A Tour of Software Architecture

Nick Sumner
wsumner@sfu.ca
Managing complexity through design

- Recall: Fundamental problem in software development is managing complexity
Managing complexity through design

- Recall: Fundamental problem in software development is managing complexity

- One key tool in managing and guiding complexity is software architecture
Managing complexity through design

- Recall: Fundamental problem in software development is managing complexity
- One key tool in managing and guiding complexity is software architecture
  - The overall structure of a system including its components,
Managing complexity through design

- Recall: Fundamental problem in software development is managing complexity

- One key tool in managing and guiding complexity is software architecture
  - The overall structure of a system including its components, how they communicate (interfaces & protocols),
Managing complexity through design

- Recall: Fundamental problem in software development is managing complexity
- One key tool in managing and guiding complexity is software architecture
  - The overall structure of a system including its components, how they communicate (interfaces & protocols), how they control behavior,
Managing complexity through design

- Recall: Fundamental problem in software development is managing complexity
- One key tool in managing and guiding complexity is software architecture
  - The overall structure of a system including its components, how they communicate (interfaces & protocols), how they control behavior, and nonfunctional requirements
Managing complexity through design

- Recall: Fundamental problem in software development is *managing complexity*

- One key tool in managing and guiding complexity is *software architecture*
  - The overall *structure* of a system including its components, how they *communicate* (interfaces & protocols), how they *control* behavior, and *nonfunctional* requirements

- The issues cross boundaries of scale and context
  - design patterns ↔ enterprise system designs
  - monolithic ↔ microservice
Goals of architecture

- Software architecture should help
  - *Identify* and analyze key design constraints
  - *Analyze* the trade-offs of design options
Goals of architecture

- Software architecture should help
  - Identify and analyze key design constraints
  - Analyze the trade-offs of design options
  - Guide the design of potential solutions
Goals of architecture

- **Software architecture should help**
  - Identify and analyze key design constraints
  - Analyze the trade-offs of design options
  - Guide the design of potential solutions
  - Direct and allocate *people*
Goals of architecture

- Software architecture should help
  - Identify and analyze key design constraints
  - Analyze the trade-offs of design options
  - Guide the design of potential solutions
  - Direct and allocate people

- Even architecture is iterative and incremental
  - Both analysis and design play a crucial role
  - Each will help refine the other iteratively
Goals of architecture

- Software architecture should help
  - Identify and analyze key design constraints
  - Analyze the trade-offs of design options
  - Guide the design of potential solutions
  - Direct and allocate people

- Even architecture is iterative and incremental
  - Both analysis and design play a crucial role
  - Each will help refine the other iteratively
  - Architecture will *drift*!
Goals of architecture

- Software architecture should help
  - Identify and analyze key design constraints
  - Analyze the trade-offs of design options
  - Guide the design of potential solutions
  - Direct and allocate people

- Even architecture is iterative and incremental
  - Both analysis and design play a crucial role
  - Each will help refine the other iteratively
  - Architecture will *drift*!

- Common patterns and styles arise from goals and requirements
  - (Several of which you are already supposed to know....)
Classical architectural styles [Garlan & Shaw, 1994]

- **Pipe and filter/ Pipeline**
  - *Filters* operate on data format
  - *Pipes* connect the filters together
Classical architectural styles [Garlan & Shaw, 1994]

- **Pipe and filter/ Pipeline**
  - Filters operate on data format
  - Pipes connect the filters together

![Pipe and filter diagram]
Classical architectural styles [Garlan & Shaw, 1994]

- **Pipe and filter/ Pipeline**
  - Filters operate on data format
  - Pipes connect the filters together

- **Pros:**
  - Adding filters is easy
  - Understanding flow & maintenance is easy
  - If pipes carry a common type, filters can be dynamic, reordered, ...
Classical architectural styles [Garlan & Shaw, 1994]

- **Pipe and filter/ Pipeline**
  - Filters operate on data format
  - Pipes connect the filters together

- **Pros:**
  - Adding filters is easy
  - Understanding flow & maintenance is easy
  - If pipes carry a common type, filters can be dynamic, reordered, ...

- **Cons:** Favor batch processing over incrementality
Classical architectural styles [Garlan & Shaw, 1994]

- **Pipe and filter/ Pipeline**
  - Filters operate on data format
  - Pipes connect the filters together

- **Pros:**
  - Adding filters is easy
  - Understanding flow & maintenance is easy
  - If pipes carry a common type, filters can be dynamic, reordered, ...

