COSC 340: Software Engineering

Design and Architecture

Michael Jantz
(adapted from slides by Ravi Sethi, University of Arizona)
The remaining lectures in this course will discuss concepts in software architecture and design.

In our discussion of software architecture, we’ll draw on lessons from the age old traditions of building architecture.

For example, we’ll look at classical principles of architecture – and discuss attributes and properties of “good” architectures.

We’ll also discuss the concept of structure as it relates to software design. Specifically, structure refers to the different components of a system and how they relate to each other.

An architectural view focuses on some aspect of a given architecture. There might be multiple views of a software architecture, depending on its intended purpose and audience.

We’ll also look at design patterns in software architecture. Just as with building architecture, we can use patterns to solve recurring problems in software.
The analogy between buildings and software is not perfect. During our discussion, you should keep in mind how software is different than traditional buildings.

For instance, software has both static (source code) and dynamic (run-time) structure; while buildings are (for the most part) static.

Software is intangible. Buildings are tangible – we have to look at them and live inside them – which has an effect on which parts of the design are most important.

Software is seemingly easy to change, and refactored. Buildings are seemingly hard to change, although buildings do get remodeled from time to time.
People have been studying architecture for literally thousands of years.

In the 1st century BC, a Roman architect named Vitruvius wrote a wide-ranging treatise on architecture.

The treatise was organized as 10 books and covered a wide range of topics including:
(read 'em off)

- Town planning, architecture, civil engineering
- Building materials
- Temples
- Civil buildings
- Domestic buildings
- Pavements and decorative plasterwork
- Water supplies and aqueducts
- Sciences: geometry, measurement, astronomy
- Machines: water mills, drainage, hoisting, pneumatics

It's important because it's the only contemporary source on classical architecture to have survived and was very influential during the renaissance.

It is also the source of the story of how Archimedes proclaimed 'eureka!' when he entered his bath tub because he realized that the volume of water displaced must be equal to the volume of the part of his body he had submerged.
In the treatise, Vitruvius laid out three principles that are still applied in architecture today.

These are:

**Utility**: Does the building conveniently serve its intended purpose?

**Strength**: Will the building stand? Are the foundations solid and have the materials been wisely selected?

**Beauty**: Is the appearance of the building pleasing and in good taste? Are the elements of the building in due proportion?
Principles from Classical Architecture
Software Equivalents for Vitruvius' Principles

• **Utility**
  – Does the system meet its requirements?

• **Strength**
  – Is the system robust? Will it scale and perform? Is the technology appropriate?

• **Beauty**
  – Is the implementation of the system elegant? Is it easy to understand and modify?

Utility: Does the system meet its requirements?

Strength: Is the system robust? Will it scale and perform? Is the technology appropriate?

Beauty: Is the implementation of the system elegant? Is it easy to understand and modify?
Let's examine these principles by looking at one of the most famous structures ever built. This picture shows the Hagia Sophia.

The Hagia Sophia was built in the mid 6th century in Constantinople or present-day Istanbul. At the time it was built it was the largest cathedral in the world, and remained the largest cathedral for the next 1000 years. It has a long and storied history.

It's currently a museum.

In 1453, it was converted to a mosque after Constantinople was conquered by the Ottomans, and in 1935, it was converted to a museum.
By any standards, the Hagia Sophia is an exceptional building.

Utility. It was obviously functional as a cathedral, but it also fulfilled Emperor Justinian I’s wish for a majestic church, grander and more imposing than all its predecessors. For centuries, it reigned as the greatest cathedral ever built. As a mosque, it showed the way for many others to come.

Strength. The Hagia Sophia stands tall, almost 1500 years after it was built. The main dome was re-architected, but that was in 562 AD.

Beauty. The beauty of the Hagia Sophia is intrinsic to its architecture and proportions. The main dome soars 182 feet from the floor. The vast open well-lit interior was richly decorated with mosaics and marble pillars. As a museum, it attracts millions of visitors every year.
Now, each of these principles, utility, strength, and beauty, are different perspectives on the same architecture.

The concept of an architectural view just refers to the fact that we can focus on some aspect of a given architecture. The view might be tailored to an intended purpose or some specific audience.

Typically a view focuses on a problem and outlines how the architecture solves that problem. Other aspects of the system that are not relevant to the solution are abstracted away.

We use different views to capture or focus on different aspects of an architecture. So, we often have to examine multiple views to understand the entire architecture.
For example, this figure is a view of the cross-section of the Hagia Sophia. It allows you to see the proportions and spaces in different parts of the structure.

Here you can easily see there’s a large round main dome, but also smaller half domes under and to the sides of the main dome.
Here is another example of a structural view of the domes Hagia Sophia.

This view allows you to see an innovative part of the structure's dome architecture called a pendentive. A pendentive distributes the load from the round dome to a square base, allowing you to place a dome over square rooms.
Hagia Sophia Dome Re-architected

- The original dome collapsed during the earthquake of 558
- The rebuilt dome is 30 feet higher to better distribute its weight to the supporting walls
- The re-architected dome is still standing, ~1500 years later
Another concept from building architecture that has been used extensively in software architecture is the concept of patterns.

The first person to really study architectural patterns in depth was Christopher Alexander, (who was himself an architect and urban planner.) He used the term pattern for a problem that occurs over and over again, together with "the core of the solution to that problem."

Alexander’s work on patterns has inspired many different applications in software, including object-oriented design patterns and software architecture patterns. There are entire conferences on Pattern Languages of Programming, and Extreme Programming was influenced by Alexander’s work, especially the belief that the occupiers of a building should design it.
Alexander's Patterns

• **Context**
  – Each pattern has both a larger and a smaller context
  – e.g., larger: roof completes a room, a room is part of a building, ...

• **Problem**
  – Some fundamental aspect of a design
  – e.g., the design of roofs for a cluster of buildings

• **"Core" of a Solution**
  – Guidance for designing a specific structure
  – e.g., high ceilings for public rooms lower for smaller gatherings very low in
    rooms or alcoves for one or two people

The description of each of Alexander’s patterns has three main parts: context, problem, solution.

Context. Each pattern fits within a context, both larger and smaller. For the larger context, consider that a roof completes a room, a room fits within a building, a building is part of a community. For the smaller context, consider the example of a door hinge. You can break down the structure of individual components to the very small scale.

Problem. The problem addressed by a pattern is some fundamental aspect of a design, such as the design of roofs for a cluster of buildings or the design of the ceiling height of a room. For instance, many of the offices here in Min Kao have similar size and shape. The Hagia Sophia consists of a cluster of buildings, with a high central building surrounded by lower wings with smaller rooms;

Solution. The “core” of a solution consists of guidance for designing a specific structure. For example, the pattern for ceiling heights includes the following guidance: “make ceilings high in rooms which are public or meant for large gatherings, lower in rooms for smaller gatherings, and very low in rooms or alcoves
for one or two people.”
Before moving on, I want to discuss the differences between the terms architecture, design, and implementation.

This quote from the Software Engineering Institute is helpful here (read it)

So, the terms software architecture, design, and implementation are often used imprecisely.

Usually, when we say implementation, we mean the actual details about how the code is written, or sometimes, the actual code itself. Design seems to mean not quite as detailed implementation, but enough details where you could follow the details and come up with something reasonably close. And architecture is a higher form of abstraction with fewer details.

But there are differences other than just the amount of detail because the architecture and design documents contain details that are not explicit in the implementation.
Both design and architecture deal with the structure of components in the system.

However, architecture deals specifically with relationships among components and their externally visible properties.

On the other hand, design includes both the internal and external structure of system components.

In this sense, architecture is part of the design of the system and is a subset of design.
SW architecture is fundamental to the practice of SW engineering.

You’ll recall our chart on the different aspects of SW engineering.

For instance, in the early stages of a project, before a design is finalized, an architecture can be used to get early feedback on design decisions and solution approaches. Is the solution approach on track to meet the major needs and goals? Have all stakeholders been considered?

