

Software Architecture (1/2)

Martin Kellogg

Software Architecture (Part 1 of 2)

Today's agenda:

- **Architecture vs Design**
- Architecture diagrams
- What makes an architecture good
- Architectural styles (with examples)

Software Architecture: motivation

Software Architecture: motivation

“There are two ways of constructing a software design:

Software Architecture: motivation

“There are two ways of constructing a software design:

- one way is to make it **so simple** that there are **obviously no deficiencies**

Software Architecture: motivation

“There are two ways of constructing a software design:

- one way is to make it **so simple** that there are **obviously no deficiencies**
 - the other is to make it **so complicated** that there are **no obvious deficiencies.**”
- Tony Hoare

Software Architecture: motivation

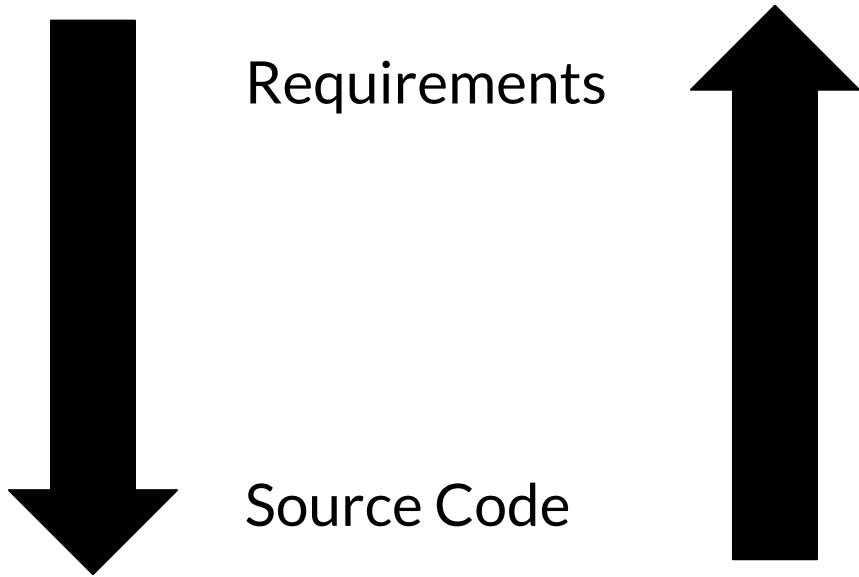
“There are two ways of constructing a software design:

- one way is to make it **so simple** that there are **obviously no deficiencies**
 - the other is to make it **so complicated** that there are **no obvious deficiencies.**”
- Tony Hoare

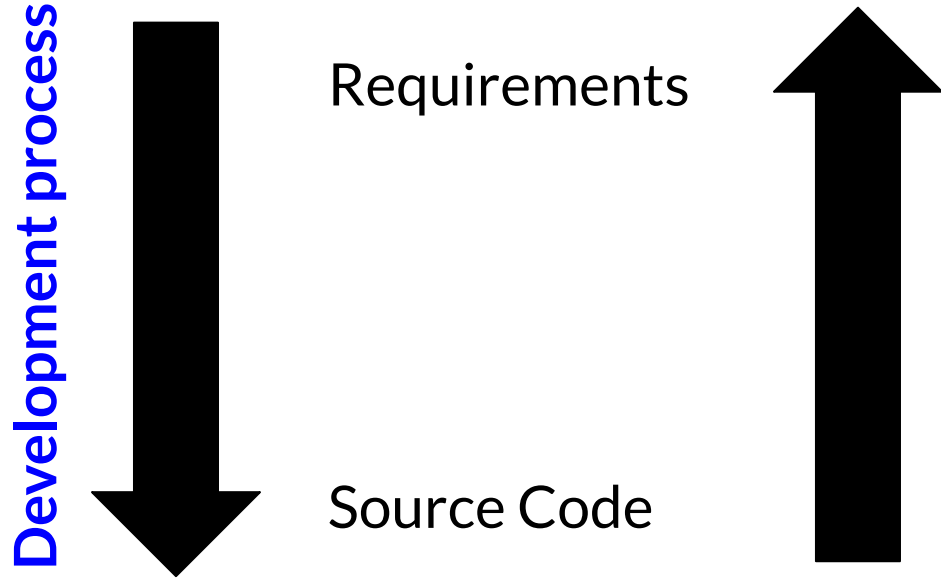
Our goals: **separation of concerns** and **modularity**

“Architecture” vs “Design”

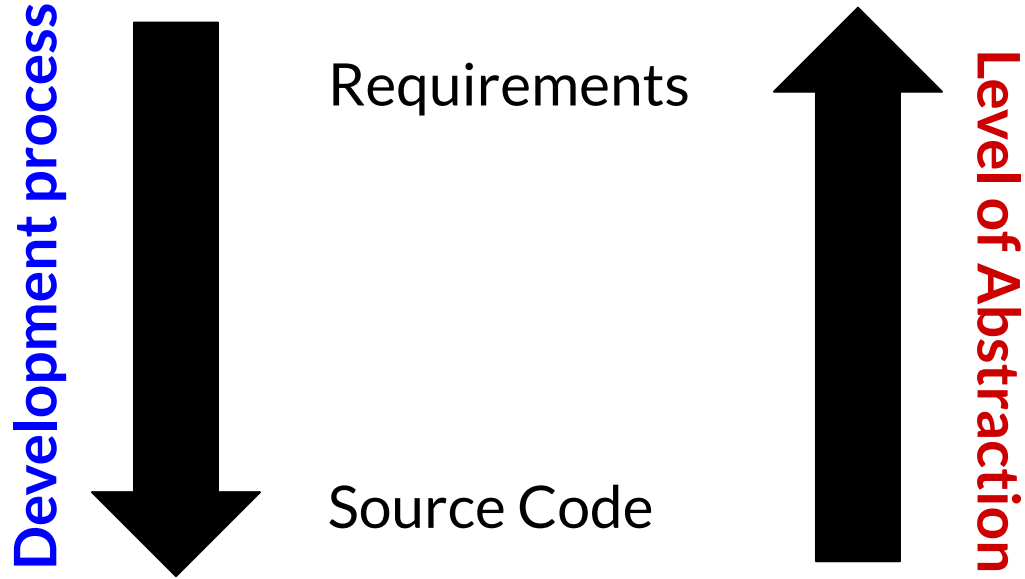
“Architecture” vs “Design”



“Architecture” vs “Design”



“Architecture” vs “Design”



Levels of abstraction

- Recall that an **abstraction** ignores some details to present a simplified representation of reality

Levels of abstraction

- Recall that an **abstraction** ignores some details to present a simplified representation of reality
- Different *levels of abstraction* are characterized by the amount of details ignored

Levels of abstraction

- Recall that an **abstraction** ignores some details to present a simplified representation of reality
- Different **levels of abstraction** are characterized by the amount of details ignored
 - more abstract = ignore more details

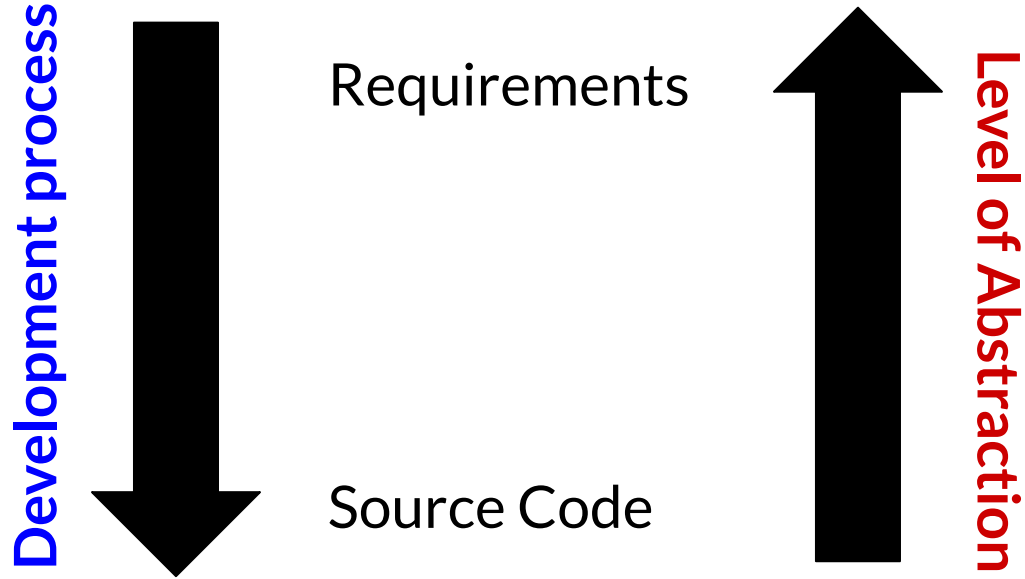
Levels of abstraction

- Recall that an **abstraction** ignores some details to present a simplified representation of reality
- Different **levels of abstraction** are characterized by the amount of details ignored
 - more abstract = ignore more details
 - which details to ignore depends on your **purpose** (analogy: what abstract values to choose in dataflow analysis?)

Levels of abstraction

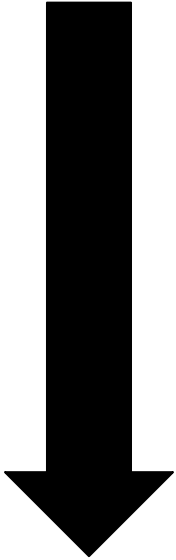
- Recall that an **abstraction** ignores some details to present a simplified representation of reality
- Different **levels of abstraction** are characterized by the amount of details ignored
 - more abstract = ignore more details
 - which details to ignore depends on your **purpose** (analogy: what abstract values to choose in dataflow analysis?)
- **Implication**: requirements have fewer details than code. Architecture and design are somewhere in the middle. But **where?**

“Architecture” vs “Design”



“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

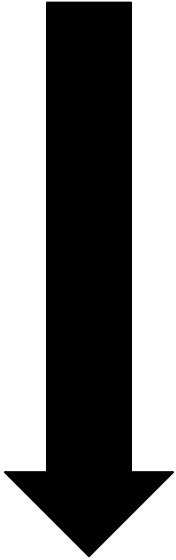
Source Code



Level of Abstraction

“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

Source Code

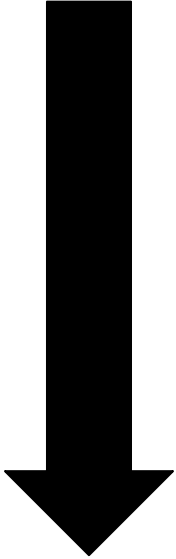


Level of Abstraction

Architecture and design are the “glue” between the **code** you actually write and what your software is **supposed to do**

“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

Source Code



Level of Abstraction

“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

Source Code



Level of Abstraction

Definition: “the *software architecture* of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them”

[L. Bass, P. Clements and R. Kazman. Software Architecture in Practice. Addison Wesley, 1999, ISBN 0- 201-19930-0.]