- **Cons:** Favor batch processing over incrementality

- **Example:** Unix Pipes
• **Client - Server**
  - Independent clients may make requests of a server
  - The server waits for requests and handles them
Classical architectural styles [Garlan & Shaw, 1994]

- **Client - Server**
  - Independent clients may make requests of a server
  - The server waits for requests and handles them
  - Often involve networking & multiple processes, but do not need to
Classical architectural styles [Garlan & Shaw, 1994]

- **Client - Server**
  - Independent clients may make requests of a server
  - The server waits for requests and handles them
  - Often involve networking & multiple processes, but do not need to

How does this relate to our discussion on complexity?
Classical architectural styles [Garlan & Shaw, 1994]

- **Client - Server**
  - Independent clients may make requests of a server
  - The server waits for requests and handles them
  - Often involve networking & multiple processes, but do not need to

- **Pros:**
  - Clients are independent & decoupled
Classical architectural styles [Garlan & Shaw, 1994]

- **Client - Server**
  - Independent clients may make requests of a server
  - The server waits for requests and handles them
  - Often involve networking & multiple processes, but do not need to

  ![Client - Server Diagram]

- **Pros:**
  - Clients are independent & decoupled

- **Cons:**
  - Clients are coupled to the server. (How easy is the server to replace?)
Classical architectural styles

- Broker
  - Servers register with a broker
Classical architectural styles

- **Broker**
  - Servers register with a broker
  - Client requests are forwarded through brokers to servers
Classical architectural styles

- **Broker**
  - Servers register with a broker
  - Client requests are forwarded through brokers to servers
  - The servers wait for requests and handle them
Classical architectural styles

- **Broker**
  - Servers register with a broker
  - Client requests are forwarded through brokers to servers
  - The servers wait for requests and handle them

How does this relate to our discussion on complexity?
Classical architectural styles

- **Broker**
  - Servers register with a broker
  - Client requests are forwarded through brokers to servers
  - The servers wait for requests and handle them

- **Pros:**
  - Clients are independent & decoupled
  - Horizontal scaling of brokers & servers
Classical architectural styles

- **Broker**
  - Servers register with a broker
  - Client requests are forwarded through brokers to servers
  - The servers wait for requests and handle them

- **Pros:**
  - Clients are independent & decoupled
  - Horizontal scaling of brokers & servers

- **Cons:**
  - Brokers themselves become a single point of failure
  - Starts to involve many components (complexity)
Classical architectural styles [Garlan & Shaw, 1994]

- Publish-Subscribe (event based / observer / ...)
  - Event subscribers register with a mediator or by broadcast
Classical architectural styles [Garlan & Shaw, 1994]

- Publish-Subscribe (event based / observer / ...)
  - Event subscribers register with a mediator or by broadcast
  - Publishers broadcast that events have happened
Classical architectural styles [Garlan & Shaw, 1994]

- Publish-Subscribe (event based / observer / ...) 
  - Event subscribers register with a mediator or by broadcast 
  - Publishers broadcast that events have happened 
  - Subscribers are notified & process events
Classical architectural styles [Garlan & Shaw, 1994]

- **Publish-Subscribe (event based / observer / ...)**
  - Event subscribers register with a mediator or by broadcast
  - Publishers broadcast that events have happened
  - Subscribers are notified & process events

- **Pros:**
  - Highly decoupled. No concrete knowledge of specific actors.
  - Very easy reuse.
Classical architectural styles [Garlan & Shaw, 1994]

- **Publish-Subscribe (event based / observer / ...)**
  - Event subscribers register with a mediator or by broadcast
  - Publishers broadcast that events have happened
  - Subscribers are notified & process events

- **Pros:**
  - Highly decoupled. No concrete knowledge of specific actors.
  - Very easy reuse.