An architecture can also be used to explore potential changes in requirements.

Project managers use an architecture to assemble a team and to assign work to team members. The architecture can be used to identify the skills and the resources needed for a project. When new members join a team, a description of the architecture can be used to train them on the goals and design decisions for the project.
In terms of technology, an architecture provides a foundation for the evolution of the system, and it can provide direction during iterative development.

Finally, in terms of constraints, project managers use an architecture to estimate the cost and schedule for a project. The estimates are based on the elements to be built and their relationships.
As an example of how software architecture and team organization are related, consider this observation by Melvin Conway – which has come to be known as Conway’s law.

Conway says that (read it)

The premise for this sociological observation is that two software modules A and B cannot interface correctly with each other unless the designer of A communicates with the designer of B. Thus, the interface structure of the system will necessarily reflect the social structure of the organization that produced it.

This has implications for how software is built and how ideas are adopted. When deciding whether or not to build some new functionality or incorporate some new idea into the system – we should ask – does this idea have some technical merit? Or are we considering this idea because of the sociological context in which we work?
As another example of the relationship between architecture and SW engineering – let us consider an example where Apple used its software architecture to make decisions involving security risks.

Every App on Apple iOS devices uses features provided by the underlying iOS system through Application Programming Interfaces (APIs).

Some of the iOS APIs are for Apple’s own use and are referred to as private APIs.

Qihoo began distributing apps that attempted to use these private API’s – which can pose a security risk. Because of the structure of their system, Apple was able to detect this, and banned all apps from Qihoo in the app store.

In 2012, Apple banned all apps from Qihoo as being potential security risks. The decision was based on the way the Chinese company used iOS APIs.

Misuse of private APIs can pose security risks, so third-party apps are not permitted to use them. The Qihoo ban was because the company distributed apps that used private APIs.

Example: Identifying Potential Security Risks

• **Architecture**
  – Apps use features provided by iOS
  – Some of the API’s are for Apple’s own use (called private APIs)
  – Enterprises may use private APIs for their own use

• **Qihoo distributed apps that used private APIs**
  – Misuse of private APIs can pose a security risk
  – Hence, apps from Qihoo were banned by Apple
Lastly, we have an example that illustrates the relationship between software architecture and team organization.

Microsoft used a modular architecture to catch up with Netscape during their browser wars of the mid 1990s. As noted in Section 3.2, Microsoft appeared to have missed the Internet disruption until it launched a companywide effort to build its own web browser. A team member observed,

“If someone asked what the most successful aspect of [Internet Explorer 3.0] was, I would say it was the job we did in ‘componentizing’ the product.”


A modular architecture meant that the components were relatively independent of each other, so they could be developed in parallel by different sub-teams. Parallel development allowed the browser to be delivered sooner, compared to sequential development.
With software, an architectural view is a way of looking at an architectural structure.

In 1995, Phillipe Kruchten developed the 4+1 view model for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views"

The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers.

The model has four views: logical, development, process and physical view. Additionally, scenarios (or use cases) are used to illustrate the architecture – hence the +1 view.
Logical Views are concerned with the functionality that the system provides to end-users. With an object-oriented approach, logical views would depict the objects and classes that are relevant to the application domain.

The development view illustrates a system from a programmer's perspective and is concerned with software management. This view focuses on modules in the static source program. These views are also called the implementation view.

The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses such issues as concurrency, performance, and scalability.

The physical view depicts the system from an engineer's point of view. It is concerned
with the topology
of software components on the physical layer, as well as the physical connections
between these components.
For example, the allocation of processes to servers is would be addressed by a
physical View.

The description of the architecture is illustrated using a small set of use cases, or
scenarios, which become a fifth
view. The scenarios describe interactions between objects and processes. They can
serve as a starting point for testing
an architecture prototype.

For our study, we'll focus mostly on logical and development views.
Information Hiding and Modules

- **Goals of modularization:** make the system easier to:
  - Understand, integrate and build, maintain (modify), test, verify, develop in collaboration with others

- **Information hiding principle**
  - Hide independently-changeable information, such as design decisions, in independently-changeable modules
  - Aim for well-defined interfaces that are stable over time that hide module implementations

- **Modules secrets**
  - Design decisions hidden inside a module

One of the most important concepts in software architecture is information hiding.

Information hiding is the principle of isolating design decisions to make software easier to change and understand. Design decisions are grouped and assigned to program units called modules.

The publically visible elements of a module are called the interface of the module. The interface defines the functionality of the module: what the module does, and the interactions between this module and the other modules. You can think of the interface like publically visible functions on a class.

By segregating design decisions from each other, we can protect other parts of the program from extensive modification if the design decision is changed.

Specifically, the information hiding principle says we should hide independently-changeable information, such as design decisions, in independently-changeable modules.

We should also aim for well-defined interfaces that are stable over time and that hide
the module implementations from other modules.

Hidden design decisions are referred to as the modules secrets.
To understand information hiding, consider an example where a store wants to count the items that it sells so that it can manage inventory and in-store displays.

Some design decisions you might consider are:

How are the counts represented? Where are they stored? How is the top selling item identified?

These decisions can be separated from actually using the counts to manage the store's inventory and displays.

These questions are all design decisions that can be separated and hidden from actually using the counts to manage the store’s inventory and displays.

Information about how the counts are implemented can be hidden in a module that is responsible for maintaining the counts.

The interface of this module will answer questions such as: What is the count for a particular item? What is the item for the highest count?
Later if you want to change how the counts are collected or how often they're re-computed, you can change that implementation without changing how the rest of the store uses the counts.
Two other important concepts in software design are coupling and cohesion.

Coupling is the degree to which modules are inter-related.

Two modules are loosely coupled if they interact only through their interfaces; they are tightly coupled if the implementation of one module depends on the implementation of the other.

The list on this slide progresses from looser (i.e. better) coupling to more tightly (worse) coupled modules.

Not meant to be an exhaustive list (modules may interact in other ways)

In designing your modules, you should aim for loose coupling.
Cohesion is the degree to which elements of a module belong together

- **Cohesion** is the degree to which the elements of a module belong together
  - *High cohesion* if module has one secret and all elements relate to that secret
  - *Low cohesion* if it has elements that are unrelated

<table>
<thead>
<tr>
<th>Cohesion</th>
<th>Group elements based on</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>One secret (e.g. parsing)</td>
</tr>
<tr>
<td>Sequential</td>
<td>Process steps (a la pipes in Unix)</td>
</tr>
<tr>
<td>Informational</td>
<td>Data that is manipulated</td>
</tr>
<tr>
<td>Temporal</td>
<td>Order in which events occur</td>
</tr>
<tr>
<td>Coincidental</td>
<td>Elements have little to do with each other</td>
</tr>
</tbody>
</table>

Cohesion is the degree to which elements of a module belong together

A module has high cohesion if the module has one secret and all its elements relate to that secret; it has low cohesion if its elements are unrelated.

The Unix philosophy of having each tool do one thing well leads to tools with high cohesion.

Different forms of cohesion reflect different approaches to grouping program elements into modules. The following list progresses from higher (better) cohesion to lower worse cohesion

Functional cohesion is when parts of a module are grouped together because they contribute to a single well-defined task. An example would be placing all the code in a compiler that parses the input program into a parser module.

Sequential cohesion refers to grouping modules together because they are steps in a process (like tying processes together with pipes in Linux)
Informational cohesion is when elements are grouped together because they operate on the same data. So, two processes that always operate on the same data base or on the same shared memory segment might be grouped together.

Temporal cohesion groups elements together by the time which they are processed. Put function A and B together because function A is always called before function B. Logging code with signal handling code.

Coincidental cohesion is the worst – when elements are grouped even though they are not really related. (Utilities class)
Although the concepts of information hiding, coupling/cohesion, and object-oriented design date back to the 1970’s, software developers still face problems related to these issues.