“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

Source Code



Level of Abstraction

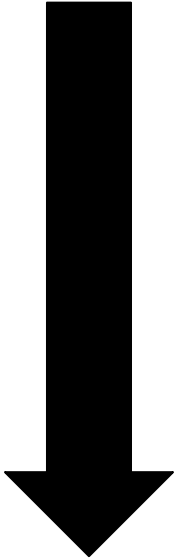
Architecture = **high-level view** of the system

Definition: “the *software architecture* of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them”

[L. Bass, P. Clements and R. Kazman. Software Architecture in Practice. Addison Wesley, 1999, ISBN 0- 201-19930-0.]

“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

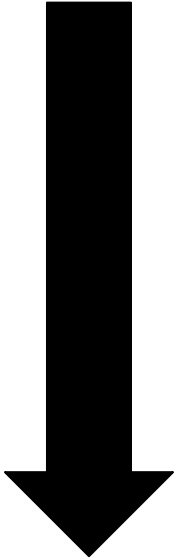
Source Code



Level of Abstraction

“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

Source Code



Level of Abstraction

Definition: *software design* is the structure or organization of a particular component of your system

“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

Source Code



Level of Abstraction

Definition: *software design* is the structure or organization of a particular component of your system

- the phrase “software design” often refers to the **process** of producing a software design

“Architecture” vs “Design”

Development process



Requirements

Architecture

Design

Source Code



Level of Abstraction

Definition: *software design* is the structure or organization of a particular component of your system

- the phrase “software design” often refers to the **process** of producing a software design
- both “design” and “architecture” are **flexible** terms, used differently by different people

“Architecture” vs “Design”: summary

- Architecture (what is developed?)
 - High-level view of the overall system:
 - What components do exist?
 - What are the protocols between components?
 - What type of storage etc.?
- Design (how are the components developed?)
 - Considers individual components:
 - Data representation
 - Interfaces, Class hierarchy
 - ...

“Architecture” vs “Design”: analogy: offices

“Architecture”



[UW Gates Center, LMN]

“Design”



[Office design, New York Times]

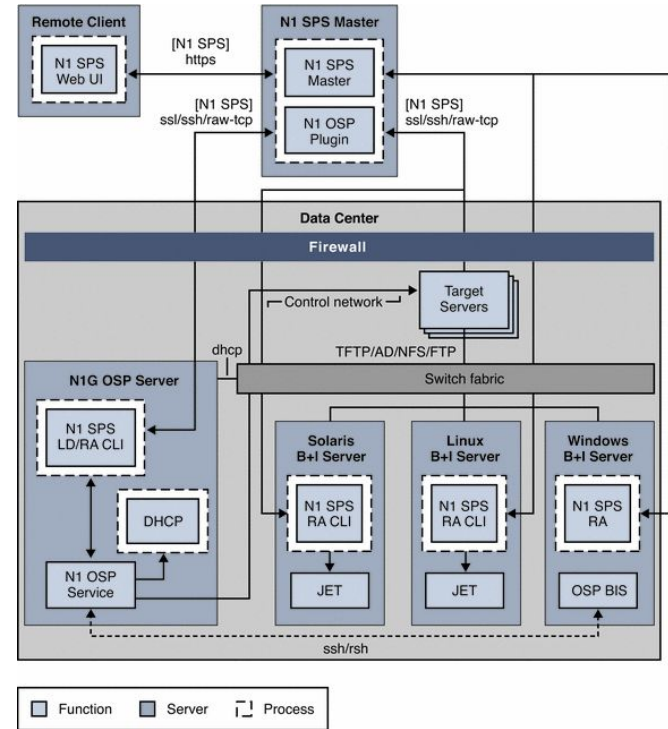
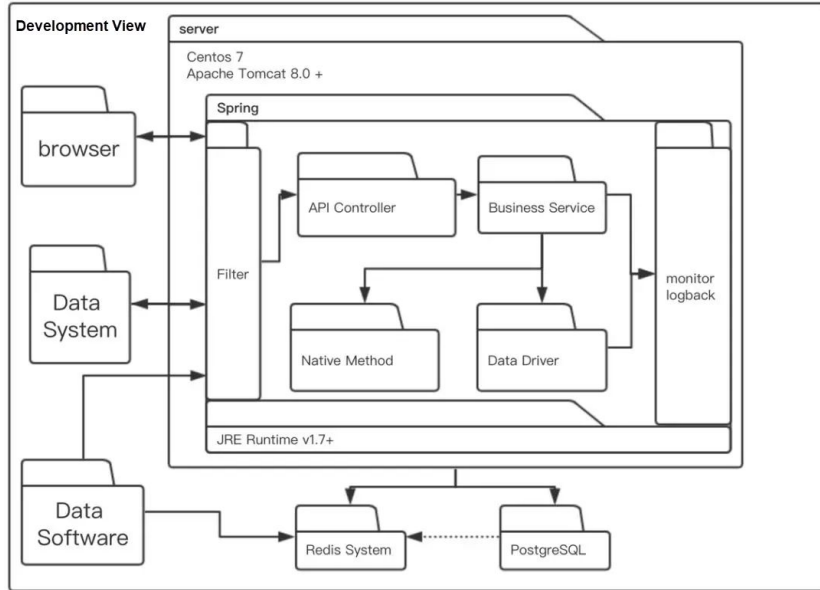
Software Architecture (Part 1 of 2)

Today's agenda:

- Architecture vs Design
- **Architecture diagrams**
- What makes an architecture good
- Architectural styles (with examples)

Architecture: diagrams

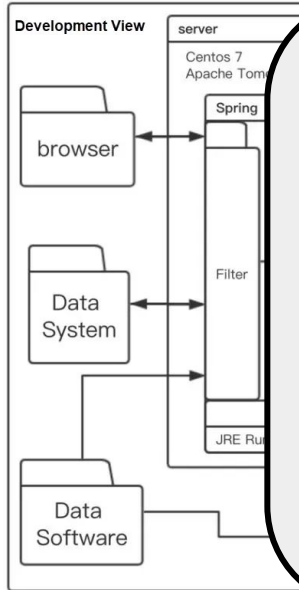
Architecture: diagrams



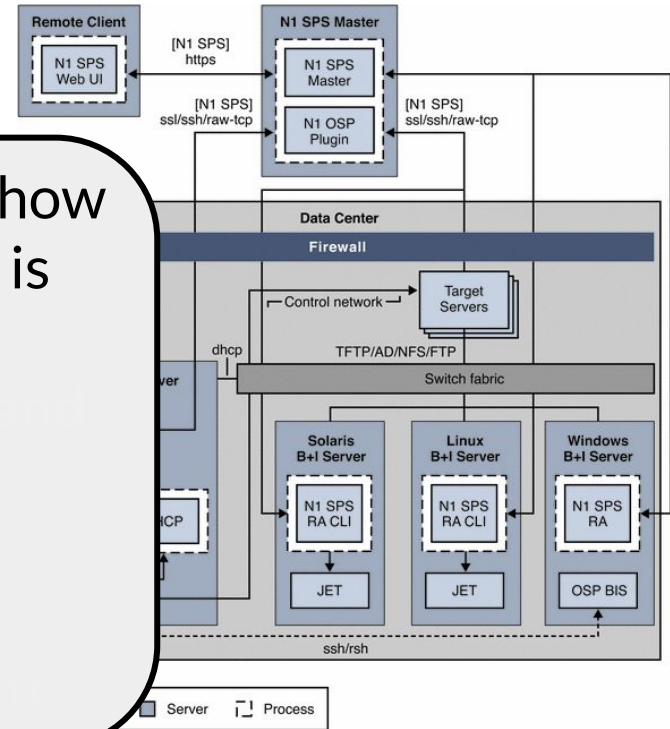
[https://www.alibabacloud.com/blog/how-to-create-an-effective-technical-architectural-diagram_596100]

[<https://docs.oracle.com/cd/E19118-01/n1.sprovsys52/819-6519/images/osp-arch-diagram.gif>]

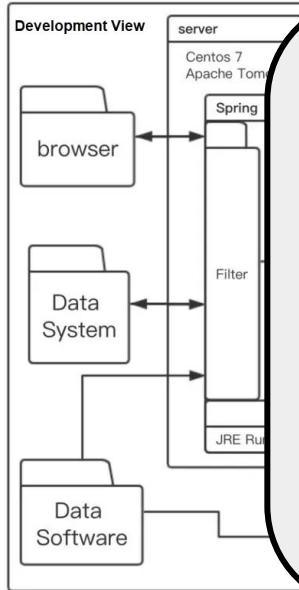
Architecture: diagrams



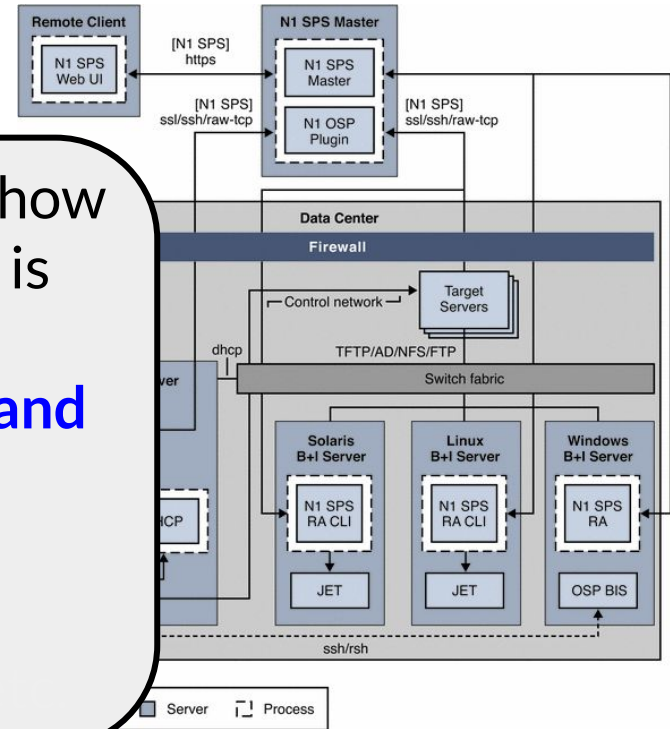
- The traditional way to show a software architecture is via a **diagram**.



