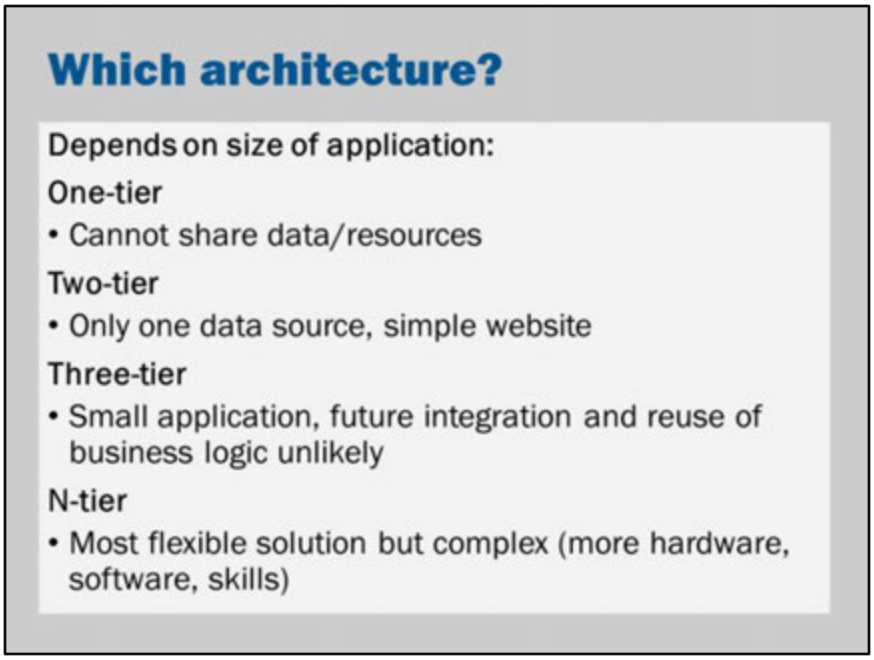
- Complex
- Costly
- Tiers can decrease response time (even though it may increase throughput and scalability)
- Load balancing issues – complex to manage



N-Tier components / services

- Operating System
- Web Server
- Application Server
- Database Server
- Directory Server
- Load Balancer
- Firewalls
- Proxies/caches
- Content delivery networks
- Monitoring and traffic analytics
- Gateways (billing, email)

In multi-tier environments, things can get very complex.
Load balancers, proxies, firewalls, caches, reverse proxies and content delivery networks can be inserted between layers.
They can improve performance, security and scalability.



Which architecture?

Depends on size of application:

One-tier

- Cannot share data/resources

Two-tier

- Only one data source, simple website

Three-tier

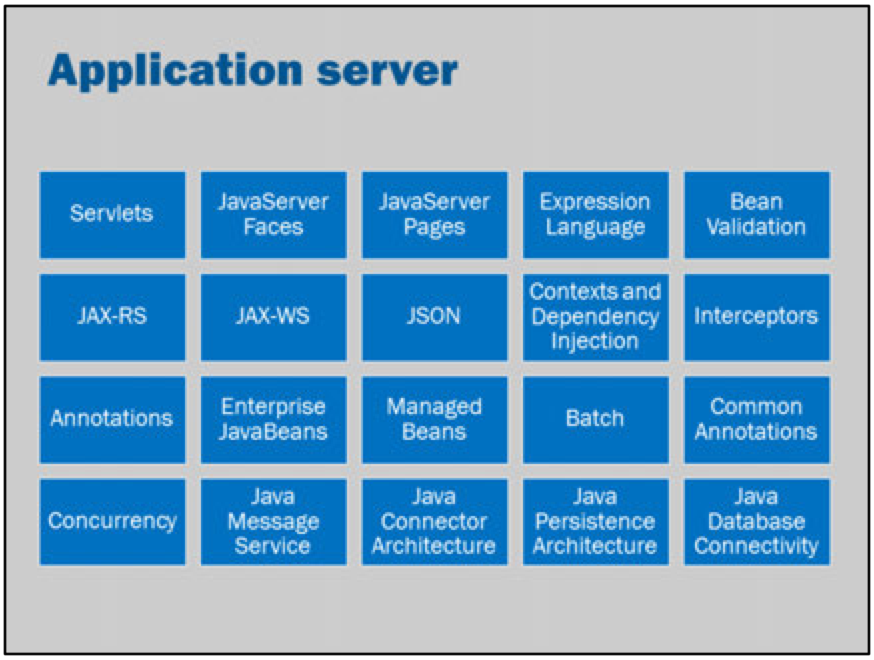
- Small application, future integration and reuse of business logic unlikely

N-tier

- Most flexible solution but complex (more hardware, software, skills)

There is no best answer to which architecture to use. It will be a trade off depending on the situation.
Consider the advantages and disadvantages of each architecture when trying to make a decision.

For most web applications, it probably makes the most sense to start out with a three-tier architecture.
By keeping n-tier application architectures in mind (but only building a 3-tier application), you can design your application with future growth in mind.



This diagram illustrates some of the services provided by a Java EE 7 application server.
We have seen it before.

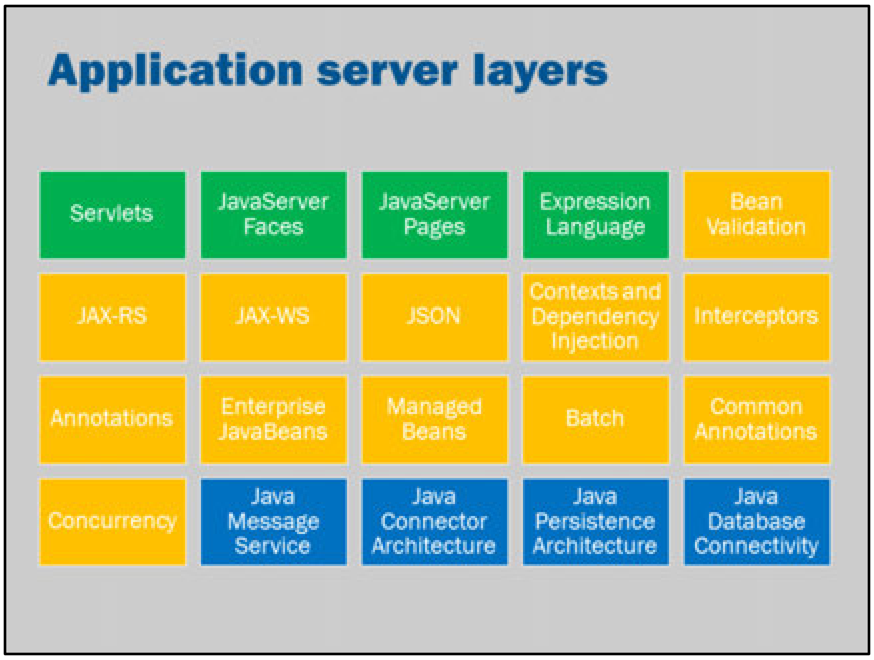
In fact, now we can recognize some of the services provided by the Java EE 7 application server.

They help us layer our software better and also distribute it into tiers.
Many of these services relate to the components and capabilities of n-tier architectures.