In 2005, a group of researchers at Microsoft asked the engineers and testers what problems they face in developing software.

7 of top 8 ...
The following is "a serious problem for me"

- Understanding the rationale behind a piece of code 66%
- Having to switch tasks because of ... teammates or manager 62%
- Being aware of changes to code elsewhere that impact my code 61%
- Finding all the places code has been duplicated 59%
- Understanding code that someone else wrote 56%
- Understanding the impact of changes I make on code elsewhere 55%
- Understanding the history of a piece of code 51%
- Understanding who “owns” a piece of code 50%

These are the results from the study. They asked which of the following are a serious problem for you.

The only one that does not have to do with modularity / information hiding is having to switch tasks
Module Descriptions and Hierarchy

- A realistic system can have hundreds of modules
  - Finding relevant modules is difficult beyond a dozen or so
- Solution
  - Group related modules into a tree-structured hierarchy
  - Provide descriptions of the modules written in plain English
- In a module hierarchy, the secret of a child module is a subsecret of its parent module

A major part of building a high-quality, cohesive software architecture is in how you organize and describe your modules.

A realistic system can have hundreds of modules, which makes it difficult to find the module you're looking for.

To help with this issue, system architects and engineers will often group modules into a tree-structured hierarchy and provide descriptions of the modules written in plain English.

In a module hierarchy, each parent module is composed from its child modules.

By design, the secret of a child module is a subsecret of its parent module.

So, one branch of the hierarchy can be studied with minimal knowledge about modules that belong to unrelated branches.
A module is a component or unit in your code you can use to hide a design decision.

Last time, we gave the example of the inventory count last class.

This slide shows some other typical secrets or design decisions you might hide in a module.

A major advantage of this approach is that you can change the design decisions internal the module without affecting other parts of the system that interact with the module.

So, on the right, the slide shows some examples the sort of changes that you might implement inside the module.

For instance, you could create a module for monitoring some sensors. Other code uses the module to read these sensors. If a new more reliable sensor is released, or you decide to update your sensor monitoring to be more accurate, the other code that uses the sensor does not have to change.

<table>
<thead>
<tr>
<th>Secret</th>
<th>Typical Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to monitor a sensor</td>
<td>New more reliable sensor, higher resolution sensor</td>
</tr>
<tr>
<td>How to control a device</td>
<td>Faster “larger” version of device</td>
</tr>
<tr>
<td>Platform characteristics</td>
<td>Faster processor, multi-processor, larger memory</td>
</tr>
<tr>
<td>How to control a display</td>
<td>Reorganization of screen real-estate, look and feel</td>
</tr>
<tr>
<td>How to exchange data</td>
<td>Protocol change</td>
</tr>
<tr>
<td>Database physical structure</td>
<td>Fields added, field access needed, field sizes change</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Change in time-space tradeoff, more accurate algorithm invented</td>
</tr>
<tr>
<td>Representation of system entities such as jobs and users</td>
<td>Change in performance requirements, more system entities</td>
</tr>
</tbody>
</table>
As another example you might have some module that hides how data is exchanged between two entities. The module might have read and write functions – but the details of how data is read or written is hidden to the user of the module. If you decide to use a different protocol for data exchange – the code that uses the module does not have to change.
It is important to document your modules.

There are different types of documents used for module descriptions.

The module interface specification gives information about how other parts of your code should use and interact with the modules in your system.

It provides details such as:
The services provided and services needed by each module
Syntax and semantics for accessing services
Data types and the program effects of different modules
Test cases used by the modules
And different design decisions and implementation notes that are important for users of the module.

In addition to the module interface specification, you might also write a module guide – which is just a textual description of the module hierarchy – with each module described by its secret.
As an example of how to use module hierarchies, let's consider a visual communication app.

This is a simple communication app – where you have a list of contacts over on the right and you can join a participant into the session by dragging their contact card into the spotlight area for the session.

I'm omitting a lot of the details because they're not important for this example
This is another view of the same application showing the screen you reach during a phone call.
This figure shows a partial module hierarchy for this app.

This application uses the model-view-controller pattern which we’ll discuss later in the course.

So, model, view, and controller are submodules of the communication app – and each of these submodules have their own submodules.

The relationship between parent and child just shows that their secrets are related. For instance, the view module encapsulates design decisions about how the application is viewed. The media buttons module is under the view module because the how the media buttons look is a subsecret of this broader set of design decisions.

Also, even if modules are not directly related, they can still use the modules across subtrees. For instance, the contact cards module can use the contacts module, without knowing how contacts are stored.
A module guide can be a useful tool for quickly understanding the modules in a system.

A module guide is just a description of a module hierarchy in plain English. The purpose of a guide is threefold:

• provide an overview of the system;
• bring out the context and assumptions behind the design approach; and
• describe the responsibilities and behavior of the modules.

The guide supplements, not replaces, a specification of module interfaces.

Someone new to the system could use the guide to navigate through the module hierarchy and find the modules that are relevant to a proposed change.

Someone familiar with the system can use the guide to check how many modules a change might affect.
Template for a Module Guide

- **Module Name**
  - Textual description
    - The responsibility of the module
    - Overview and context for the service and the secret of the module
  - Service Provided
    - Service provided to the other modules through the module interface
  - Secret
    - Service provided to the other modules through the module interface
    - Any secondary design decisions that are needed for the implementation
  - Error and Exception Handling
    - List of possible errors and exceptions

This slides shows the kind of information that what be included for each module in a module guide.

The template begins with the name and a textual description of the module. The textual description is in plain English. It includes the responsibility of the module and an overview of the design decisions that are hidden by the module.

The subsection on the service provided by the module is intended to help the reader decide whether the module is relevant to the aspect of the system that they are interested in. If the module seems relevant, then the reader can consult a module interface specification for more information.

The module guide also tells you what the secret is, not how the secret is implemented. In some cases, it is helpful to include secondary design decisions that are needed for the implementation of the secret.

And we include information about error and exception handling to help understand how
the module will behave in these cases.
So, in sum, information hiding is one of the most important principles for good design.

You use modules to hide design decisions and create black box components with abstract interfaces.

Organizing your modules into a tree or hierarchical structure and keeping good documentation in a module guide can help facilitate the use of modules.

You see and use modules every day in your programming. There are many different examples.

(cover a few)

Abstract data type – you can use a queue class without knowing how the queue is implemented.

Modules Summary

- Information hiding modules
  - Hide design decisions (secrets)
  - They can be viewed as black boxes with abstract interfaces
- Module hierarchy
  - Organizes information hiding modules into a tree structure
  - Described in the module guide
- Examples: Using modules services
  - Abstract data types: use data without knowing its representation
  - GUI creation environments: construct user interfaces without knowing how to display
  - Protocols: send and receive data – but hide channel details
  - Methods: invoke methods without knowing their implementation
Now, I want to talk about product families and product lines.

Product families and product lines are a design and development strategy that are often used to reduce engineering effort and cost by exploiting reuse.

The concept of product families in software was originally proposed or described by David Parnas, who was an early pioneer in software engineering. He also helped develop the concepts of information hiding and modular programming.

Parnas says a set of programs constitutes a family if it is worthwhile to study the programs from the set by
1) Looking at their common properties
2) And by determining the special properties of individual family members.

Today, many applications are produced as part of software product lines — that is, they are specifically designed and implemented as a family.

The annual Software Product Lines Conference has a Hall of Fame that lists about 20 organizations that have been honored for commercially successful product lines. The
application areas for their product lines include automotive software, avionics, financial services, firmware, medical systems, property rentals, telecommunications, television sets, and training
Families may be planned or result from product evolution.

The slide lists some examples of product families you might have used or heard of.

IBM 360 was a mainframe computer line developed by IBM in the 60's and 70's

Portable C compiler was a predecessor of gcc – spawned many other c compilers that were based on it
A product line is a family of products that are explicitly designed to take advantage of commonalities and variabilities between members of the family.

Commonalities are the common aspects of the family or product line.