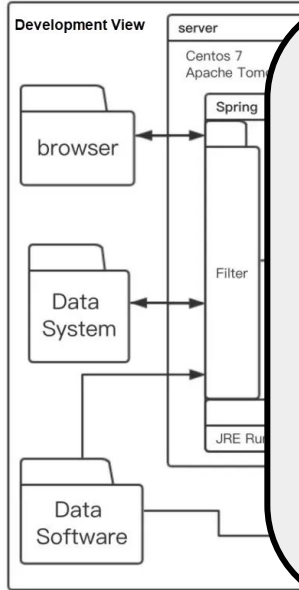
Architecture: diagrams



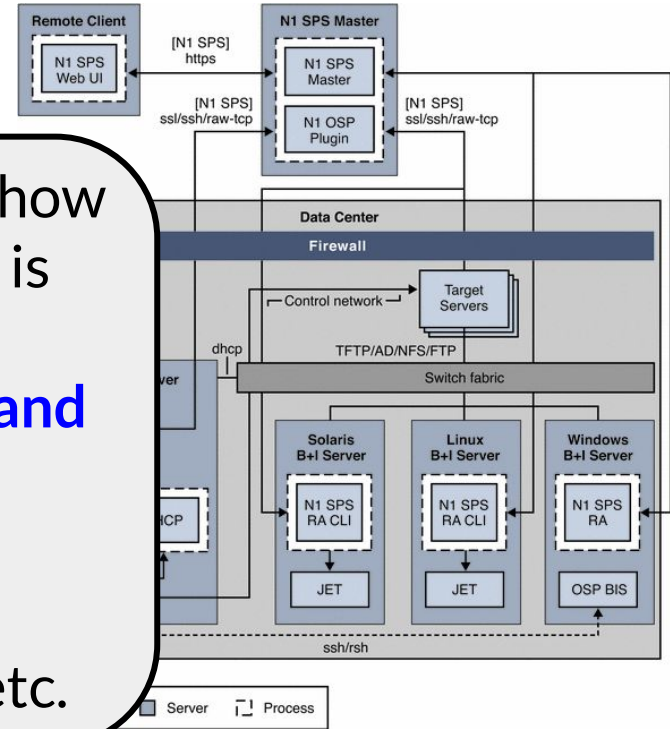
- The traditional way to show a software architecture is via a **diagram**.
- Diagrams are **common and helpful**.



Architecture: diagrams



- The traditional way to show a software architecture is via a **diagram**.
- Diagrams are **common and helpful**.
- But, what does a box represent? an arrow? a layer? adjacent boxes? etc.



Architecture: components and connectors

Architecture: components and connectors

Definition: *Components* define the basic computations comprising the system and their behaviors

- e.g., abstract data types, filters, etc.

Architecture: components and connectors

Definition: *Components* define the basic computations comprising the system and their behaviors

- e.g., abstract data types, filters, etc.

Definition: *Connectors* define the interconnections between components

- e.g., procedure calls, event announcements, asynchronous message sends, etc.

Architecture: components and connectors

Definition: **Components** define the basic computations comprising the system and their behaviors

- e.g., abstract data types, filters,

Definition: **Connectors** define the interactions between components

- e.g., procedure calls, event announcements, asynchronous message sends, etc.

Note: the line between them may be **fuzzy**. For example, a connector might (de)serialize data, but can it perform other, richer computations?

Aside: UML diagrams

- You might have heard the terms “components” and “connectors” before in the context of a **UML diagram**

Aside: UML diagrams

- You might have heard the terms “components” and “connectors” before in the context of a **UML diagram**
- UML diagrams are just one of many **modeling languages** that you can use to visualize an architecture or design

Aside: UML diagrams

- You might have heard the terms “components” and “connectors” before in the context of a **UML diagram**
- UML diagrams are just one of many **modeling languages** that you can use to visualize an architecture or design
 - UML is relatively popular, but I don't see much value in making you memorize it

Aside: UML diagrams

- You might have heard the terms “components” and “connectors” before in the context of a **UML diagram**
- UML diagrams are just one of many **modeling languages** that you can use to visualize an architecture or design
 - UML is relatively popular, but I don't see much value in making you memorize it
 - so, it's not going to be the topic of this lecture

Aside: UML diagrams

- You might have heard the terms “components” and “connectors” before in the context of a **UML diagram**
- UML diagrams are just one of many **modeling languages** that you can use to visualize an architecture or design
 - UML is relatively popular, but I don't see much value in making you memorize it
 - so, it's not going to be the topic of this lecture
 - if and when you do encounter UML, look up the symbols and map them back to the **concepts** we're discussing today

Software Architecture (Part 1 of 2)

Today's agenda:

- Architecture vs Design
- Architecture diagrams
- **What makes an architecture good**
- Architectural styles (with examples)

Properties of a good architecture

Properties of a good architecture

- Satisfies functional and performance requirements
- Manages complexity
- Accommodates future change
- Is concerned with reliability, safety, understandability, compatibility, robustness, etc.
 - but, the emphasis on these may more larger or smaller depending on the domain

Properties of a good architecture

A good architecture helps with all (or at least many) of the following:

Properties of a good architecture

A good architecture helps with all (or at least many) of the following:

- **System understanding**: interactions between modules

Properties of a good architecture

A good architecture helps with all (or at least many) of the following:

- **System understanding**: interactions between modules
- **Reuse**: high-level view shows opportunity for reuse

Properties of a good architecture

A good architecture helps with all (or at least many) of the following:

- **System understanding**: interactions between modules
- **Reuse**: high-level view shows opportunity for reuse
- **Construction**: breaks development down into work items and provides a path from requirements to code

Properties of a good architecture

A good architecture helps with all (or at least many) of the following:

- **System understanding**: interactions between modules
- **Reuse**: high-level view shows opportunity for reuse
- **Construction**: breaks development down into work items and provides a path from requirements to code
- **Evolution**: high-level view shows evolution path

Properties of a good architecture

A good architecture helps with all (or at least many) of the following:

- **System understanding**: interactions between modules
- **Reuse**: high-level view shows opportunity for reuse
- **Construction**: breaks development down into work items and provides a path from requirements to code
- **Evolution**: high-level view shows evolution path
- **Management**: helps understand work items and track progress

Properties of a good architecture

A good architecture helps with all (or at least many) of the following:

- **System understanding**: interactions between modules
- **Reuse**: high-level view shows opportunity for reuse
- **Construction**: breaks development down into work items and provides a path from requirements to code
- **Evolution**: high-level view shows evolution path
- **Management**: helps understand work items and track progress
- **Communication**: provides vocabulary; a picture says 1000 words

Properties of a good architecture: modularity

Properties of a good architecture: modularity

Definition: *modularity* is the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use

Properties of a good architecture: modularity

Definition: *modularity* is the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use

- modularity is **the key** to good architecture
 - use of abstraction leads to modularity
 - choice of abstractions is extremely important!

Properties of a good architecture: modularity

Definition: *modularity* is the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use

- modularity is **the key** to good architecture
 - use of abstraction leads to modularity
 - choice of abstractions is extremely important!
- to achieve modularity, you need:

Properties of a good architecture: modularity

Definition: *modularity* is the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use

- modularity is **the key** to good architecture
 - use of abstraction leads to modularity
 - choice of abstractions is extremely important!
- to achieve modularity, you need:
 - strong **cohesion** within a component
 - loose **coupling** between components

Properties of a good architecture: modularity

Definition: *modularity* is the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use

- modularity is **the key** to good architecture
 - use of abstraction leads to modularity
 - choice of abstractions is extremely important!
- to achieve modularity, you need:
 - strong **cohesion** within a component
 - loose **coupling** between components
 - and these properties should be true at each level

Properties of a good architecture: modularity

Definition: *modularity* is the degree to which a system's components may be separated and recombined, providing flexibility and variety in use

- modularity is **the key** to good architecture
 - use of abstraction leads to modularity
 - choice of abstractions is essential
- to achieve modularity, you need:
 - strong **cohesion** within a component
 - loose **coupling** between components
 - and these properties should be true at each level

Modularity also enables **decomposition**, which:

- decreases size of tasks
- supports independent testing and analysis
- enables separate work assignments
- eases understanding

Modularity: cohesion

Definition: *cohesion* is how closely the operations in a module are related

Modularity: cohesion

Definition: *cohesion* is how closely the operations in a module are related

- Scale is usually “**strong**” vs “**weak**”

Modularity: cohesion

Definition: *cohesion* is how closely the operations in a module are related

- Scale is usually “**strong**” vs “**weak**”
- Tight relationships **improve clarity and understanding**

Modularity: cohesion

Definition: *cohesion* is how closely the operations in a module are related

- Scale is usually “**strong**” vs “**weak**”
- Tight relationships **improve clarity and understanding**
- A class with good abstraction usually has strong internal cohesion

Modularity: cohesion

Definition: *cohesion* is how closely the operations in a module are related

- Scale is usually “**strong**” vs “**weak**”
- Tight relationships **improve clarity and understanding**
- A class with good abstraction usually has strong internal cohesion
- **Avoid** classes that have multiple, independent jobs

Modularity: cohesion

Definition: *cohesion* is how closely the operations in a module are related

- Scale is usually “**strong**” vs “**weak**”
- Tight relationships **improve clarity and understanding**
- A class with good abstraction usually has strong internal cohesion
- **Avoid** classes that have multiple, independent jobs
 - and especially avoid “**god**” classes that control the entire application!
 - such classes almost always have weak cohesion

Modularity: cohesion: strong or weak?

```
class Employee {
    public:
    ...
    FullName GetName() const;
    Address GetAddress() const;
    PhoneNumber GetWorkPhone() const;
    ...
    bool IsJobClassificationValid(JobClassification jobClass);
    bool IsZipCodeValid (Address address);
    bool IsPhoneNumberValid (PhoneNumber phoneNumber);
    ...
    SqlQuery GetQueryToCreateNewEmployee() const;
    SqlQuery GetQueryToModifyEmployee() const;
    SqlQuery GetQueryToRetrieveEmployee() const;
    ...
}
```

Modularity: cohesion: strong or weak?

```
class Employee {  
    public:  
    ...  
    FullName GetName() const;  
    Address GetAddress() const;  
    PhoneNumber GetWorkPhone() const;  
    ...  
    bool IsJobClassificationValid(JobClassification jobClass);  
    bool IsZipCodeValid (Address address);  
    bool IsPhoneNumberValid (PhoneNumber phoneNumber);  
    ...  
    SqlQuery GetQueryToCreateNewEmployee() const;  
    SqlQuery GetQueryToModifyEmployee() const;  
    SqlQuery GetQueryToRetrieveEmployee() const;  
    ...  
}
```


No problem for
cohesion here



Modularity: cohesion: strong or weak?

```
class Employee {
    public:
    ...
    FullName GetName() const;
    Address GetAddress() const;
    PhoneNumber GetWorkPhone() const;
    ...
    bool IsJobClassificationValid(JobClassification jobClass);
    bool IsZipCodeValid (Address address);
    bool IsPhoneNumberValid (PhoneNumber phoneNumber);
    ...
    SqlQuery GetQueryToCreateNewEmployee() const;
    SqlQuery GetQueryToModifyEmployee() const;
    SqlQuery GetQueryToRetrieveEmployee() const;
    ...
}
```


Probably a cohesion problem here (what does “valid” mean? is it a property of being an Employee?)