Can you recognize how these features relate to the three layers mentioned at the start of the lecture?

What is the difference between a web server and an application server?

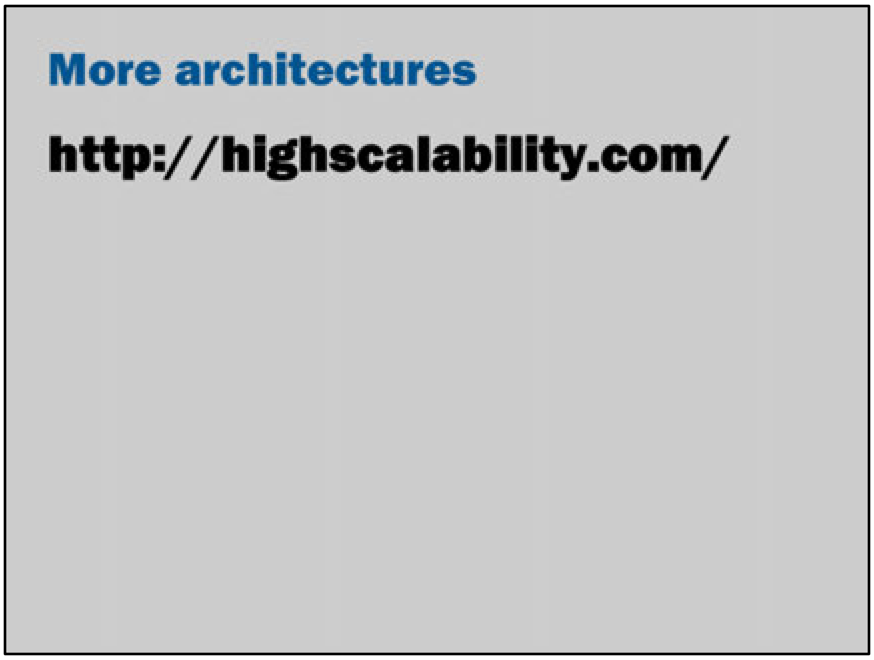
An application server is a “superset” of a web server. A web server provides features and services relating to serving dynamic content on the web. Application servers provide many more features including remote method invocation (i.e., non-web clients), transactions, database connectivity, messaging services and so on.



Presentation:
Clearly Servlets, JSF, JSP and EL are presentation-layer technologies.

Domain Logic:
We have seen some of these technologies already (e.g., JSF backing beans are a type of managed bean).
Will cover many of them later in the subject.
These services are designed with n-tier application architectures in mind.
For example, if you use Enterprise JavaBeans to implement your domain logic, then the Java EE container can help with distributing that domain logic across multiple computers.

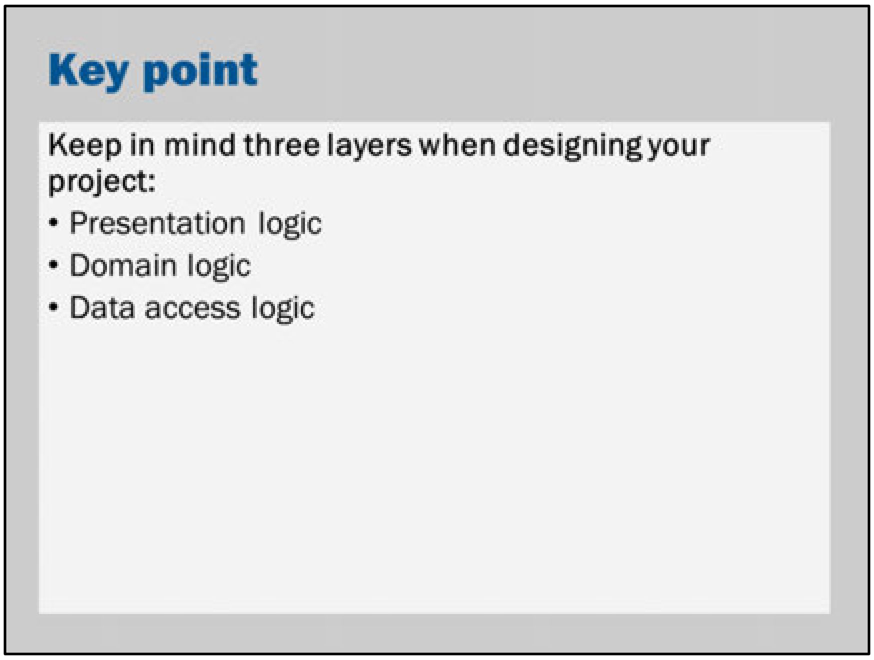
Data Source:
JDBC we have covered as a persistence technology.
JPA will be covered later as an alternative to JDBC.
JCA and JMS are technologies used to integrate with other systems.



More architectures

http://highscalability.com/

This is a fantastic resource for learning about how real world systems (Twitter, Youtube, Facebook, etc) are made to scale to millions of users.

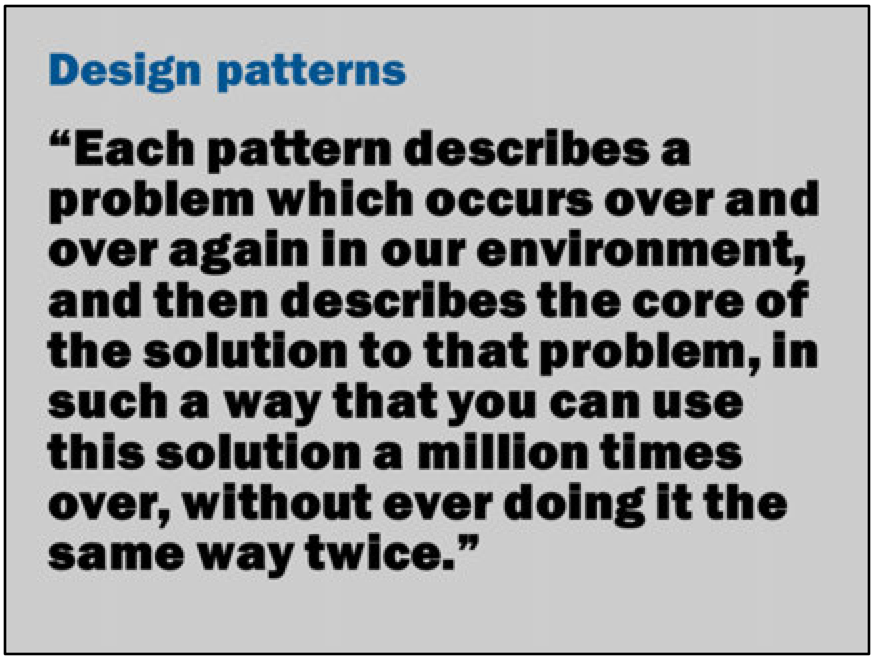


Key point

Keep in mind three layers when designing your project:

- Presentation logic
- Domain logic
- Data access logic

Design patterns



Design patterns

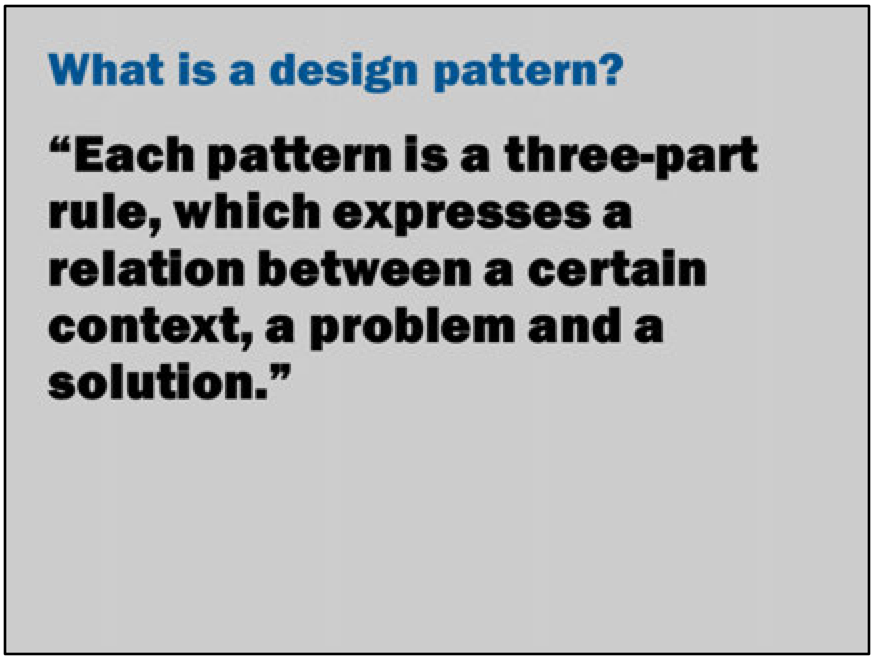
"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Alexander, C., Ishikawa, S. and Silverstein, M. (1977) A Pattern Language, Oxford University Press, page x.

Design patterns are inspired by the Architect Christopher Alexander who created a pattern language to describe common solutions to architectural problems.

For example, architects use "Pools of Light" as a solution to the problem of creating a sense of privacy and intimacy. Warm, low, lights should pool in places where people congregate and socialize.

It is important to remember that design patterns are just guidelines. They will be adapted to suit your particular situation. You do not need to follow them as though they are a strict rule. They are the collected wisdom of many people who have found a solution to a common problem. The wisdom can provide insight and help but it should not be blindly followed without understanding.

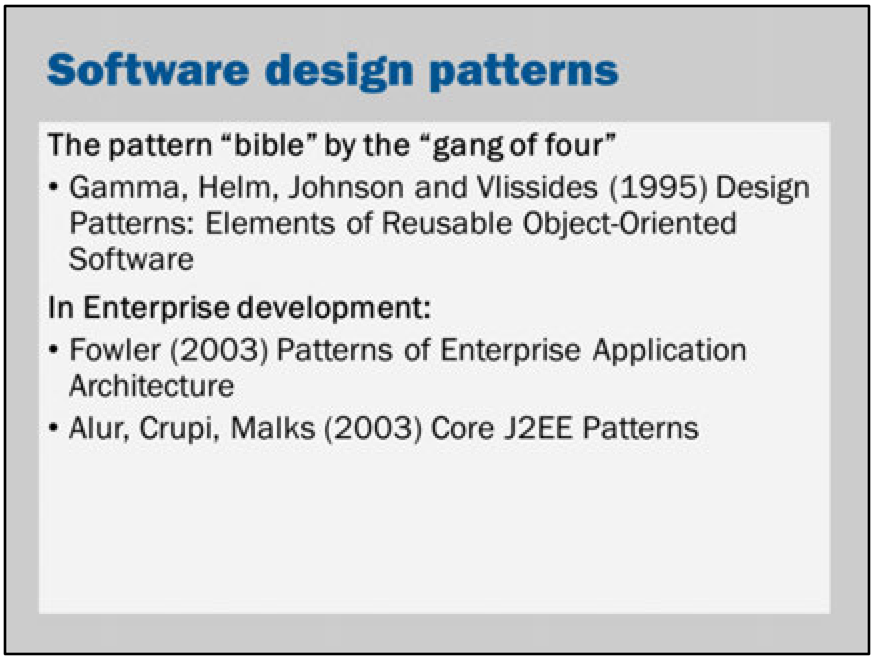


What is a design pattern?

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution."

Alexander, C., (1979) The Timeless Way of Building, Oxford University Press, page 247.

A design pattern captures a situation, a problem and a solution. More importantly, however, it also captures how these three elements relate to each other.


Software design patterns

The pattern "bible" by the "gang of four"

- Gamma, Helm, Johnson and Vlissides (1995) Design Patterns: Elements of Reusable Object-Oriented Software

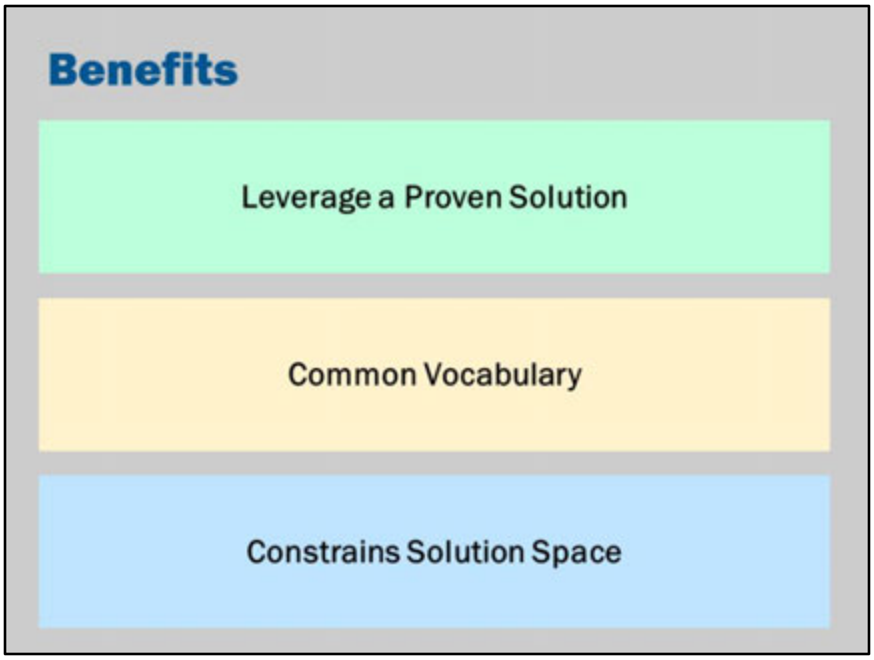
In Enterprise development:

- Fowler (2003) Patterns of Enterprise Application Architecture
- Alur, Crupi, Malks (2003) Core J2EE Patterns

Even though pattern languages started out in architecture, they've become extremely popular in software engineering.

The classic book, written by four authors is known as the "Gang of Four" book.

In Enterprise development there are two "classic" books. Core J2EE Patterns is primarily aimed at Java developers but both have introduced common and widely used patterns.