And variabilities are the special properties that are unique to the individual members of the family.

A product line can also be decomposed into subfamilies.

Where each sub-family contributes a member to members of the product line or sub-families may themselves be product lines.

One example of a sub-family of products would be the iPod Nano line of products – which are a sub-family of the basic iPod products.

Product lines arise because people want products that come in different shapes, sizes, performance levels, and price points, and they want them all to be available at the same time. Without a family approach, each version would need to be developed and
maintained separately. A family approach can lead to an order of magnitude reduction in the cost of fielding the members of the family.

Savings come from re-use of commonalities and planning for variabilities (using modularity to improve efficiency of development).
Product Line Engineering
Underlying Assumptions

- Redevelopment Hypothesis
  - Most software development is mostly redevelopment

- Oracle Hypothesis
  - It is possible to plan for changes that are likely to be needed

- Organizational Hypothesis
  - It is possible to organize both software and the organization that develops it so as to take advantage of predicted changes

The approach to using software product lines is based on three underlying assumptions (or hypotheses)

Redevelopment says most SW development consists of creating variations of existing systems. Among the created systems, a major part is identical in any system, and the variations which make up the differences are relatively small. Thus, the redevelopment of common parts is avoided.

The oracle hypothesis says that the variability and changes to a system that are likely to be needed are predictable and so its possible to plan for changes. Without being able to plan for changes, its very hard to write code that could be used by other members of the family.

The organization hypothesis says that the organization as well as the software itself can take advantage of the predictability.
The downside to product line engineering is the high initial investment.

Product-line engineering requires a high initial investment for a variety of reasons:

(read the slide)

Management support is also essential for the product line approach to work.

For example, a company might use a small dedicated group of professionals to help manage development groups working on projects that are part of a product line.

Projects that have lacked management support or initial investment have failed to deliver improvements in productivity, quality, cost, and time to market that are promised by product-line engineering.
This figure illustrates the economic tradeoffs of product line engineering. With the traditional approach of building each family member separately, costs rise in proportion to the number and complexity of the family member. For simplicity, the schematic shows costs rising linearly with the number of family members.

With product-line engineering, there is an initial investment, which adds to the cost of the first product. The payoff begins as more products are delivered, since the incremental cost of adding a product is lower.
There is some crossover point where the initial investment starts to pay off. The greater the number of products in the family, the greater the savings past the crossover point.
A review of projects at Bell Labs, found that the crossover point for their organization was between two and three family members. So, the idea is, if you have a product where you plan to release more than two variations of the product, it makes sense to build it as a product line and do the initial investment for product line engineering.

- Bell Labs experience with several projects: crossover between 2 and 3
As I mentioned earlier, we draw linear lines to simplify the idea. A more realistic picture might look like this where using the traditional approach is fine for some time, but costs really to start to ramp up compared to the product line approach as you introduce more products.
Next, I want to talk about a project that was enormously important for the history of software architecture and the history of computing, and that was a project at Bell Labs to port the Unix operating system and much of its supporting application software to another machine.

Specifically, the project, which was conducted in 1977, was to port Unix from the PDP-11 to the Interdata-8/32.

I say it was important for software architecture and the history of computing because it led to the release of Open Systems from multiple vendors during the 1980's. In this case, Open System refers to a system with some combination of interoperability, portability, and open software standards.

The creators of Unix and the C programming language understood this importance and were actually major advocates for portable systems and software. In their report (from 1978), they wrote:

Thus, we take the position that essentially all programs should be written in a language well above the level of machine instructions. While many of the arguments...
for this position are independent of portability, portability is itself a very important goal; we will try to show how it can be achieved almost as a by-product of the use of a suitable language
Other than paving the way for portable software systems, Unix portability was important for a number of other reasons that make it worthy of our study.

For one the design of Unix illustrates information hiding

Being mostly written in a high-level language meant that many of the machine implementation details were hidden from the operating system software.

Also, the portable C compiler was the first to propose a front-end/back-end design to hide details about the upper-level language from machine dependencies in the back end.

What is front-end/back-end?

The authors of Unix also published a well-written account of their experience in porting the system software. This case study is interesting because it illustrates the design decisions and performance trade-offs that had to be made to achieve portability. (Post this to the course website)
To sort of drive the point about the importance of Unix portability, this figure shows some of the different operating systems that descended from the original Unix system.

Indeed, it's very likely that you're using a device (either laptop or cell phone) that is running an operating system descended from the original Unix system.

The quote on the slide, which says, “The real growth of Unix began only after portability had been achieved,” is from a 1993 paper by Dennis Ritchie, the co-creator of Unix and C.
C was developed for the PDP-11 on the UNIX system in 1972. Less than a year later, it was available on the Honeywell 6000. Shortly thereafter, it was available on the IBM 310 series. As soon as the compilers were available on these other machines, people began moving these programs, some of them quite substantial, from Unix to the new environments. And although portability was not an explicit goal in its design, people found that porting Unix applications written in C was relatively easy.

This figure shows the machines and system software that were available for the early versions of C. These early systems had to use different operating systems and different C compilers on each system. But the applications, written in C, were portable across the machines.

In their report, they discuss some of the issues with porting C programs. For instance, keeping the compilers compatible across machines was a challenge.
But the biggest challenge was the operating system interface for each machine. Many of the UNIX primitives (i.e. the system calls) were impossible to imitate on other operating systems.

Other problems include file format and I/O conventions that were not compatible across systems, and incompatibilities with the assemblers and loaders on the host machine.
So, the Unix portability project had three main goals

1) Write a portable C compiler. They say – "to write a compiler for C that could be changed without grave difficulty to generate code for a variety of machines."

2) Refine C for portability – For this goal, they want to make changes to the C language to improve the portability of programs written in C

3) And the final goal was to rewrite a substantial portion of Unix in C and port it to another machine
Even before they even thought to port Unix, it was clear that the C language was successful enough to warrant porting the C compiler to a variety of machines.

This figure shows the module hierarchy for their portable C compiler on the Interdata 8/32 machine.

Note, about 20% of the 8000 lines (or 1600 lines of source code) were machine dependent in this compiler.

This number both overstates and understates the difficulty in porting the compiler. In the machine-dependent code, only about a 1/2 or a 1/3 of the machine-dependent functions actually differ from machine to machine.

On the other hand, the author's say, 'the hardest part of moving the compiler is not reflected in the number of lines changed, but is instead concerned with understanding the code generation issues, the C language, and the target machine well enough to make the modifications effectively.'
The Portable C compiler design made use of information hiding to segregate the functionality for parsing the source language from the functionality for generating machine code.

The front-end parses the source code and generates an IR consisting of expression trees and code for subroutine entry/exit. Only about 13% of the front end is machine dependent (mostly for generating subroutine entry/exit).

The back-end generates most of the target machine code. Surprisingly, most of the backend code is also machine independent. That is because, even in the machine dependent routines, only about a 1/3 or 1/2 varies from machine to machine.

Because of this design, the Portable C Compiler was largely a success.

Within months, the compiler was running on a multitude of machines.

Many bugs could be fixed in machine-independent portions of the compiler and the fixes could be instantly ported to the other machines.
There were even able to build a Fortran compiler by writing a front end to process Fortran programs, and attaching this front end to the backend of the portable C compiler.
OK, now let's talk about how they actually ported the Unix operating system.

First, Johnson and Ritchie, the authors of the report, note that this is not the first a port of an operating system had been performed, but it was still pretty rare.

The difference here was that Unix was a moderately large, complete, and mature system in wide use, while previous ports were with much smaller kernels on non-production systems.

This figure summarizes their task. They want to take the source for the Unix system, run it through their compiler and install the built code on the Interdata 8/32 machine.
The kernel port was divided into three main sections: hardware primitives, device drivers, and the OS proper, and there were also a substantial amount of user level utilities that had to be ported. Let's start from the bottom.