Modularity: cohesion: strong or weak?

```
class Employee {
    public:
    ...
    FullName GetName() const;
    Address GetAddress() const;
    PhoneNumber GetWorkPhone() const;
    ...
    bool IsJobClassificationValid(JobClassification jobClass);
    bool IsZipCodeValid (Address address);
    bool IsPhoneNumberValid (PhoneNumber phoneNumber);
    ...
    SqlQuery GetQueryToCreateNewEmployee() const;
    SqlQuery GetQueryToModifyEmployee() const;
    SqlQuery GetQueryToRetrieveEmployee() const;
    ...
}
```

Definitely a cohesion problem here!
(SQL query generation != model of employee)



Modularity: coupling

Definition: the *coupling* of a software project is the kind and quantity of interconnections among its modules

Modularity: coupling

Definition: the *coupling* of a software project is the kind and quantity of interconnections among its modules

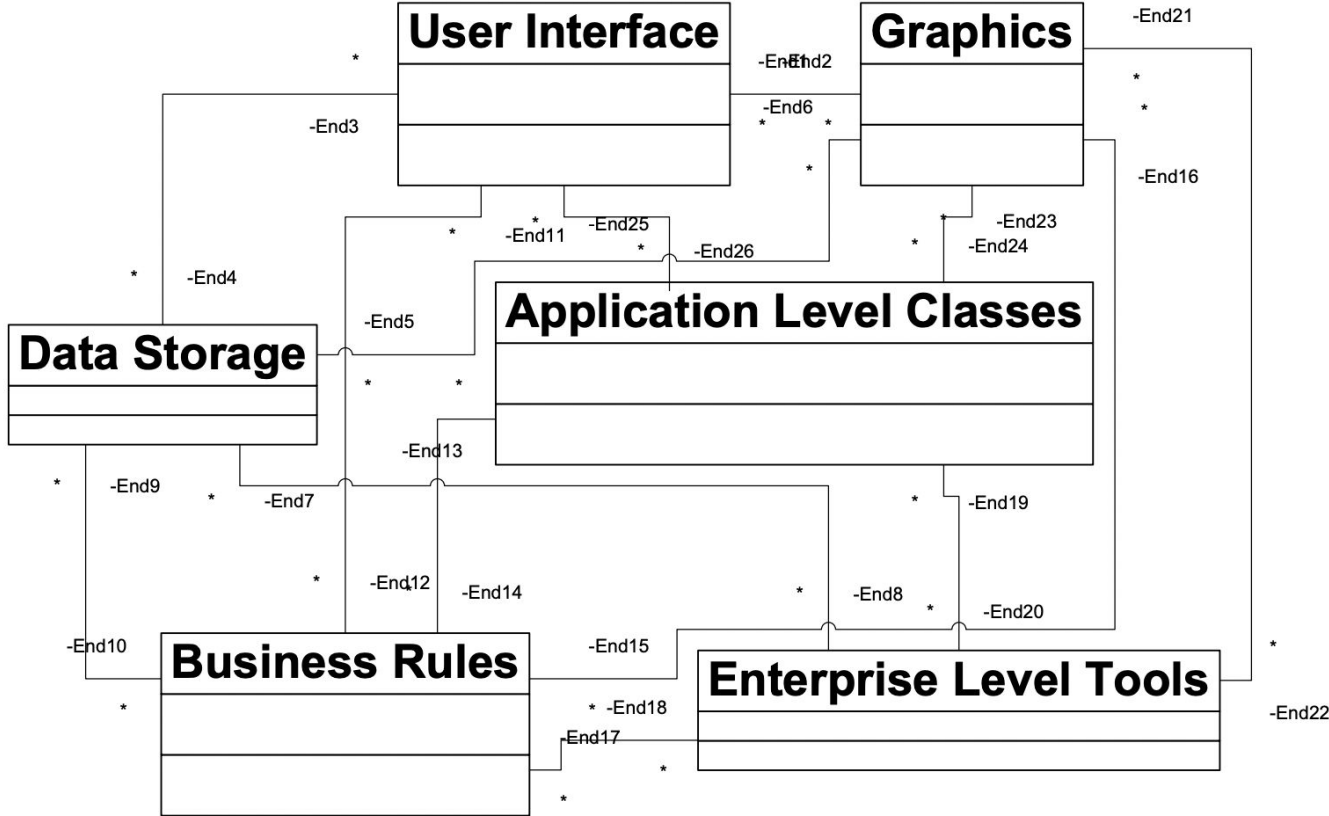
- scale: “loose” vs “tight”

Modularity: coupling

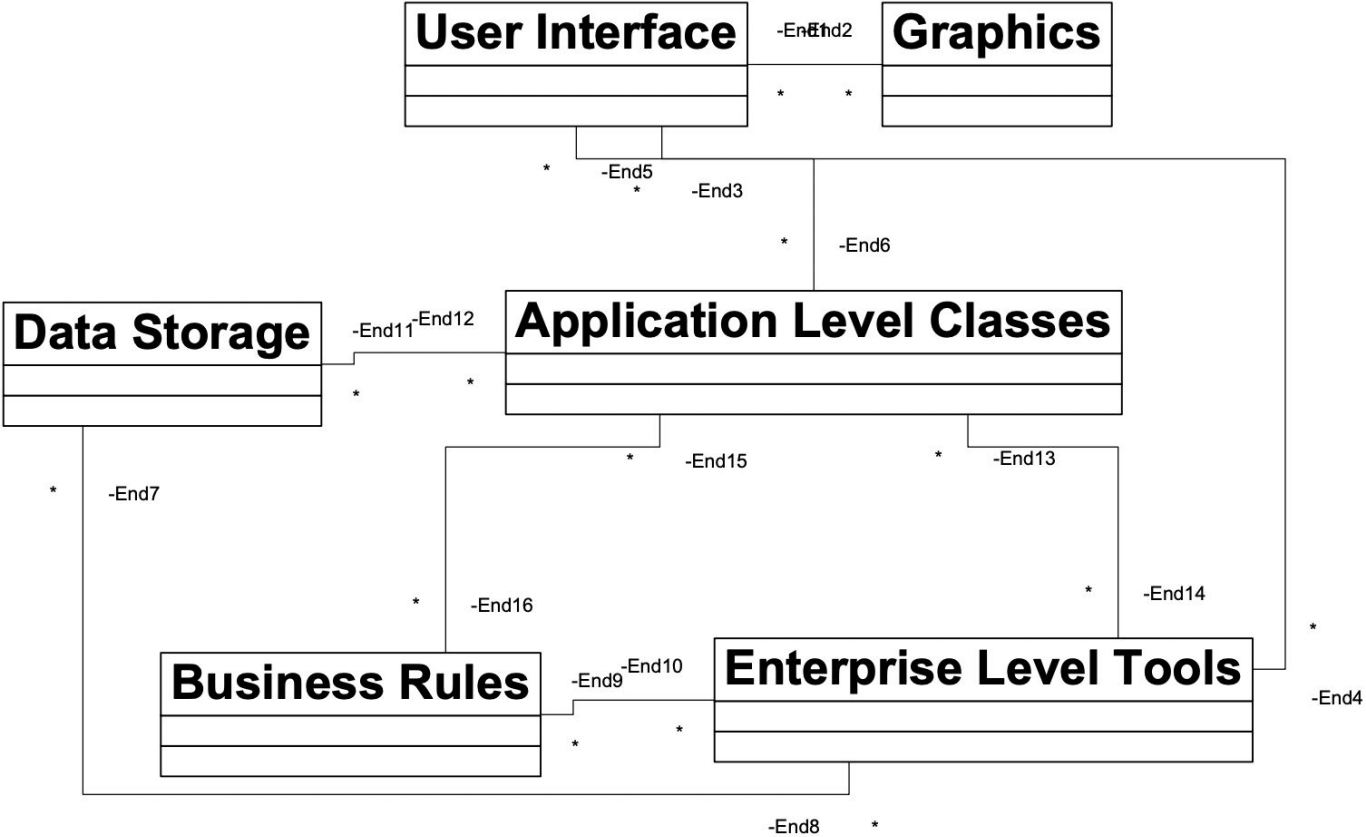
Definition: the *coupling* of a software project is the kind and quantity of interconnections among its modules

- scale: “loose” vs “tight”
- modules that are **loosely coupled** (or uncoupled) are **better** than those that are tightly coupled
 - the more tightly coupled two modules are, the harder it is to work with them separately