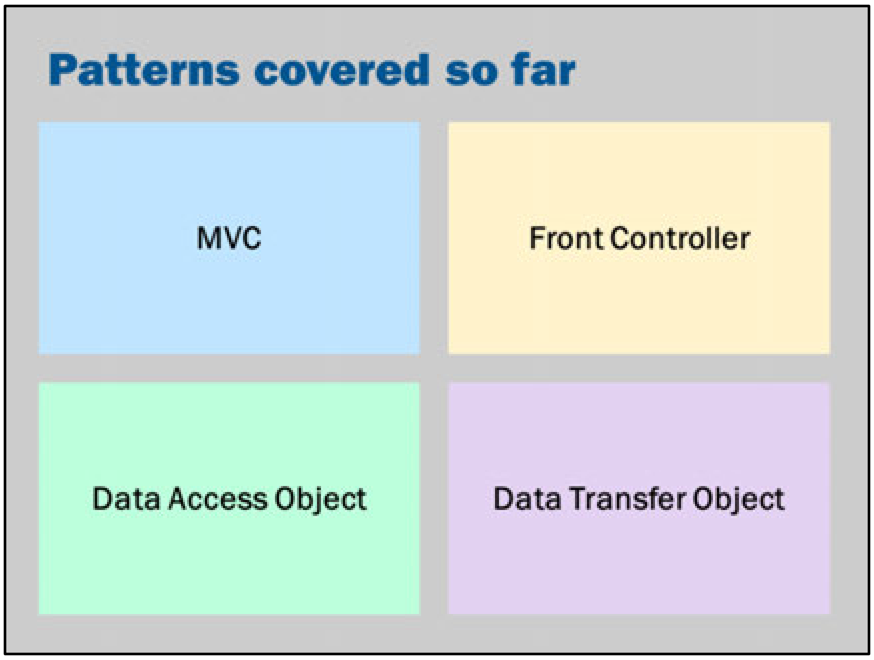
Benefits

Leverage a Proven Solution

Common Vocabulary

Constrains Solution Space

- Design patterns codify the wisdom of many experts. By using a design pattern, you know that you are using a solution that has worked many times before.
- Design patterns can be used to describe your system to other experts quickly and efficiently. For example, "In Assignment 1, you will build an **MVC**-based web app using JSF. The data access is encapsulated in **Data Access Objects**, with values stored in **Data Transfer Objects**." The basic structure and architecture of the code is clear from the description, even though I have not shown you a detailed class diagram.
- Design patterns help narrow down possible designs and architectures. This can help avoid bad ideas, focus on the essential business logic of the problem and speed up problem solving.


Patterns covered so far

MVC

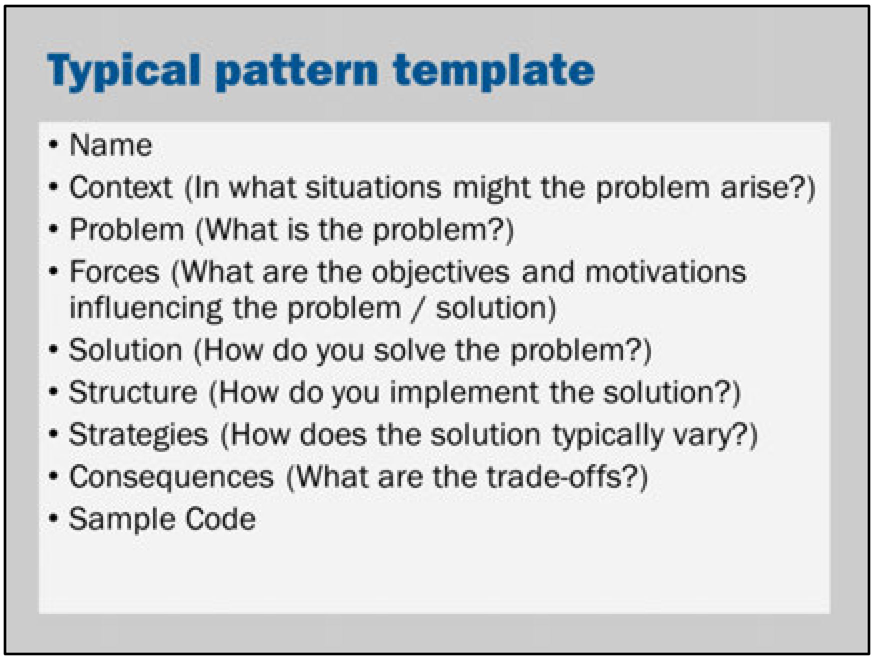
Front Controller

Data Access Object

Data Transfer Object

We have looked at many patterns already:

- MVC in Week 3
- Front Controller in Week 3 tutorials (This is where we have a Servlet that handles all requests and decides which action or view to show depending on the nature of the request. JavaServer Faces uses a front controller called FacesServlet to handle all incoming requests.)
- Data Access Object and Data Transfer Object in Week 5



Typical pattern template

- Name
- Context (In what situations might the problem arise?)
- Problem (What is the problem?)
- Forces (What are the objectives and motivations influencing the problem / solution)
- Solution (How do you solve the problem?)
- Structure (How do you implement the solution?)
- Strategies (How does the solution typically vary?)
- Consequences (What are the trade-offs?)
- Sample Code

In practice, design patterns are presented in a structured format.
The structured format highlights the key dimensions of the pattern (namely the context, problem, solution and the relationships between these three).

This is just one of many possible ways of structuring design patterns.

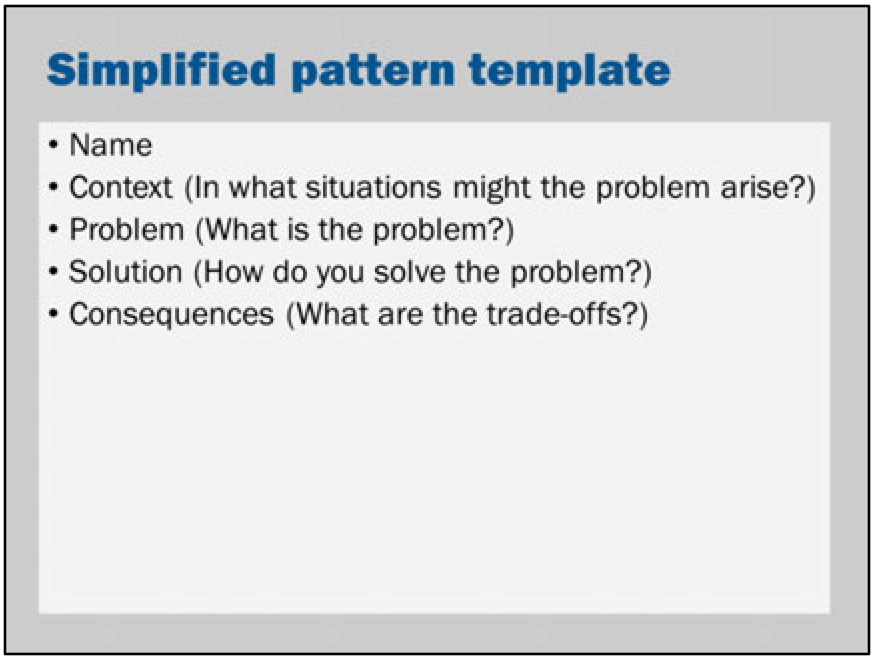
"... design patterns are structured according to a defined pattern template. The pattern template consists of sections as follows:

- **Context:** Sets the environment under which the pattern exists.
- **Problem:** Describes the design issues faced by the developer.
- **Forces:** Lists the reasons and motivations that affect the problem and the solution. The list of forces high-lights the reasons why one might choose to use the pattern and provides a justification for using the pattern.
- **Solution:** Describes the solution approach briefly and the solution elements in detail. The solution section contains two subsections:
- **Structure:** Uses diagrams to show the basic structure of the solution. The diagrams present the dynamic mechanisms of the solution. There is a detailed explanation of the participants and collaborations.
- **Strategies:** Describes different ways a pattern may be implemented. Where a

strategy can be demonstrated using code, code may be provided in this section or in a separate sample code section.