The hardware primitives are the least portable part of the system. These are made up of 800 lines of assembly routines that provides system services to the rest of the system, such as enabling and disabling interrupts, doing I/O, or changing the memory map. This code has to be completely re-written every time Unix is ported to a new machine. Johnson and Ritchie reported that this was the most difficult part of the port.

Next, they have to port about 1,100 lines of machine-specific C code for the device drivers. The device drivers control such functionality as interrupt handling, I/O command processing, and error handling for various peripheral devices. This code also had to be changed to work with the devices on the Interdata, but Johnson and Ritchie reported that these changes were relatively straightforward.

Next, there are about 7,000 lines in C for the OS proper. Most of this code was already portable and did not have to be changed. There were about 350 lines of changes that
were necessary – mostly due to differences in the way the two machines handle memory management and function calls.

Most of the Unix code is not in the operating system, but in user-level utilities. Just the sheer size of the source code made this part of the port daunting. But Johnson and Ritchie report that porting of user-level libraries is manageable and a great deal of the code did not have to change.
OK – so what can we conclude about our experience of porting the Unix system?

The standard of portability achieved is fairly high for such an ambitious project: the operating system (outside of device drivers and assembly language primitives) is about 95 percent unchanged between the two systems; inherently machine-dependent software such as the compiler, assembler, loader, and debugger are 75 to 80 percent unchanged; other user-level software (amounting to about 20,000 lines so far) is identical, with few exceptions, on the two machines.

But there were limitations to the generality of porting systems. For example, the Interdata system as written does not take advantage of extra capability beyond its model, so it cannot take advantage of powerful machine features, such as demand paging.

More generally, algorithms do not always scale well; the optimal methods of sorting files of ten, a thousand, and a million elements do not much resemble one another. Likewise, some of the design of the system as it exists may have to be reworked to take full advantage of machines that are much more powerful than the system for
which it was originally designed
Keeping with this theme of portability, I want to discuss the idea of self-hosting programming languages.

Self-hosting is seen as an important principle for programming languages. Self-hosting means that the programming language should allow enough expression that the programming language can be written in its own language.

For example, a Pascal compiler written in Pascal is self-hosting, whereas a Pascal compiler written in C is not.

Self-hosting allows the programming language developer to use the features of the programming language for which they are responsible.

Critically, self-hosting creates a virtuous cycle in which language implementers desire to utilize advanced and/or expressive language features in performance-critical parts of the language implementation, and therefore often discover innovative ways to efficiently implement said language features.
Now, although self-hosting is seen as important, many runtime environments for high-level languages are not written in the language in which they typically run.

For example, the Java VM that I use for much of my research is written in C++. But there are good reasons to try to implement the Java VM in the Java language itself.

For instance, one of the big benefits of Java is that it gives you automatic memory management with garbage collection. This is nice because you no longer have to explicitly free memory yourself or deal with pointers. This can significantly reduce memory bugs, like memory leaks or seg faults, in your program. However, if your runtime is implemented in C, a memory bug in the runtime could still crash your application.

Another advantage is that self-hosting can simplify communication between the runtime and application layers. For instance, the runtime language might have a different format for objects than the application language. And so, the VM might have to change the format of the object from one format to the other – which can be very expensive. Having the runtime code and application code written in the same language makes communication between these layers much simpler.
And lastly, another advantage is that you can reuse the JIT compiler for the application code to compile and optimize your runtime code. (explain concept of JIT compilation). Now, any parts of the runtime that are particularly hot will be JIT compiled and adaptively optimized in the same way as your application code.
Building a self-hosting programming language requires a process known as bootstrapping.

To describe this process, we use a formalism called a T-diagram, which is depicted on this slide.

A T-diagram depicts a compiler that translates from source language $S$ to target language $T$, written in the implementation language $I$.

Cannot run a compiler like this:

\[
\text{C ---x86} \\
\text{C}
\]

We can run this compiler:

\[
\text{C ----x86} \\
\text{x86}
\]
This diagram shows how you create a boot image (also known as bootstrap JVM) for building a self-hosting JVM.

The boot image itself contains several files that represent the environment the self-hosting JVM executes in. The boot image is just code and data (just like any other compiled object file). It also contains the root map – which are the memory objects reachable from the boot image. This is useful for garbage collection.

Creating the boot image is similar to bootstrapping a static language.

So, you have a program written in Java Byte Code. That is your boot image. You need to write C code that can translate the bytecode to machine code. (That is, you have a compiler, written in C, that compiles Java Byte Code to machine code).

You compile your C compiler to machine code. Then, run the Java Byte Code through the built compiler to generate a boot image (or bootstrap JVM) in machine code.
Next, to create the self-hosting runtime system, you have your virtual machine written in Java Byte Code.

Remember, the virtual machine has its own compiler (which is also written in Java), that can convert Java to machine code.

Now, you use your bootstrap VM to convert the necessary Java code to machine code – so now the Java Byte Code compiler is represented as machine code. And your application code, written in Java, can be compiled to Java, by code that was originally written in Java.
Next, I want to go back to talking about architectural design patterns.

We briefly introduced design patterns back when we talked about the patterns described the architect Christopher Alexander, who said patterns are a problem and a 'core of a solution' to that problem which occurs over and over again.

There are multiple definitions of architectural design patterns. This one on the slide is from a text by three researchers at the software engineering institute. They say an 'architectural pattern' is a package of design decisions that is found repeatedly in practice, has known properties that permit reuse, and describes a class of architectures.

I like this definition because it highlights the importance of reusability of design patterns. The main reason we study architectural design patterns is that, by being able to recognize and apply a pattern solution to your problem, you can get the design right earlier, and save time. Using design patterns can also improve the quality of the software you write and can make it easier to maintain and reuse.
Patterns are abstracted from software found in practice. There is no complete catalog or correct set of software patterns. Actually what is and is not a pattern depends on your point of view.

For instance, at a certain level of abstraction, one might consider a linked list or hash table to be a design pattern for storing data, but that's not typically the level of abstraction that is referred to when discussing design patterns. More commonly, design patterns are described as related sets of objects and classes and how they interact.

As you gain experience architecting software, you will adapt patterns into your code. Many people do this consciously, for instance, by searching texts for patterns that match their particular problem. But many people, myself included, do this subconsciously. It's just how most people write code. For instance, if I need to write an algorithm to solve a problem, I don't usually grab the text on design patterns, but I google for similar problems.

Often times, there might be patterns that work well with a particular language. (e.g. find the most pythonic solution for a problem)
So, for the next part of the course, we'll discuss several design patterns. Next week, we'll look at a text on object-oriented design patterns by Gamma et. al. (also known as the gang of four).

First, however, I'd like to discuss a few canonical examples of design patterns. The first pattern I want to discuss is the model view controller pattern. This was originally developed for the Smalltalk programming language at Xerox in the late 1970's. It is one of the most well-known and most widely used design patterns.

It is particularly important for programming GUI applications and has been widely adopted for web applications. Actually, the Smalltalk group were also the first group to produce the WIMP (or windows, icon, mouse, pointer) user interface, which also used their MVC design pattern.
So, here's the basic idea behind the model-view-controller pattern.

The model is responsible for managing application data as well as the logic and rules of the application. For instance, if you're building an app with a web form, the model might hold the data that the user has actually entered into the form. It hides all the design decisions related to the state of the application.

The view component manages how output is presented to the user. It accesses data from the model and specifies how the data from the model is drawn on the screen. So, for instance, if the user has indicated they 'accept the terms and conditions' the model would hold that information in a data structure, while the view might show that on the screen as a checkbox.

The controller determines how user interactions with the view in the form of gestures and events cause the data in the model to change. For example, clicking on the checkbox toggles the on-off state, or typing in the text entry field changes the underlying string stored in the model.

When the model is updated by the controller, it will also send events to the view.
component to notify it that something might need to change.