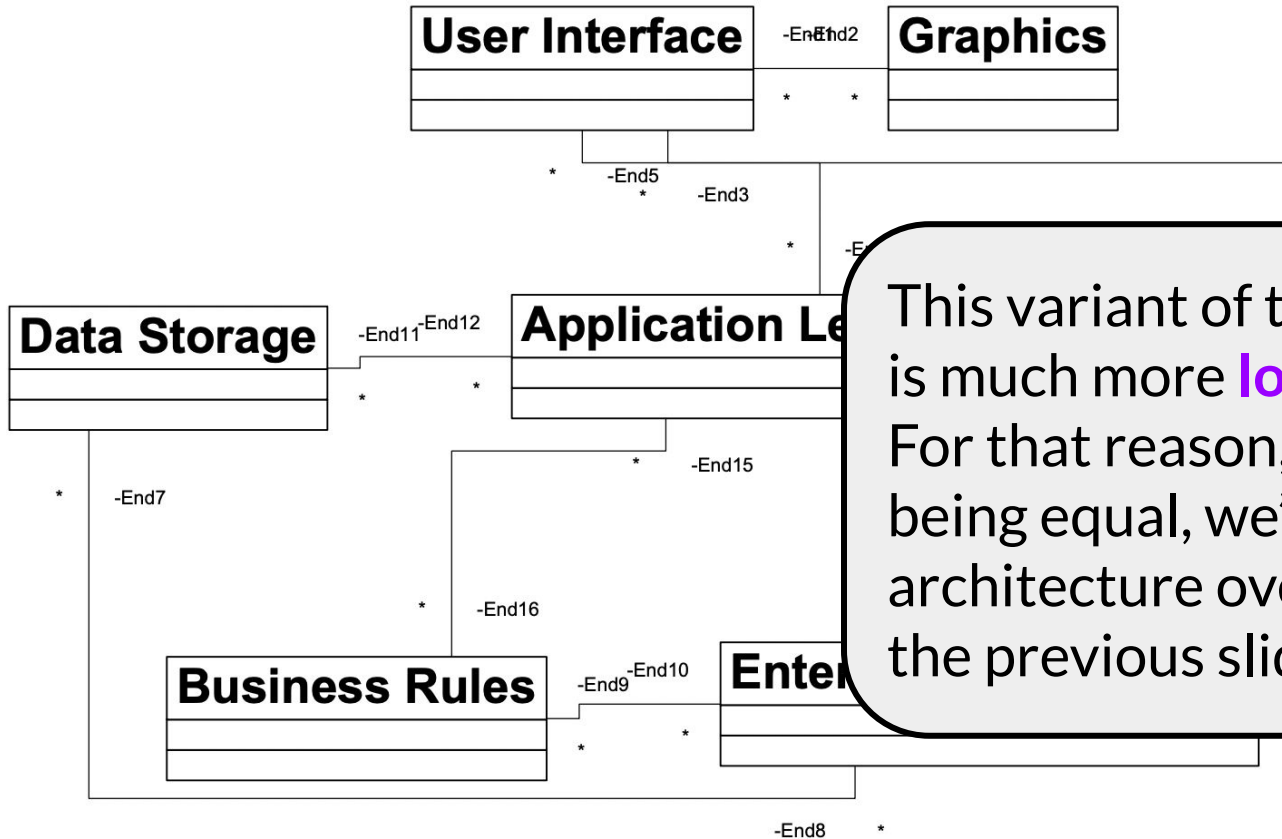
Modularity: coupling: loose or tight?



Modularity: coupling: loose or tight?



Modularity: coupling: loose or tight?



This variant of the architecture is much more **loosely coupled**. For that reason, all other things being equal, we'd prefer this architecture over the one on the previous slide.

Modularity: implementation

- How do you actually achieve modularity?

Modularity: implementation

- How do you actually achieve modularity?
 - Implementation techniques: information hiding, interfaces

Modularity: implementation

- How do you actually achieve modularity?
 - Implementation techniques: information hiding, **interfaces**

Modularity: implementation

- How do you actually achieve modularity?
 - Implementation techniques: information hiding, **interfaces**

public interface: data and behavior of the object that can be seen and executed externally by "client" code

Modularity: implementation

- How do you actually achieve modularity?
 - Implementation techniques: information hiding, **interfaces**

public interface: data and behavior of the object that can be seen and executed externally by "client" code

private implementation: internal data and methods in the object, used to help implement the public interface, but cannot be directly accessed

Modularity: implementation

- How do you actually achieve modularity?
 - Implementation techniques: information hiding, **interfaces**

public interface: data and behavior of the object that can be seen and executed externally by "client" code

private implementation: internal data and methods in the object, used to help implement the public interface, but cannot be directly accessed

client: code that uses your class/subsystem

Modularity: implementation

- How do you actually achieve modularity?

- Implementation techniques

Example: a radio

public interface: data and behavior that is exposed and executed externally by "clients"

private implementation: internal code that is used to help implement the public interface and is not accessed

client: code that uses your class/subsystem

Modularity: implementation

- How do you actually achieve modularity?

- Implementation techniques

public interface: data and behavior that is exposed and executed externally by "clients"

private implementation: internal code that is used to help implement the public interface and is not accessed

client: code that uses your class/subsystem

Example: a radio

- public interface is the speaker, volume buttons, station dial

Modularity: implementation

- How do you actually achieve modularity?

- Implementation techniques

public interface: data and behavior that is exposed and executed externally by "clients"

private implementation: internal details used to help implement the public interface that is accessed

client: code that uses your class/subsystem

Example: a radio

- public interface is the speaker, volume buttons, station dial
- private implementation is the guts of the radio: the transistors, capacitors, voltage readings, frequencies, etc. that a user should not see

Software Architecture (Part 1 of 2)

Today's agenda:

- Architecture vs Design
- Architecture diagrams
- What makes an architecture good
- **Architectural styles (with examples)**

Architecture: styles

Architecture: styles

Definition: an *architectural style* is a class of architectures sharing common features

Architecture: styles

Definition: an *architectural style* is a class of architectures sharing common features

An architectural style defines:

Architecture: styles

Definition: an *architectural style* is a class of architectures sharing common features

An architectural style defines:

- the **vocabulary** of components and connectors

Architecture: styles

Definition: an *architectural style* is a class of architectures sharing common features

An architectural style defines:

- the **vocabulary** of components and connectors
- **constraints** on the elements and their combination

Architecture: styles

Definition: an *architectural style* is a class of architectures sharing common features

An architectural style defines:

- the **vocabulary** of components and connectors
- **constraints** on the elements and their combination
 - topological constraints (no cycles, etc.)
 - execution constraints (timing, etc.)

Architecture: styles

Definition: an *architectural style* is a class of architectures sharing common features

An architectural style defines

- the **vocabulary** of components
- **constraints** on the elements
 - topological constraints
 - execution constraints

By choosing a style, one gets all the **known properties** of that style (for any architecture in that style)

- for example: performance, lack of deadlock, ease of making particular classes of changes, etc.

Architecture: styles: pipe and filter

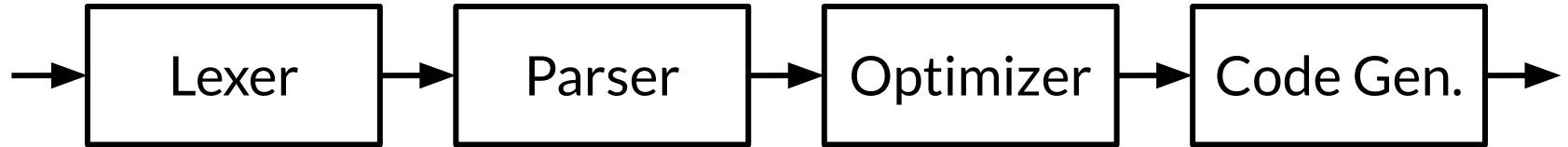
Architecture: styles: pipe and filter

Definition: a *pipe-and-filter architecture* consists of a series of discrete stages (*filters*) connected end to end (by *pipes*)

Architecture: styles: pipe and filter

Definition: a *pipe-and-filter architecture* consists of a series of discrete stages (*filters*) connected end to end (by *pipes*)

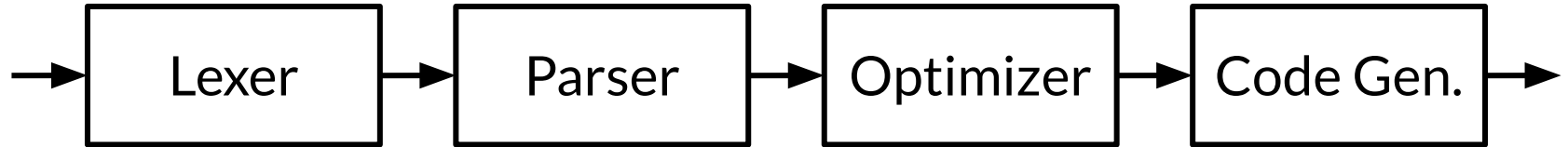
- e.g., a compiler:



Architecture: styles: pipe and filter

Definition: a *pipe-and-filter architecture* consists of a series of discrete stages (*filters*) connected end to end (by *pipes*)

- e.g., a compiler:

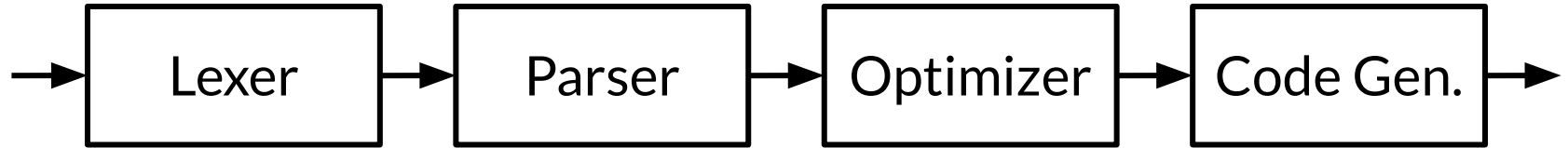


- Constraints:

Architecture: styles: pipe and filter

Definition: a *pipe-and-filter architecture* consists of a series of discrete stages (*filters*) connected end to end (by *pipes*)

- e.g., a compiler:

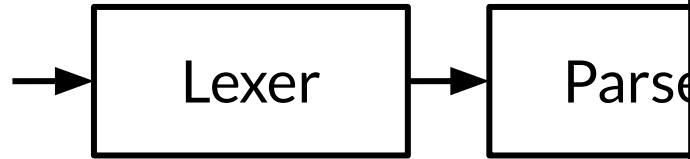


- Constraints:
 - pipes must compute local transformations
 - filters must not share state with other filters
 - there must be no cycles

Architecture: styles: pipe and filter

Definition: a *pipe-and-filter architecture* consists of a series of discrete stages (*filters*) connected

- e.g., a compiler:



If these constraints are violated, it's not a pipe-and-filter architecture anymore!