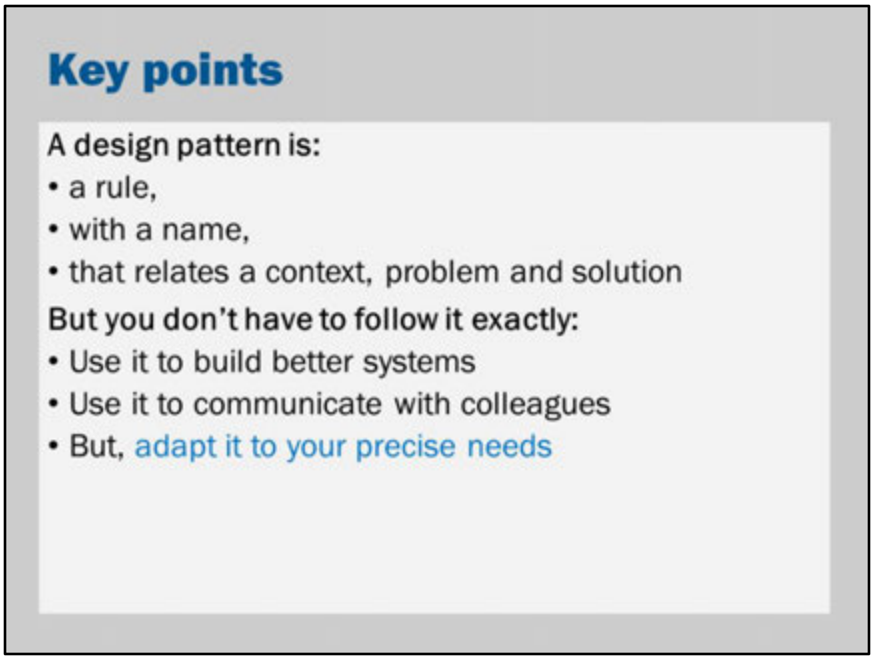
- **Consequences:** Here we describe the pattern trade-offs. (Generally, this section focuses on the results of using a particular pattern or its strategy, and notes the pros and cons that may result from the application of the pattern.)
- **Sample Code:** this section includes example implementations and code listings for the patterns and the strategies.
- **Related Patterns:** This section lists other relevant patterns from other sources. "

-- Method and apparatus for developing enterprise applications using design patterns (patent number: US 20020073396 A1) by John Crupi, Deepak Alur and Daniel Malks


Simplified pattern template

- Name
- Context (In what situations might the problem arise?)
- Problem (What is the problem?)
- Solution (How do you solve the problem?)
- Consequences (What are the trade-offs?)

There's a lot in the pattern template of the previous slide.
Here's a simplified template (that I may use during in-class exercises).



Key points

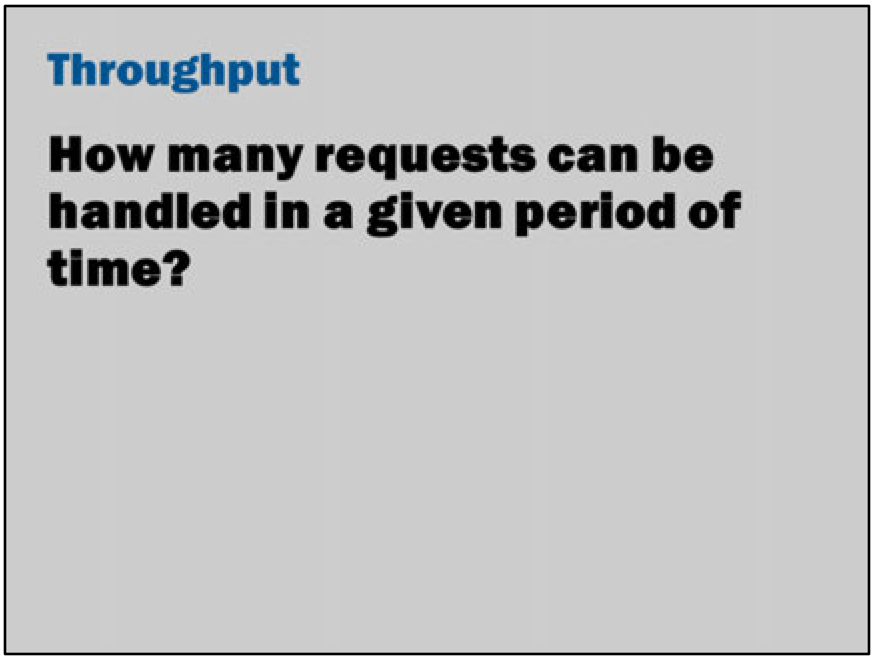
A design pattern is:

- a rule,
- with a name,
- that relates a context, problem and solution

But you don't have to follow it exactly:

- Use it to build better systems
- Use it to communicate with colleagues
- But, adapt it to your precise needs

Performance



Throughput

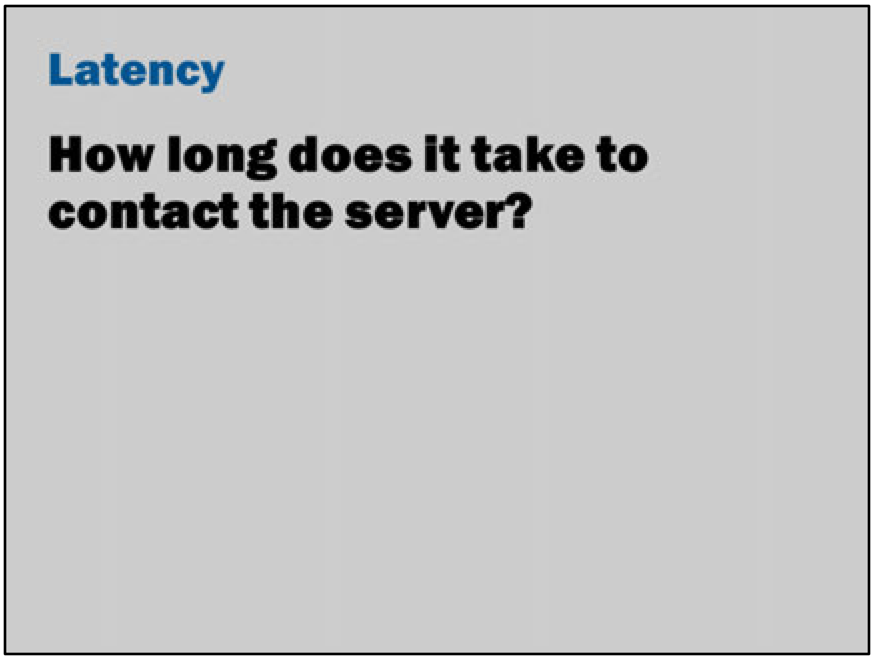
How many requests can be handled in a given period of time?