In the original Smalltalk system, programmers would create GUI objects by subclassing model, view, and controller base classes. They could also create classes and objects with a tight, pre-defined relationship between the three components. For instance, an object for a pop-up dialog box would be composed of multiple instances of the different model, view, and controller objects.
Let's look at an application that employs the model view controller pattern.

This is the same communication app we saw last week. This is the Avaya Flare Experience application, which is advertised as a one-stop communications and collaboration app. The main functionality it provides is to facilitate voice and video calls between groups of people.

This slide shows two versions of the app. The one on the top shows the earlier "in the lab" version. In this version, participants in a communication session are simply shown as cards against a black screen. To indicate participants are in the same session, the application draws a white box around them.

In the production version, shown at the bottom of the slide, all of the potential participants are represented as cards on the right. The user drags cards into the spotlight in the middle of the screen to add them to a particular communication session.
This figure shows the different modules for the model, view, and controller components of the communicator application.

The model component maintains contacts data and information about the current and previous sessions.
The view component describes how contact cards, session windows, and media buttons are displayed.
And the controller component interprets commands from the user, and issues commands to update the model and view components. It also interfaces with the underlying infrastructure to initiate and end communication across a particular channel.

So, I ask you, what do you think of this design? Is it a good design? Notice each component has a session related module. We said earlier that we desire modules that exhibit high cohesion. And cohesion is the degree to which elements in the module are related. So maybe we should group these into one module?

I would say no – because even though they are related to the session – they represent fundamentally different design decisions. For highly cohesive systems, we desire
modules which encapsulate only one design decision and have elements that all relate to that decision.
I would say no – because even though they are related to the session – they represent fundamentally different design decisions. For highly cohesive systems, we desire modules which encapsulate only one design decision and have elements that all relate to that decision.

Notice also, that if we do keep them separate, it makes it much easier to update the interface for the production version of the app. Remember that the lab version had a simplified view of communication sessions. If we use this model-view-controller architecture, we can change the view for the production version by changing the Session Windows module, with very little changes to the rest of the app.
This figure summarizes the model-view-controller pattern.

Remember, the model manages the state and logic of the application. It receives updates from the controller and can send information to the view component.

The view component updates the display based off information from the model – it presents output to the user and prompts for new input.

The controller interprets the user input from the view component and determines how the model should change based off feedback from the user and other devices and events.
In the mid-1990's, Taligent, a subsidiary of IBM, aimed to refine the model-view-controller pattern for more modern applications.

The result is the model-view-presenter pattern – which they describe as a generalization of MVC – that can represent the structure of any interactive program.

Their approach starts by breaking down the problem into two fundamental questions.

That is, "How do I manage my data?" Data management is traditionally specified by the model component.

And "How does the user interact with my data?" This is traditionally specified by the view and controller components.
Taligent's model-view-presenter pattern decomposes the problem of data management into three subquestions. (refining the model component)

The first question is "What is my data?". This is the same as the model component model-view-controller pattern we just saw. It encapsulates data and the other components access this data with read and write access methods.

The second question is "How do I specify my data?". The abstractions for specifying different subsets in the model's data are called selections.

The third question is "How do I change my data?". The abstractions for representing the operations that you can perform on selections in the model are called commands.

These figures show an example of how the model, view, selections, and commands components are used in the MVP pattern.

So, consider that the model (i.e. the data is just a two-dimensional array of integers). The view component still describes how the data is displayed. You could display a two-dimensional array of integers any number of ways – using a line graph, or bar...
graph – or in a table for instance. For this example, we display it as a bar graph.

A selection in the model is some way of selecting the data. You could select a column of data (as shown here), or select a single element, or select all the data – or some other combination of the data.

A command is then an operation of the selection in the model. For example, you could change the color of the data in the chart. Or, you could edit the data, add to it, delete it, move it, etc. Depending on the type of the data, you might have different commands. For instance, with text data, you could change its font, or check its spelling.
Next, they also refine the relationship between the view and controller components of the model-view-controller patterns.

They do this by breaking the question "How does the user interact with my data?" into three more questions.

The first question is "How do I display my data?" The solution to this is the same as the view component of the traditional MVC pattern.

The next question is "How do events map into changes in my data?" This question is answered by a new component called the Interactor. The Interactor specifies the user-initiated actions like mouse movements and clicks, keyboard keystrokes, or operations on other input devices, like flipping a switch, changing a dial, or inserting a disk.

And the final question is "How do I put it all together?" This represents the function of the traditional controller component, but elevated to an application level and taking into account the new components: selection, command, and interactor.
The role of the presenter within MVP is to interpret the events and gestures initiated by the user and provide the business logic that maps them onto the appropriate commands for manipulating the model in the intended fashion.
To finish off the example from earlier with the bar chart, we would say the interactors account for the mouse tracking and events, specification of selections, menu picking, and keyboard equivalents.

And the presenter then represents the traditional "main" or "event loop" part of the application, creating the appropriate models, selections, commands, views, and interactors, and providing the business logic that directs what happens when.
In sum, the model-view-presenter pattern looks like this – where the model component is now broken into three components that answer these data management questions:

1) What is my data? (Model)
2) How do I specify my data? (Selections)
3) How do I change my data? (Commands)

And the user-interface part of the model-view-controller (which was previously shared between the view and controller components) is now made up of three components which answer these questions:

4) How do I display my data? (View)
5) How do events map into changes in my data? (Interactor)
6) How do I put it all together? (Presenter)

Together, these are the six questions a developer answers when creating a model-view-presenter based program.
Lastly, I want to discuss how this pattern can be used for distributed applications.

For these applications, there is one more design question that needs to be considered, and that is, "How do I partition my application between client and server?"

A traditional client/server split could be made by factoring the presenter. That is, the model, selections, and commands represent typical server-side functionality. The view and interactor represent typical client-side functionality. And then the presenter then "straddles" the boundary between client and server. That is, some of the presenter code is on the server, some is on the client, but they work together as a conceptual unit.

However, this is only one way to do things. Interestingly, depending on where you put the functionality seems to correspond to a broad range of popular client/server solutions. This figure indicates some of those solutions.

Suppose, for instance, the model is on the server, but everything else is on the client. This corresponds to having a remote data store or file server.
If only the model and selections are on the server, this corresponds to what a basic relational database does. For instance, the SQL "SELECT" command, which is a central primitive of relational databases, specifies what data is to be chosen for processing, but the processing (commands) is usually performed on the client side.

Suppose the model, selections, and commands are on the server. By moving the command processing to the server, we have modeled essentially a stored program database. Now the client simply requests commands for processing the data that execute on the server.

Let’s go to the other extreme. Suppose only the view is on the client. At its simplest, this is essentially what a dumb terminal does, or what an X Window server does in allowing remote viewing, or what simple screen sharing is. The client is only a view and everything else is happening on the server.

Suppose the view and the interactor are on the client, but all the data management and business logic (presenter) is on the server. This is essentially what today’s ordinary web applications do. The view is the HTML page rendered in a browser, and the interactor is using the mouse to click on HTML links or form-based entry fields and buttons. But all execution, business logic, commands, and data are on the Web (HTTP) server.

A Java-enabled Web application, where some code is downloaded for execution on the client, is essentially the beginning of migrating business logic, hence some of the presenter, to the client side. If all the logic stays on the server side, we have the classic server-based application. If all the logic goes on the client side, we have the classic PC application.

With this example, we can see that this pattern covers a broad range of computing applications, which indicates it’s a fundamental design pattern that would apply to many different situations and problems.
Now, let's talk about another type of architecture: the layered architecture.

This is one of the most common types of architectural patterns, and you've no doubt seen it in use or even used it yourself without thinking about it.

The idea is to group modules into sets or layers that are stacked vertically on top of each other. Functionality within each layer is related by a common role or responsibility.

And then modules in the different layers interact through well-defined interfaces, and communication across layers is always explicit. The goal is to keep modules in each layer loosely coupled.