- you can't necessarily tell this from a picture, either

- Constraints:

- pipes must compute local transformations
- filters must not share state with other filters
- there must be no cycles

Architecture vs. reality

Architecture vs. reality

- Remember, the architecture is an **abstraction** of the real system

Architecture vs. reality

- Remember, the architecture is an **abstraction** of the real system
 - The code is often less clean than the architecture, with many more little details

Architecture vs. reality

- Remember, the architecture is an **abstraction** of the real system
 - The code is often less clean than the architecture, with many more little details
- The architecture is still useful (as long as the little details don't **contradict** it):

Architecture vs. reality

- Remember, the architecture is an **abstraction** of the real system
 - The code is often less clean than the architecture, with many more little details
- The architecture is still useful (as long as the little details don't **contradict** it):
 - enables easy **communication** among team members

Architecture vs. reality

- Remember, the architecture is an **abstraction** of the real system
 - The code is often less clean than the architecture, with many more little details
- The architecture is still useful (as long as the little details don't **contradict** it):
 - enables easy **communication** among team members
 - selected **deviations** can be explained more concisely and with clearer reasoning

Architecture vs. reality: interfaces

- When looking at an architecture, small details do matter a lot at the **interface** between components

Architecture vs. reality: interfaces

- When looking at an architecture, small details do matter a lot at the **interface** between components
 - e.g., NASA lost a \$125 million Mars orbiter because one engineering team used metric units while another used Imperial units

Architecture vs. reality: interfaces

- When looking at an architecture, small details do matter a lot at the **interface** between components
 - e.g., NASA lost a \$125 million Mars orbiter because one engineering team used metric units while another used Imperial units
- Architecture should warn about **incompatibility between components**, which can be caused by (among other things):
 - mismatched interfaces
 - mismatched operating assumptions (e.g., one component assumes Windows, the other assumes Linux)

Architecture: styles: other examples

Examples of architectural styles:

- pipe-and-filter
- client-server
- model-view-controller
- microservices

Architecture: styles: other examples

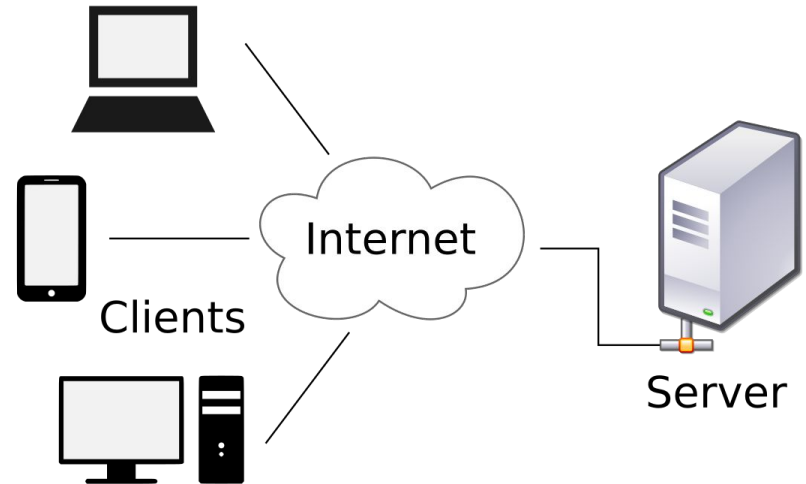
Examples of architectural styles:

- pipe-and-filter
- **client-server**
- model-view-controller
- microservices

Architecture: styles: client-server

Architecture: styles: client-server

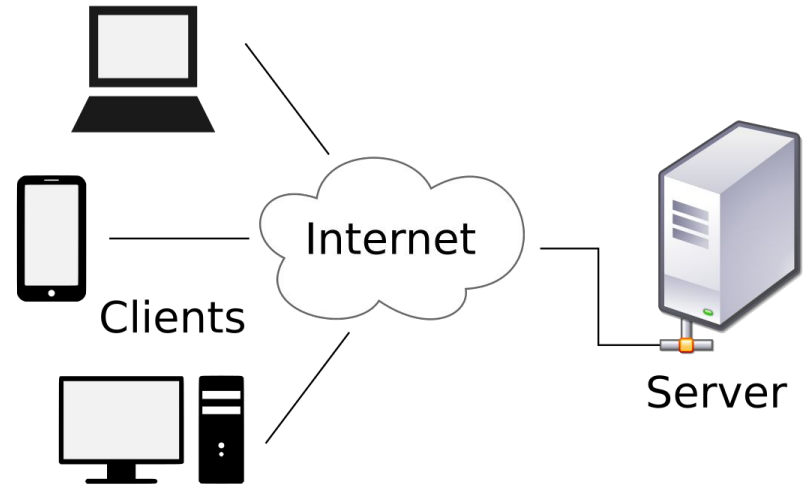
Definition: a *client-server architecture* partitions tasks or workloads between the providers of a resource or service (*servers*) and service requesters (*clients*) [Wikipedia]



Architecture: styles: client-server

Definition: a *client-server architecture* partitions tasks or workloads between the providers of a resource or service (*servers*) and service requesters (*clients*) [Wikipedia]

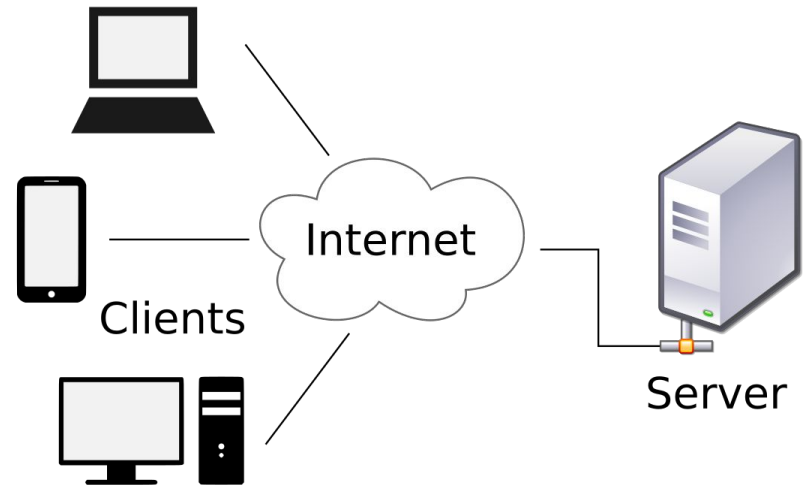
- network doesn't have to be the internet (client and server can even be on the same machine!)



Architecture: styles: client-server

Definition: a *client-server architecture* partitions tasks or workloads between the providers of a resource or service (*servers*) and service requesters (*clients*) [Wikipedia]

- network doesn't have to be the internet (client and server can even be on the same machine!)
- example of decomposition: server has its **own architecture** internally, but we don't see it



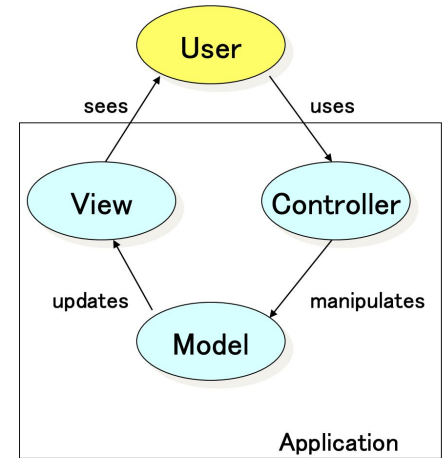
Architecture: styles: model-view-controller

Architecture: styles: model-view-controller

Definition: a *model-view-controller architecture* splits the project into three parts:

Architecture: styles: model-view-controller

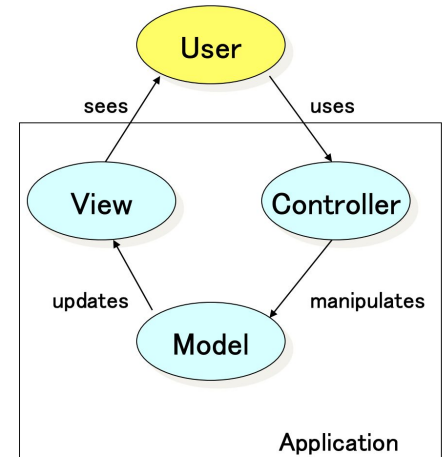
Definition: a *model-view-controller architecture* splits the project into three parts:



Architecture: styles: model-view-controller

Definition: a *model-view-controller architecture* splits the project into three parts:

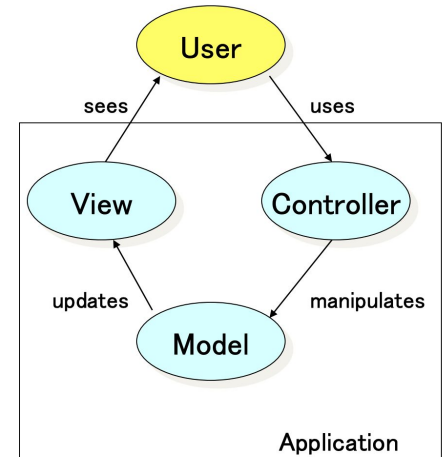
- a single *model*, which is the application's dynamic data structure, independent of the user interface



Architecture: styles: model-view-controller

Definition: a *model-view-controller architecture* splits the project into three parts:

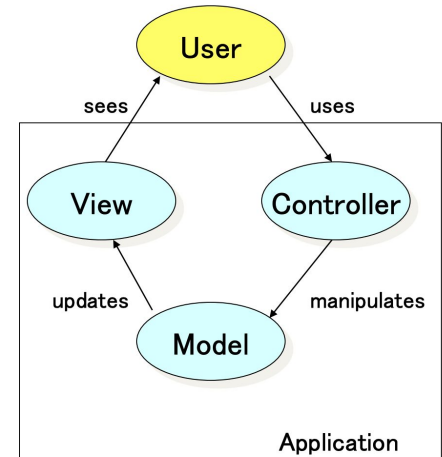
- a single *model*, which is the application's dynamic data structure, independent of the user interface
- one or more *views*, which are representations of information (e.g., charts, tables, or UIs)



Architecture: styles: model-view-controller

Definition: a *model-view-controller architecture* splits the project into three parts:

- a single *model*, which is the application's dynamic data structure, independent of the user interface
- one or more *views*, which are representations of information (e.g., charts, tables, or UIs)
- one or more *controllers*, which accept input and convert it to commands for the model or view

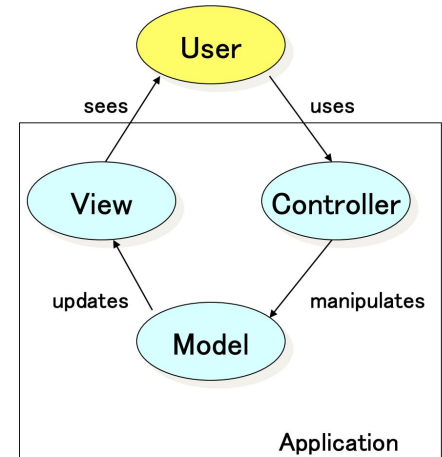


Architecture: styles: model-view-controller

Definition: a *model-view-controller architecture* splits the project into three parts:

- a single *model*, which is the application's dynamic data structure, independent of the user interface
- one or more *views*, which are representations of information
- one or more *controllers*, which convert input

Key advantage of MVC:

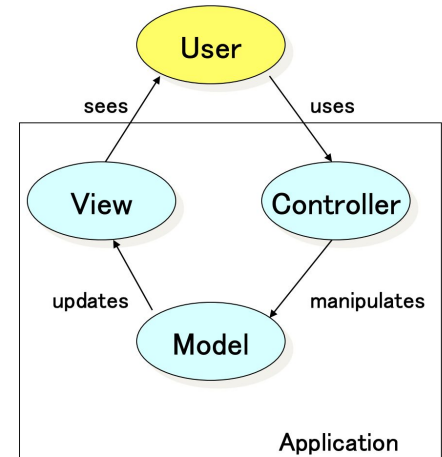


Architecture: styles: model-view-controller

Definition: a *model-view-controller architecture* splits the project into three parts:

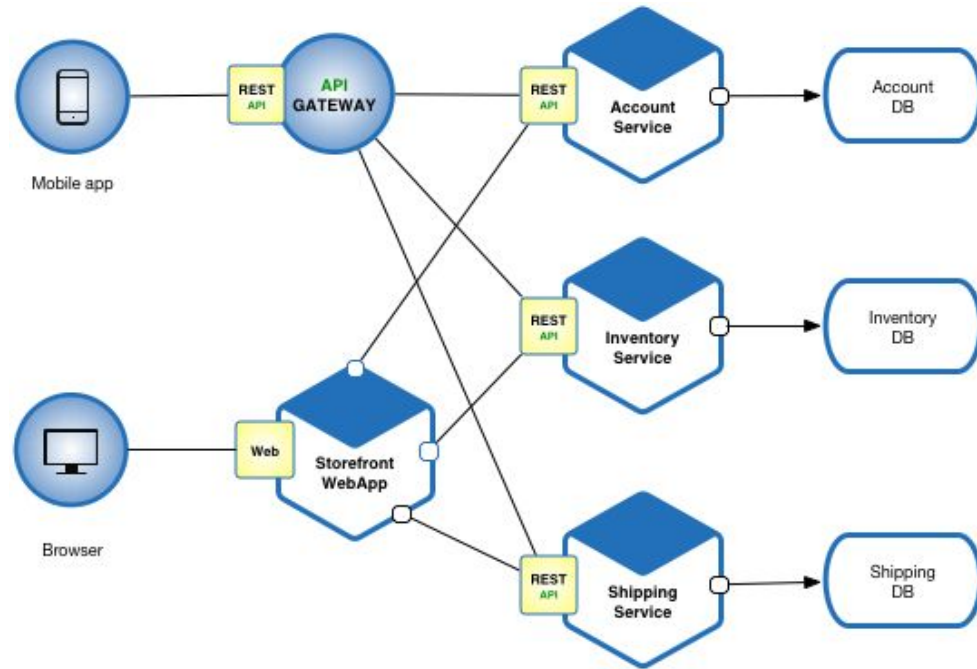
- a single *model*, which is the application's dynamic data structure, independent of the user interface
- one or more *views*, which are visual representations of information
- one or more *controllers*, which convert input

Key advantage of MVC:
separates data representation (Model), visualization/user interface (View), and client interaction (Controller)



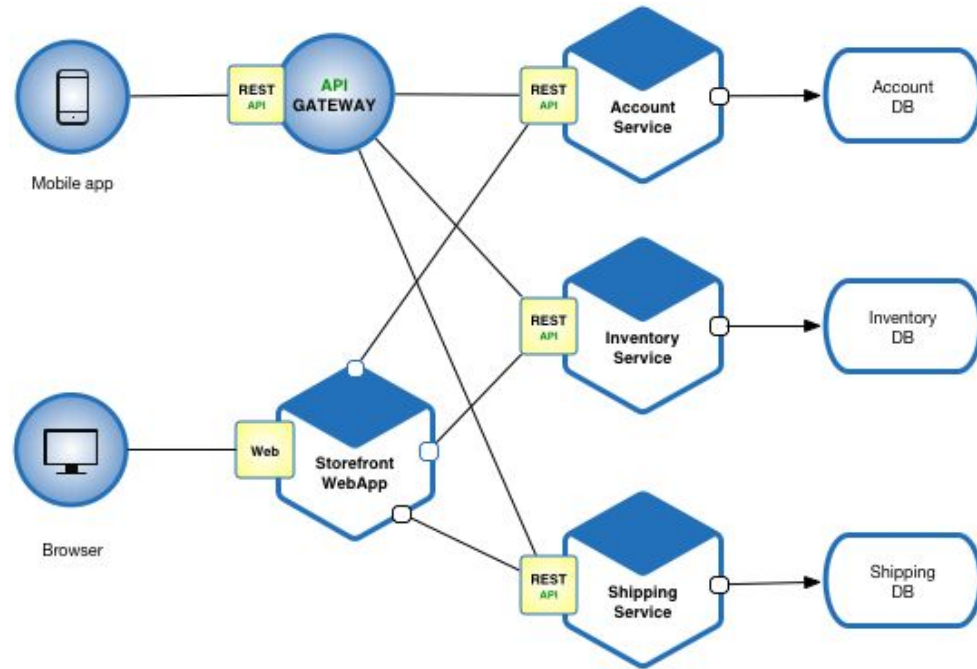
Architecture: styles: microservices

Architecture: styles: microservices



Architecture: styles: microservices

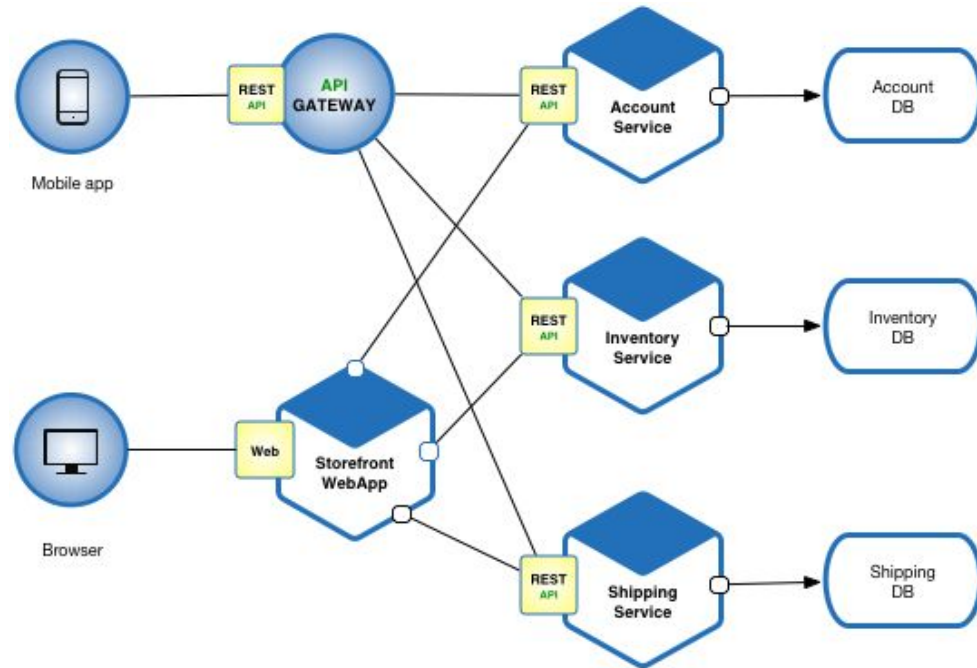
Definition: a *microservice architecture* structures an application as a collection of *services* that are:



Architecture: styles: microservices

Definition: a *microservice architecture* structures an application as a collection of **services** that are:

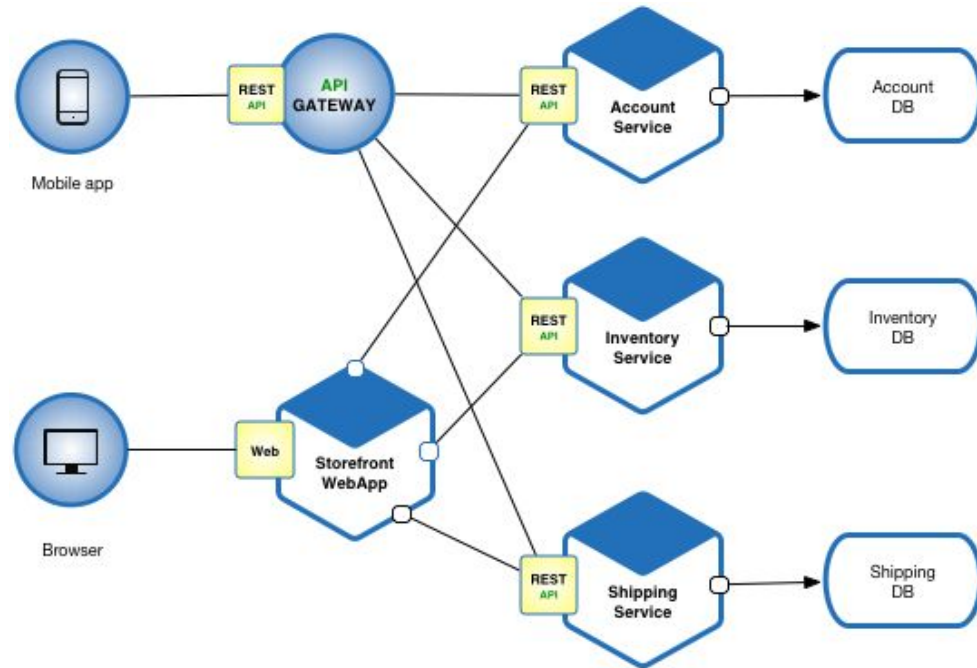
- Independently deployable



Architecture: styles: microservices

Definition: a *microservice architecture* structures an application as a collection of **services** that are:

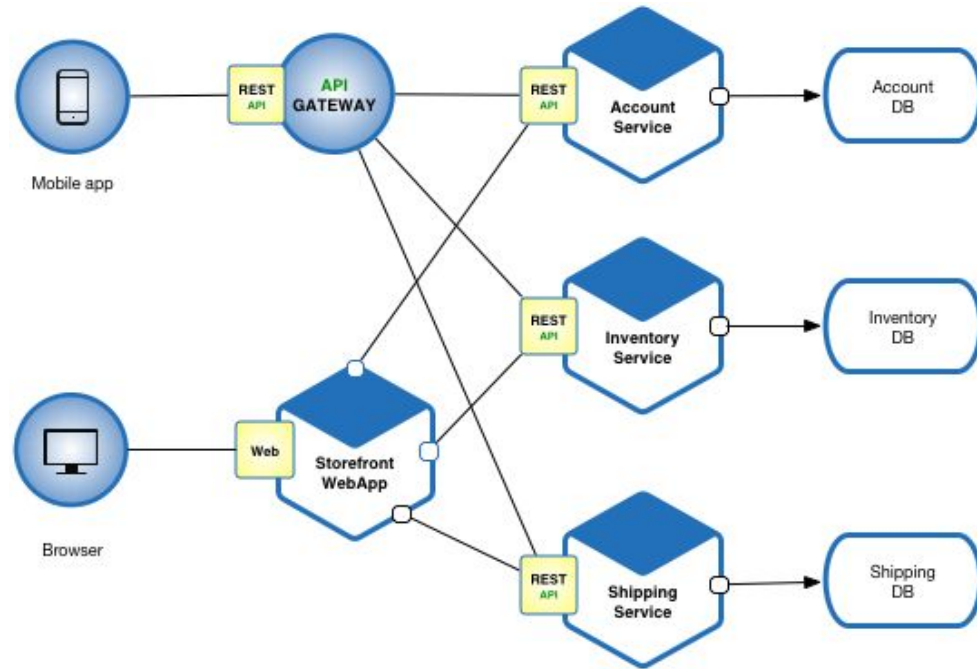
- Independently deployable
- Loosely coupled