Throughput: How many requests can be handled in a given period of time.
Latency: How long it takes to contact the server.
Response time: How long it takes to respond to an individual request.

Illustrative example:

Consider an example, of calling your bank.
It is:
- High throughput (thousands of people call the bank every day)
- Low latency (you may have to wait a long time on hold to get connected to a person)
- High response time (you get your answer quickly and don't spend a long time waiting for the answer)

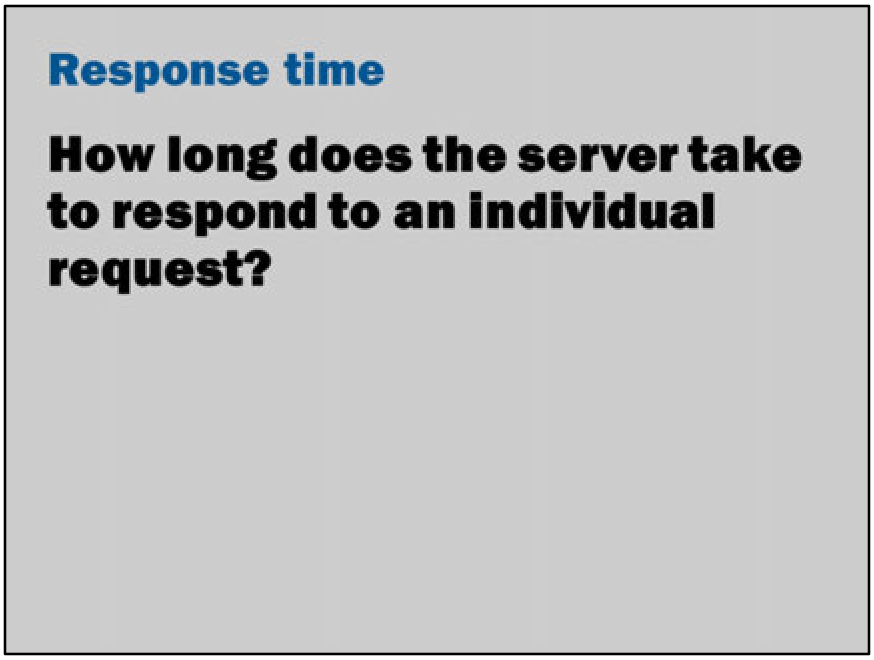
On the other hand, calling your mother is:
- Low throughput (only a small number of people can call her in a day)
- Low latency (contacting her is usually very fast – you aren't put on hold)
- High response time (you have a long conversation)



Latency

How long does it take to contact the server?

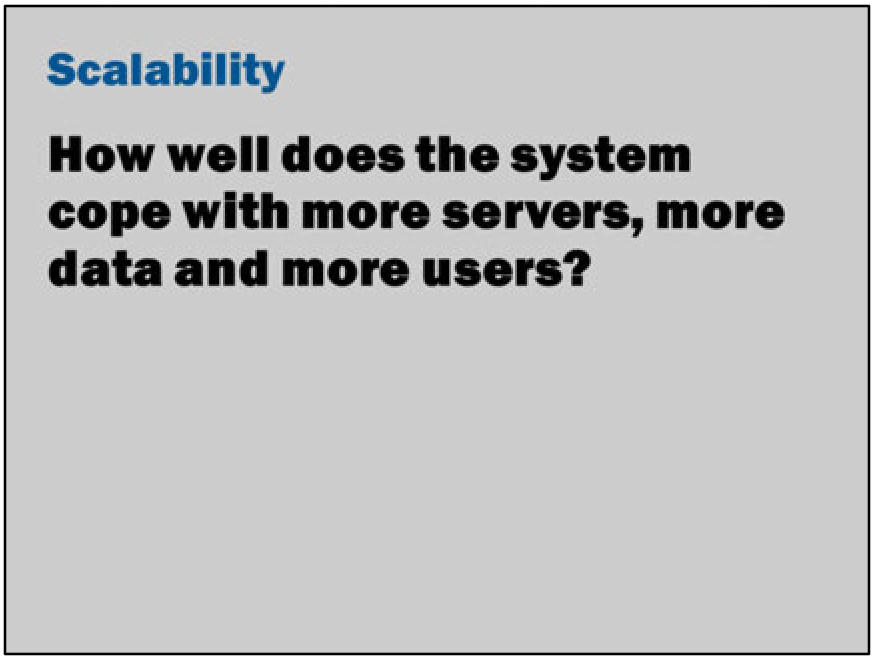
Refer to the notes on the throughput slide.



Response time

How long does the server take to respond to an individual request?

Refer to the notes on the throughput slide.



Scalability

How well does the system cope with more servers, more data and more users?

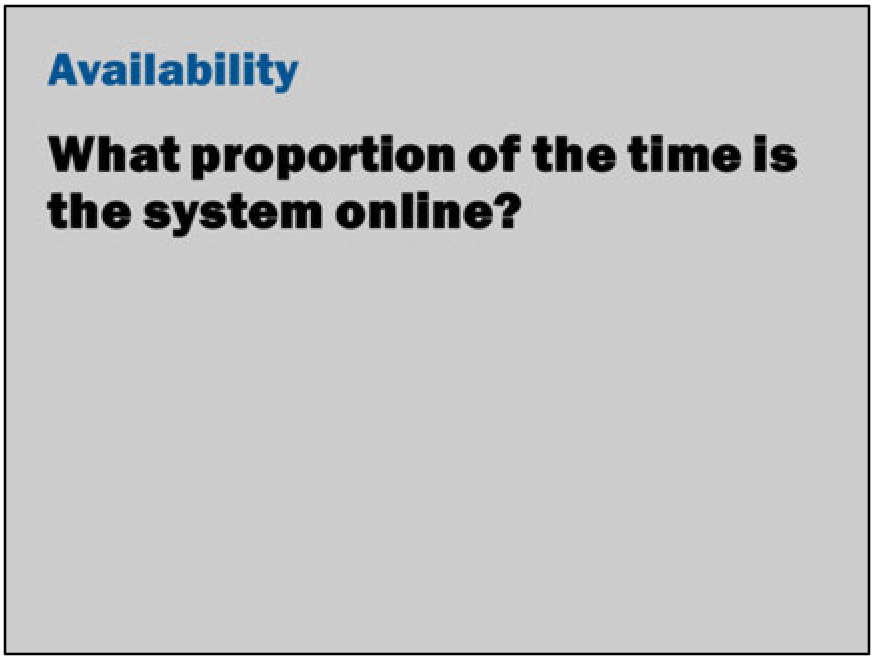
Scalability: How does the system cope with more servers, more data and more users.