There are different types of layering, but in the strictest model, components in one layer can interact only with components in the same layer or with components from the layer directly below it. More relaxed layering allows components in a layer to interact with components in the same layer or with components in any lower layer.

So, there is a relationship between the upper and lower layers. If an upper layer A
uses a lower layer B, then B must be present & satisfy its specification for A to satisfy its specification. But, by design, the lower layers have no dependency on the upper layers – allowing them to be reusable in other scenarios.

There are many examples of the layered architecture. We'll briefly discuss a couple – the internet protocol suite, TCP/IP, and virtual machines.
In sum, a good layered architecture will follow these principles.

**Abstraction.** Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles and responsibilities of individual layers and the relationship between them.

**Encapsulation.** No assumptions need to be made about data types, methods and properties, or implementation during design, as these features are not exposed at layer boundaries.

**Clearly defined functional layers.** The separation between functionality in each layer is clear. Upper layers such as the presentation layer send commands to lower layers, such as the business and data layers, and may react to events in these layers, allowing data to flow both up and down between the layers.

**High cohesion.** Well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality directly related to the tasks of that layer, will help to maximize cohesion within the layer.
**Reusable.** Lower layers have no dependencies on higher layers, potentially allowing them to be reusable in other scenarios.

**Loose coupling.** Communication between layers is based on abstraction and events to provide loose coupling between layers.
The Internet Protocol suite, or TCP/IP was originally proposed and developed by DARPA (which is a defense research agency of the US government) in the 1970’s.

The Internet Protocol uses a layered design, which I’ll discuss shortly. In 1988, David Clark at MIT, who was the chief protocol architect in the development of the Internet, wrote a paper about the design of the Internet protocols.

In the paper, Clark discusses how the design of the Internet protocols evolved. One aspect that is particularly interesting has to do with the layering of the TCP (transmission control protocol) and IP (internet protocol). Today, the TCP/IP layers are considered basic to the design of the Internet, but these parts were not considered separate layers in the original design.

Clark writes, “These changes in the Internet design arose through the repeated pattern of implementation and testing that occurred before the standards were set.”
Let’s first discuss the design of the Internet Protocol stack as we know it today.

The key design decision is that the intelligence of what you do with your data is kept in the application layer, while the routing and transmission of data in the form of packets is handled by the lower layers.

So, let me go through each of these. The application layer creates data and can communicate to other applications on the same host machine or across the network. At this layer, the applications are represented as processes, which pass data (in the form of packets) through the transport layer to send and receive data from other processes. This is also where high-level protocols, such as FTP, SSH, or HTTP operate.

The transport layer is responsible for packaging the data for host-to-host delivery. The host can be on the same system or different systems. This layer basically provides a communication channel between two processes. The channel can use a reliable protocol, such as TCP, which keeps track of communication between the sender and receiver and makes sure packets aren’t dropped or received out of order, or an unreliable protocol, such as UDP, which allows packets to be dropped or packet duplication.
The Internet layer provides best effort packet delivery. It packages the data in structures called datagrams. Each Internet Datagram holds the data you want to send along with the IP address of the source and destination to provide an unreliable, connectionless delivery system. However, since this structure only includes the source and destination IP address, it uses a routing algorithm to select the next hop as it routes the data to its destination.

The network layer includes protocols that are used to implement networks at the local level, and to interface between the hardware-oriented physical layer (which is just below it), and the more abstract, software-oriented functions of the network layer and those above it.

Difference between OSI/RM and TCP/IP:

OSI/RM is a standardized model for how the functionality of a protocol stack can be organized. It doesn’t specify the exact services and protocols to be used in each layer – whereas the TCP/IP is the result of experimental research – and actually gives details about the services that each layer provides.
This design supports what’s known as “peer-to-peer” communication between the layers of two communicating systems.

Each layer provides a protocol to communicate with its peer in the other system. When a packet is sent from one system to another, it travels down through the application, transport, internet, and network interface layers. Communication at each layer relies on the lower layers – but does not have to be aware of how they operate.
One of their original goals was that the Internet had to support multiple types of service, meaning they had to build their protocol so that it could handle different requirements for things such as speed, latency, and reliability.

So, their initial design of the transmission control protocol (TCP) at the transport layer was that it would be general enough to support these different types of service.

However, as the full range of needed services became clear, it seemed too difficult to build support for all of them into one protocol.

One example of a service outside the range of TCP was support for XNET, the cross-internet debugger. The XNET debugger needed access to whatever got through in times of stress or failure. In these times, having a protocol that guaranteed reliability may prevent data from getting through. So, they wanted a protocol that supported unreliable data transfer because it was better to have some (possibly bad) data get through than nothing at all.

Another service which did not fit TCP was real time delivery of digitized speech, which was needed to support the teleconferencing aspect of command and control.
applications. In these applications, reliability was not the primary requirement because some dropped packets could be smoothed over. The main requirement was a service that could minimize the delay in the delivery of packets.

It turns out, perhaps surprisingly, that the biggest source of delay in these systems was the mechanism to provide reliable delivery. A typical system would respond to a missing packet by requesting re-transmission, and stopping delivery of any packets until the lost packet was received. In something like voice communication, a better solution would be to just replace the lost packet with silence, and continue the conversation. At worst, maybe I’d have to ask you to repeat yourself.

So, to handle applications with different types of requirements, TCP and IP were separated into two layers. TCP supported reliable delivery of data, but some packets might be delayed. While IP supported fast delivery, with the caveat that some packets could be dropped.
So, in sum, layered architectures have proven to be enormously successful for Internet protocols.

The layered architecture supports designs with increasing levels of abstraction – making it easier to partition a complex problem into smaller sub-problems that you can focus on and solve.

It also supports enhancement without changing the other layers. For instance, the HTTP protocol that we use to specify hyperlinks on the Internet was added as a new application-layer protocol well after the lower-level Internet architecture had already been in wide use for a number of years (in 1991).

However, there are some drawbacks to this approach. One drawback is performance. The Internet protocol adds and removes header data to each packet of data as it goes up and down the protocol stack – which can be a significant source of overhead.

Additionally, this approach favors clarity for the user over ease for the designer – and what I mean by that is the designer of a new protocol might have to change their protocol to make it fit the layers of the IP suite – it might be easier to design the tool...
without having to worry about what parts go in each layer.
There are other variants of layering that are in wide use outside of the Internet protocol.

For instance, programs written in managed languages, like Java, execute in a layered runtime environment like the one shown in the image on the left. In this type of layered architecture, which is called layer bridging, a module at one layer can use modules in any layer below them.

So, for instance, the Java application might use middleware in the Java libraries, make calls directly into the JVM itself (system.GC), or issue its own system calls to the operating system.

Another type of layered architecture, called sidecar is shown in the figure on the right. In this case, the monitoring layer can access the modules in any other layer but not vice versa. In this way, the monitoring modules sort of exist outside the layered architecture.
Next, I want to talk another pattern that I know you’ve seen before, but perhaps haven’t thought of as a pattern. That is the pipe and filter pattern.

The idea is summarized here from a quote from a memo at Bell Labs by Doug McIlroy – a computer scientist and engineer who is widely regarded as the inventor of Unix pipes.

He says: (read it)
Pipe-and-Filter Pattern

- Core of a solution
  - Assemble modules into a pipeline, where the output of one module becomes the input to the next
  - Modules are independent and unaware of each other
  - Not limited to linear pipelines
  - Elements of a pipeline are co-routines; they can execute in parallel, constrained only by availability of input

- Traditional Examples
  - Unix pipelines (which are linear)
  - Exception: tee command copies input to two output streams

So, here is our definition of the pipe-and-filter pipeline.

This is pretty straightforward. Basically, we want to assemble modules into a pipeline, where the output of one module becomes the input to the next.

The modules themselves should be independent and operate unaware of each other. Each module just operates on its input and produces an output.

While its useful to think of this pattern as a linear pipeline, the pipeline does not need to be linear. For instance, has anyone heard of the tee command in Unix? tee will duplicate the output from standard out to another file or process that you’ve specified – effectively creating a tee in the pipeline.