Architecture: styles: microservices

Definition: a *microservice architecture* structures an application as a collection of *services* that are:

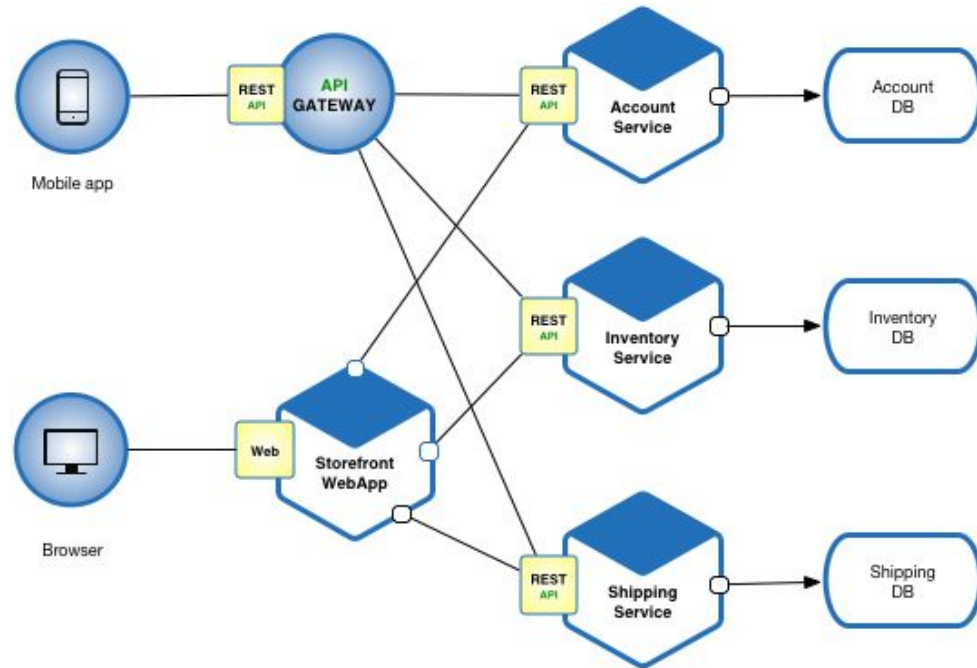
- Independently deployable
- Loosely coupled
- Organized around business capabilities



Architecture: styles: microservices

Definition: a *microservice architecture* structures an application as a collection of *services* that are:

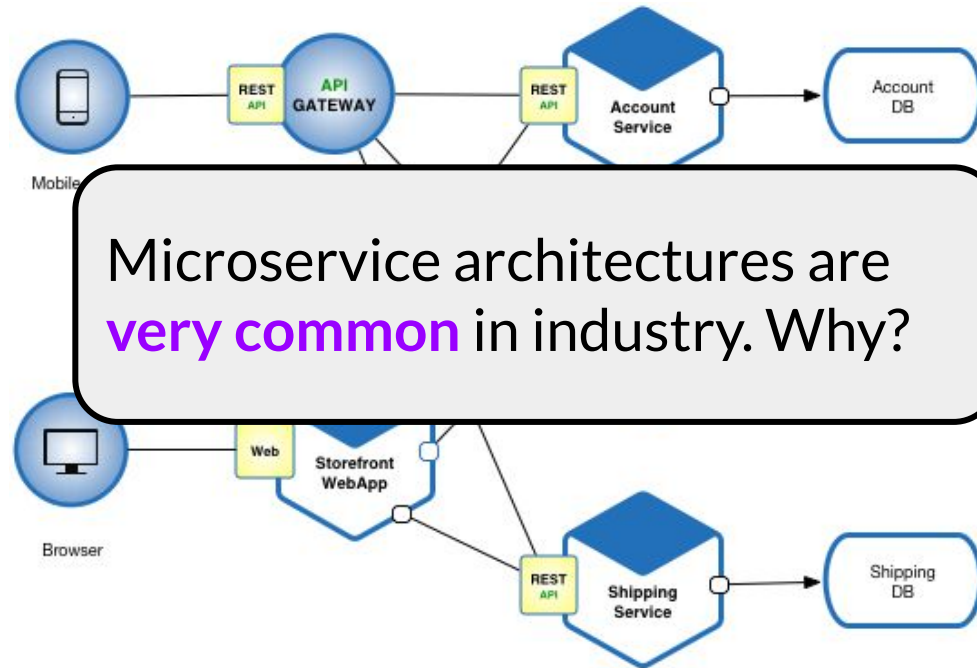
- Independently deployable
- Loosely coupled
- Organized around business capabilities
- Owned by a small team



Architecture: styles: microservices

Definition: a *microservice architecture* structures an application as a collection of *services* that are:

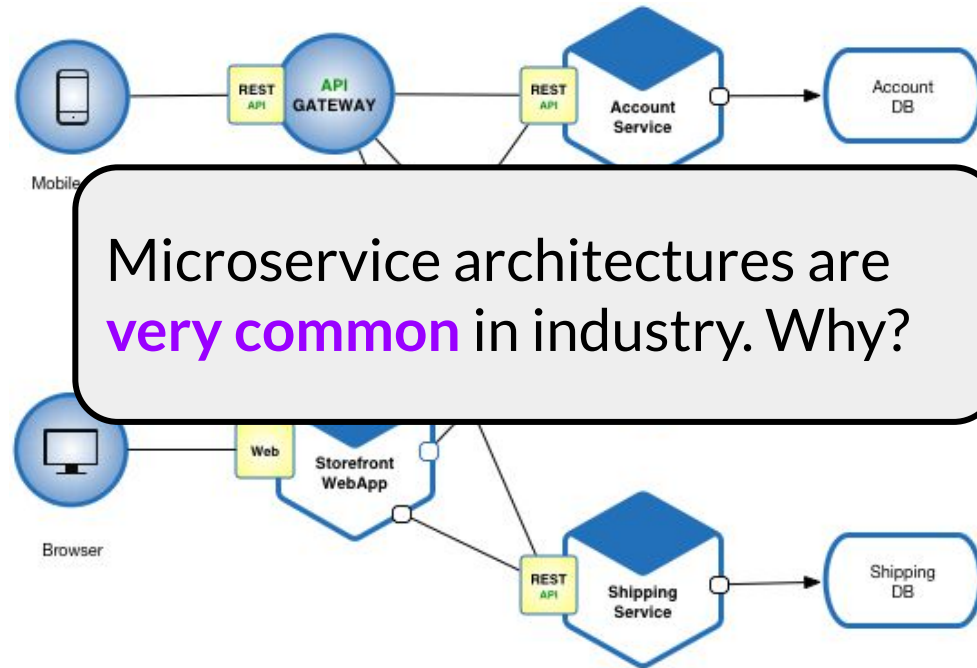
- Independently deployable
- Loosely coupled
- Organized around business capabilities
- Owned by a small team



Architecture: styles: microservices

Definition: a *microservice architecture* structures an application as a collection of *services* that are:

- Independently deployable
- Loosely coupled
- Organized around business capabilities
- **Owned by a small team (makes management easy)**



Architecture: styles: others

- This has been a whirlwind tour of a series of **examples** of common architectural styles

Architecture: styles: others

- This has been a whirlwind tour of a series of **examples** of common architectural styles
 - There are **many others!**

Architecture: styles: others

- This has been a whirlwind tour of a series of **examples** of common architectural styles
 - There are **many others!**
- **Key skill:** understand what an architecture diagram is and is not communicating

Architecture: styles: others

- This has been a whirlwind tour of a series of **examples** of common architectural styles
 - There are **many others!**
- **Key skill:** understand what an architecture diagram is and is not communicating
 - does communicate overall structure of the system

Architecture: styles: others

- This has been a whirlwind tour of a series of **examples** of common architectural styles
 - There are **many others!**
- **Key skill:** understand what an architecture diagram is and is not communicating
 - does communicate overall structure of the system
 - does communicate how components are related

Architecture: styles: others

- This has been a whirlwind tour of a series of **examples** of common architectural styles
 - There are **many others!**
- **Key skill:** understand what an architecture diagram is and is not communicating
 - does communicate overall structure of the system
 - does communicate how components are related
 - does not communicate internal structure of components
 - definitely does not tell you how to implement them!

Reading quiz: software architecture (1/2)

Reading quiz: software architecture (1/2)

Q1: One of the articles describes a rewrite of the backend system for a popular website/app due to an issue with its original architecture.

Which website/app was it?

- A. Twitter
- B. TikTok
- C. Reddit
- D. Discord

Q2: **TRUE** or **FALSE**: the author of “How architecture diagrams enable better conversations” used UML to model their system

Reading quiz: software architecture (1/2)

Q1: One of the articles describes a rewrite of the backend system for a popular website/app due to an issue with its original architecture.

Which website/app was it?

- A. Twitter
- B. TikTok
- C. Reddit
- D. Discord

Q2: **TRUE** or **FALSE**: the author of “How architecture diagrams enable better conversations” used UML to model their system

Reading quiz: software architecture (1/2)

Q1: One of the articles describes a rewrite of the backend system for a popular website/app due to an issue with its original architecture.

Which website/app was it?

- A. Twitter
- B. TikTok
- C. Reddit
- D. Discord

Q2: **TRUE** or **FALSE**: the author of “How architecture diagrams enable better conversations” used UML to model their system

Takeaways: architecture

- An architecture is a high-level view of a software system
- Good architectures communicate how the pieces of the system (the components) fit together
- Many architectural styles exist, and you should have a passing familiarity with several
 - common interview question: “on the whiteboard, design a [insert architectural style here] system to do X”
- Architectural styles are a guide, but are not prescriptive
 - real systems usually deviate from their “whiteboard architecture”, but deviations can be explained