Illustrative example:

Storing data in an Excel file on a drive is perfectly fine for one user. However, when 100 users attempt to simultaneously modify a spreadsheet with millions of rows on a shared drive, things will not work well. This is not scalable. Any more than one computer and you experience problems.

Illustrative example:

The bank can always employ more staff (high scalability).
But there's only one of your mother (not scalable).



Availability

What proportion of the time is the system online?

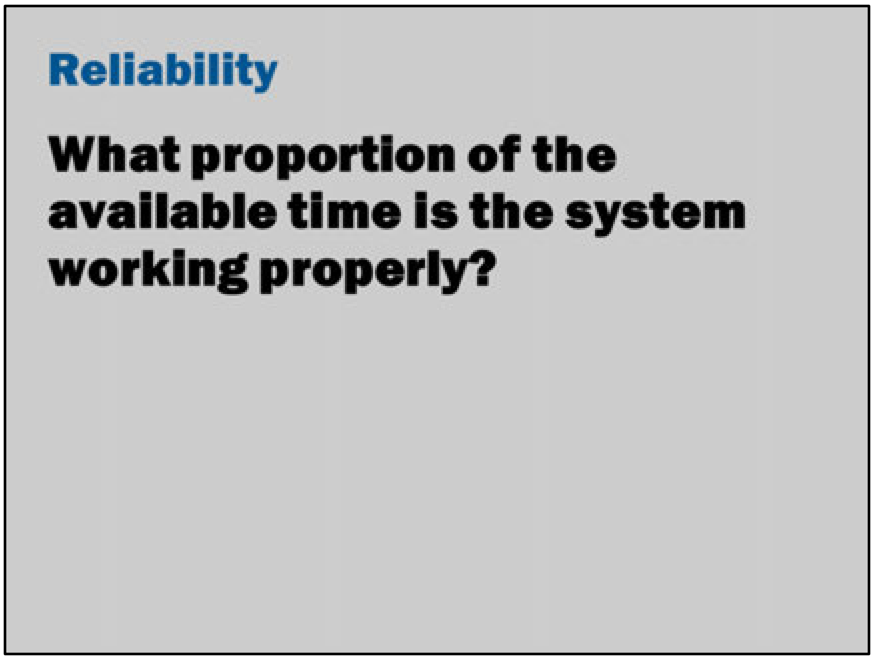
Availability: What proportion of the time is the system online.
Reliability: How often does it fail.

Availability and reliability are related.

Illustrative example:

Telephone banking for my bank has low availability (it is only open from 7am to 9pm).
However, it has very high reliability (it is almost never closed during these hours).

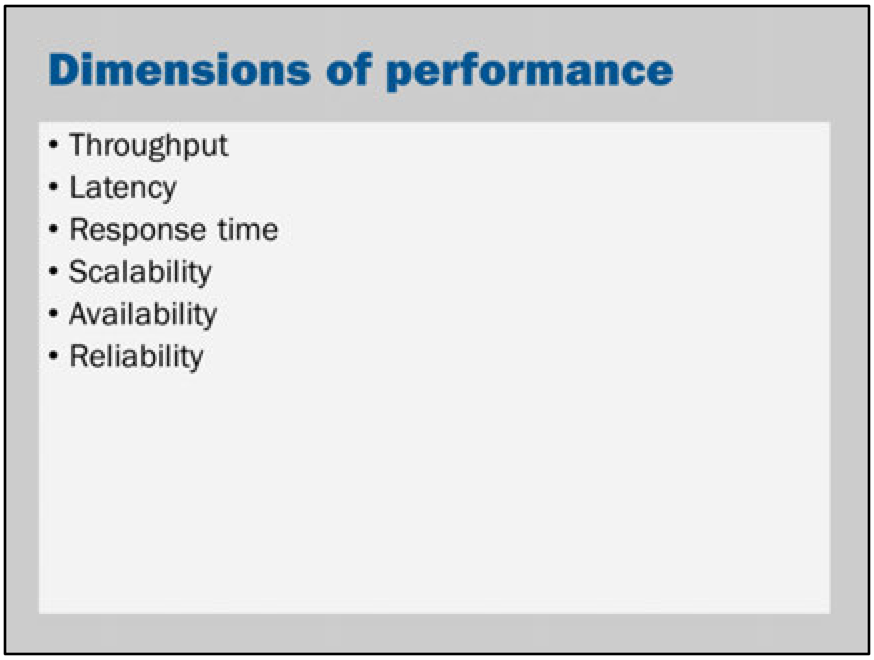
Calling my mother has high availability (I can probably call her 24 hours a day, 7 days a week).
However, it has low reliability (she sometimes can't get to the phone).



Reliability

What proportion of the available time is the system working properly?

Refer to the notes on the availability slide.

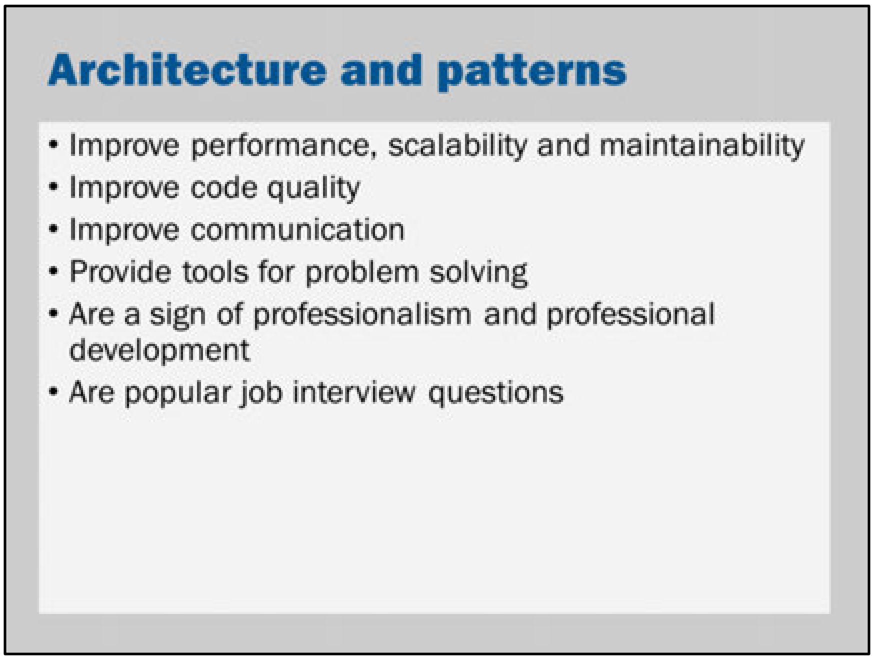


Dimensions of performance

- Throughput
- Latency
- Response time
- Scalability
- Availability
- Reliability