Also note that the processes in the pipeline do not need to have any sort of parent/child relationship – they are co-routines that execute in parallel. The only constraint for a process in the pipeline might be its availability of input.
Let’s look at an example of a Unix pipeline.

Has anyone used the `tr` command?

It stands for translate. It takes input on standard in and replaces characters in the first set with characters in the second set – and prints them to standard out. If SET1 and SET2 are the same length, it will replace the letter in SET1 with the letter in the same position in SET2. If SET2 is shorter than SET1, it will repeat the last letter of SET2 as needed.

```
tr 'abc' 'def' < “baby” ➔ “edey”
tr ‘abc’ ‘x’ < “baby” ➔ “xxxy”
```

So, say you want a command that gives you all the unique words on the input – and you don’t want any punctuation or duplicate words. And you want your tool to be case-insensitive (that is, ignore if the words are upper or lower case).

In the first part of this command – we use the `-C` option – which says replace the complement of the first set with the second set. So, the first part of this command
says replace anything that is not a word with a newline.

The next part says replace all the upper case letters with lower case.

The next part sorts the input alphabetically

And the last part uses the uniq command to filter out duplicate words.

So, this is an example of how some basic commands can be strung together in a pipeline to implement some useful functionality.
A more recent example of a system that uses the pipe and filter pattern is the Dataflow model introduced by Google in June 2014.

The idea is that unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. applications like web logs, mobile usage statistics, and sensor networks need to handle this sort of streaming, unordered data).

To deal with the changing requirements of modern data processing, the authors of dataflow propose a fundamental shift of approach.

They say we should stop trying to groom unbounded datasets into finite pools of information that eventually become complete, and instead we should live under the assumption that we will never know if or when we have seen all of our data.

Current systems, like Map-Reduce and its successors like Hadoop and Spark, suffer from latency problems because they have to collect all their input data into a batch before processing it. The Dataflow model is intended to replace these systems for applications that need to handle these unbounded, unordered data sets.
As an example of an application of the Dataflow system, consider this sample application.

Say we have a streaming video provider that wants to display video ads. The platform supports online and offline views for content and ads.

The video provider wants to know how much to bill each advertiser each day.

In order to do this, they need to be able aggregate statistics about the videos and ads. They want to know things like how often and how long are videos being watched, with which content/ads, and by which demographic groups?

They want all of this information as quickly as possible, so that they can adjust budgets and bids, change targeting, tweak campaigns, and plan future directions in as close to real time as possible. Since money is involved, correctness is important.

So, existing systems would struggle to meet these requirements because they have to collect all their input data into a batch before processing it. Dataflow is designed to
handle continuous streams of unordered data like what this video provider would want to process
This is an example of using the dataflow system from Google’s tutorial about the system.

In this example, the application first sets up a pipeline and applies several “transforms” to the pipeline. The transforms are the co-routines in the pipeline.

So, they have a transform for reading the file input, breaking it up into words, counting up the words from the input and printing the formatted counts to an output file.
The first part of the pipeline, where we just break the input text into words – can be done on the fly.

Notice – if we just have a stream of input – we can just keep applying the same transformation to break up the input text into words.

However, the second part of the pipeline, where you count up the occurrences of each word, is a problem. Since there’s no end to the stream, how do you know you’ve got the right count?
To handle unbounded data, Google dataflow provides a windowing mechanism.

Basically, this mechanism allows you to attach timestamps to the individual elements in your data stream. Then, Dataflow can process the input as a succession of finite-sized windows over certain time periods.

There are multiple types of windows in the dataflow system:

Fixed windows (sometimes called tumbling windows) are defined by a static window size, e.g. hourly windows or daily windows. They are generally aligned, i.e. every window applies across all of the data for the corresponding period of time.

Sliding windows are defined by a window size and slide period, e.g. hourly windows starting every minute. The period may be less than the size – so the windows can overlap.

You can also have one-off windows, called session windows, Sessions that capture some period of activity over a subset of the data.
OK, so let's sum up our chapter on software architecture. We discussed multiple definitions of architecture and particularly software architecture. There are literally hundreds of definitions – you can browse the software engineering institute's website if you want to read some definitions of software architecture.

A good definition is that an architecture is a set of structures that satisfies the requirements.

The structures are just parts of the whole that answer questions about the problem you're trying to solve. By structuring your solution, you break it down into its constituent parts and specify the relations among the parts.

And by requirements, I mean, of course the functional requirements, and the features promised to the user, but I also mean other engineering requirements, such as performance, scalability, maintainability, and other 'ilities (like portability and extensibility). And also, non-functional requirements, such as legal, ethical, regulatory, or environmental requirements.

An important aspect of good architecture is elegance. By elegance, we mean...
pleasingly simple and easy to use and change (if necessary). If the system is easy to modify – then it is easy to understand.
Fred Brooks, the author of The Design of Design, which is a series of opinionated essays on design, says that great design has conceptual integrity.

That is – designs that exhibit unity, economy, and clarity. This means a central idea or approach is repeated throughout the system. This allows you to develop a mental model about the design that lets you make predictions about how it will operate. If you’re trying to use or understand the system and your predictions come true – you appreciate the design.

A good conceptual design also gives the development team a way to talk about the system. If you can relate your system to an easy-to-understand common set of concepts – it makes it easier to talk and think about.

A good example of conceptual integrity would be the concept of "everything's a file" in Unix systems. Obviously the files on disks are files – but by extending these concepts to other types of things that are read or written (such as pipes and sockets) it makes it easier to predict how different I/O should be used. As another example, you might find it surprising that the way the kernel shares its data structures to user space is also done by files (through the /proc filesystem), but once you understand...
this, it's obvious and easy to read and use the data exposed by /proc.
Brooks also mentions different types of design.

Routine designs are those standard models that exist that are used and reused all the time. He compares these to short suspension bridges – of which the basic design was codified into textbooks long ago. In software architecture, an example might be the client-server model or the design of a list or queue structure.

Adaptive designs are those that adapt or change an existing design for a new context or purpose. Using design patterns is an example of adaptive design. He gives the example of a tandem bicycle. Another example would be like an in-memory database. For years, databases were designed to hold data in permanent storage on the disk. As big data trends pushed for faster data processing at larger and larger scales, server systems began to add more memory -- and database software had to be adapted to do in-memory processing to achieve high performance.

And then there is original design. It is much less common, but sometimes you do have to invent a new design. For instance, someone had to design the first skyscraper or first personal computer.
In conclusion, design is a process of discovery and satisficing. Satisficing is a combination of satisfy and suffice – it just means to iterate through your options and select the one that satisfies the minimum result.

Design is iterative – you need to carefully consider and make design decisions, but be prepared to backtrack and seek alternatives.

Document why you made the decisions you've made so others can consider your reasoning.

And make sure you do not over-design, select a design that works to satisfy your requirements.

When you can, use standard models to aid in your design. In just a moment, we'll discuss design patterns. When possible, use a pattern and adapt the design as necessary. If you do decide you need a new design, recognize that being innovative is risky, and can require more time and resources to get right.
Backup
What is Architecture?

• "The art or science of building; esp. the art or practice of designing and building edifices for human use, taking both aesthetic and practical factors into account."
  – Merriam Webster Online Dictionary

• "In wider use, the term 'architecture' always means 'unchanging deep structure.'"
  – Stewart Brand, How Buildings Learn

What is architecture?

There might be many definitions.

The Oxford English Dictionary calls it the art or science of building – so it can refer to the actual practice or process of actually building system.

Writer Stewart Brand points out that architecture often refers to the ‘unchanging deep structure’ of a building.

Brand’s book, How Buildings Learn, discusses how buildings evolve over time to adapt to changing requirements – and so – a building’s architecture refers to the deep structure in the building that does not change over time.

The quote is from a book on the evolution of buildings and how buildings adapt to changing requirements over long periods