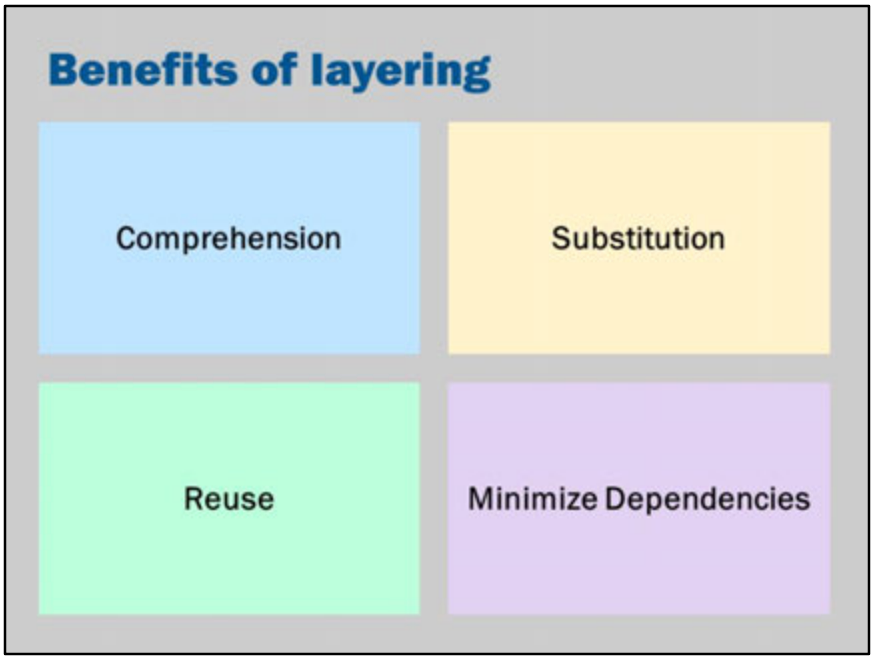
Throughput: How many requests can be handled in a given period of time?
Latency: How long it takes to contact the server?
Response time: How long it takes to respond to an individual request?
Scalability: How does the system cope with more servers, more data and more users?
Availability: What proportion of the time is the system online?
Reliability: How often does it fail?

Bonus slides


Architecture and patterns

- Improve performance, scalability and maintainability
- Improve code quality
- Improve communication
- Provide tools for problem solving
- Are a sign of professionalism and professional development
- Are popular job interview questions

Architectures and patterns are tools to improve the quality of your code and your designs.



"Breaking down a system into layers has a number of important benefits.

- You can understand a single layer as a coherent whole without knowing much about the other layers. You can understand how to build an FTP service on top of TCP without knowing the details of how ethernet works.
- You can substitute layers with alternative implementations of the same basic services. An FTP service can run without change over ethernet, PPP, or whatever a cable company uses.
- You minimize dependencies between layers. If the cable company changes its physical transmission system, providing they make IP work, we don't have to alter our FTP service.
- Layers make good places for standardization. TCP and IP are standards because they define how their layers should operate.
- Once you have a layer built, you can use it for many higher-level services. Thus, TCP/IP is used by FTP, telnet, SSH, and HTTP. Otherwise, all of these higher-level protocols would have to write their own lower-level protocols.

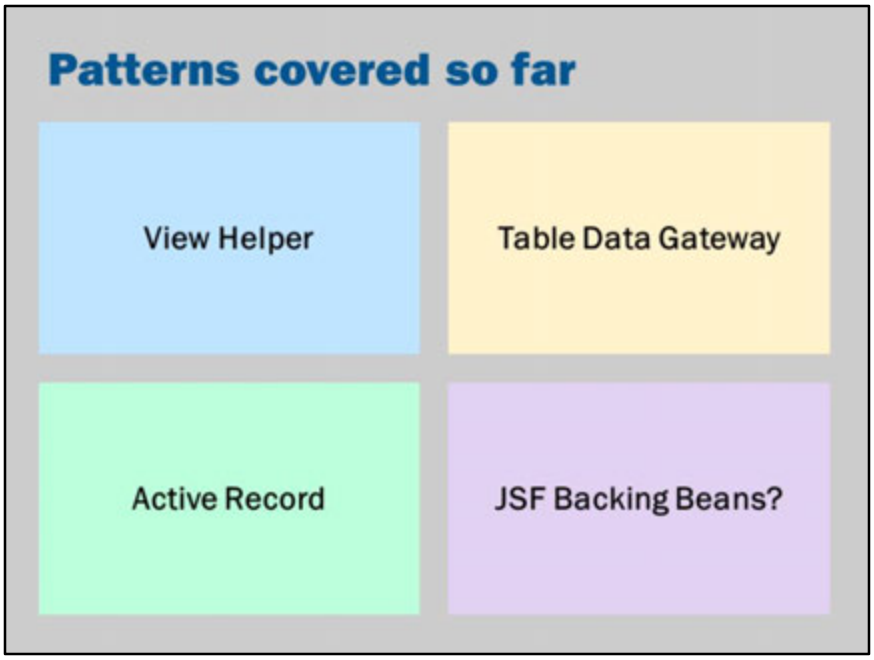
Layering is an important technique, but there are downsides.

- Layers encapsulate some, but not all, things well. As a result you sometimes get cascading changes. The classic example of this in a layered enterprise application is adding a field that needs to display on the UI, must be in the database, and thus

must be added to every layer in between.
- Extra layers can harm performance. At every layer things typically need to be transformed from one representation to another. However, the encapsulation of an underlying function often gives you efficiency gains that more than compensate. A layer that controls transactions can be optimized and will then make everything faster.

But the hardest part of a layered architecture is deciding what layers to have and what the responsibility of each layer should be."

---Fowler, M. (2003) Patterns of Enterprise Application Architecture, Addison Wesley, p. 17.



Patterns covered so far

View Helper

Table Data Gateway

Active Record

JSF Backing Beans?

These are some more patterns encountered in the subject:

Also:

- View Helper in Week 3 (We used beans in a JSP Model 1 architecture as "helpers". At the time, we did not refer to it as a design pattern.)
- Table Data Gateway (briefly at the end of Week 5)
- Active Record (briefly at the end of Week 5)

Perhaps our structure of exposing the model via a field in a JavaServer Faces backing bean is also a design pattern.
I have not seen it formalized but it seems to be a common pattern.

There is quite a lot that we have covered in this subject already. Congratulations!

Factory

Factory design pattern pattern

- **Context:** In multi-layer applications you often want to allow layers to be modular and/or pluggable. For example, a persistence layer may need to be changed depending on the environment (e.g., development vs test vs production), the database (e.g., Derby vs Oracle vs Microsoft) or the underlying storage technology (e.g., database vs filesystem vs remote server).
- **Problem:** You want to choose the particular implementation of an interface at run-time.
- **Forces:** You want to allow multiple implementations of a single interface. You want to allow the choice to be made in configuration files or by scanning the execution environment.
- **Solution:** Use a Factory to encapsulate the object creation process. At runtime, the factory determines an appropriate implementation to use.
- **Structure:** See http://www.oodesign.com/factory-pattern.html for diagrams
- **Strategies:** See http://www.oodesign.com/factory-pattern.html for various implementation strategies: class registration, reflection, hard-coded switch statements, configuration files.
- **Consequences:** Adds a layer for object creation, adding overhead and complexity.

Reduces dependencies on specific implementations. Makes the returned object instance less predictable. Enforces programming to an interface (rather than to a concrete implementation).

- **Sample Code:** See http://www.oodesign.com/factory-pattern.html for sample code
- **Related Patterns:** Abstract Factory Pattern, Factory Method Pattern, Dependency Injection.