- **Cons:**
  - No guarantees on ordering
  - If actors are not actually independent, it becomes challenging to understand
Classical architectural styles [Garlan & Shaw, 1994]

- Layered
  - Cohesive abstractions separated into layers
Classical architectural styles [Garlan & Shaw, 1994]

- **Layered**
  - Cohesive abstractions separated into layers

<table>
<thead>
<tr>
<th>Persistence</th>
<th>Logic</th>
<th>Presentation</th>
</tr>
</thead>
</table>

![Layered Architecture Diagram](image-url)
Classical architectural styles [Garlan & Shaw, 1994]

- **Layered**
  - Cohesive abstractions separated into layers

```
<table>
<thead>
<tr>
<th>Presentation</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>Application</td>
</tr>
<tr>
<td>Persistence</td>
<td>Domain</td>
</tr>
<tr>
<td></td>
<td>Infrastructure</td>
</tr>
</tbody>
</table>
```

[Cunningham]
Classical architectural styles [Garlan & Shaw, 1994]

- **Layered**
  - Cohesive abstractions separated into layers

```
Presentation
Logic
Persistence
```

```
View
Application
Domain
Infrastructure
```

- **Pros:**
  - Clear interfaces can allow layers to be replaced
  - Each layer can be focused

[Cunningham]
Classical architectural styles [Garlan & Shaw, 1994]

- **Layered**
  - Cohesive abstractions separated into layers

  ![Layered Architecture Diagram]

- **Pros:**
  - Clear interfaces can allow layers to be replaced
  - Each layer can be focused

- **Cons:**
  - How can we identify clear layer boundaries?
  - Higher layers may be coupled to lower layers

[Cunningham]
Classical architectural styles [Garlan & Shaw, 1994]

- **Others**
  - MVC, MVVM, ...
  - Blackboard
  - Repository
  - Table driven
  - ...

More recent styles

- Layering and decoupling are pushed further
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
More recent styles

- Layering and decoupling are pushed further
- **A problem with layers:**
  - Higher layers may be coupled with lower layers
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
More recent styles

- Layering and decoupling are pushed further

- **A problem with layers:**
  - Higher layers may be coupled with lower layers

Dependency Inversion from SOLID design
More recent styles

- Layering and decoupling are pushed further
- **A problem with layers:**
  - Higher layers may be coupled with lower layers

What happens when we apply this to all interactions with the domain?
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”

Admin Console → Domain
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”
More recent styles

- Layering and decoupling are pushed further
- A problem with layers:
  - Higher layers may be coupled with lower layers
- Focus on your “layer” or component at the core, and depend on interfaces for other “layers”

This is known as:
- Hexagonal architecture
- Ports & adaptors
- Onion architecture
- Clean architecture
More recent styles

- This idiom is common in many contexts
More recent styles

• This idiom is common in many contexts
  – Modern monoliths (single program apps)
  – Service oriented architecture
More recent styles

- This idiom is common in many contexts
  - Modern monoliths (single program apps)
  - Service oriented architecture
  - Microservices
  - ...
You have seen several UML diagrams in CMPT 276 that help communicate
- Be careful. UML is only a tool for communication. It is not design.
Visualizing Designs

- You have seen several UML diagrams in CMPT 276 that help communicate
  - Be careful. UML is only a tool for communication. It is not design.

- 2 Common hurdles prevent visualizing & discussing design well
You have seen several UML diagrams in CMPT 276 that help communicate.

- Be careful. UML is only a tool for communication. It is not design.

2 Common hurdles prevent visualizing & discussing design well

- Hierarchy / Abstraction
Visualizing Designs

- You have seen several UML diagrams in CMPT 276 that help communicate
  - Be careful. UML is only a tool for communication. It is not design.

- 2 Common hurdles prevent visualizing & discussing design well
  - Hierarchy / Abstraction
Visualizing Designs

- You have seen several UML diagrams in CMPT 276 that help communicate
  - Be careful. UML is only a tool for communication. It is not design.

- 2 Common hurdles prevent visualizing & discussing design well
  - Hierarchy / Abstraction
Visualizing Designs

- You have seen several UML diagrams in CMPT 276 that help communicate
  - Be careful. UML is only a tool for communication. It is not design.

- 2 Common hurdles prevent visualizing & discussing design well
  - Hierarchy / Abstraction
  - Perspective
Visualizing Designs

- You have seen several UML diagrams in CMPT 276 that help communicate
  - Be careful. UML is only a tool for communication. It is not design.

- 2 Common hurdles prevent visualizing & discussing design well
  - Hierarchy / Abstraction
  - Perspective

| Person | state ∈ {running, sleeping} |
Visualizing Designs

- You have seen several UML diagrams in CMPT 276 that help communicate
  - Be careful. UML is only a tool for communication. It is not design.

- **2 Common hurdles prevent visualizing & discussing design well**
  - Hierarchy / Abstraction
  - Perspective

<table>
<thead>
<tr>
<th>Person</th>
<th>state ∈ {running, sleeping}</th>
</tr>
</thead>
</table>

![Diagram showing state transition between Running and Sleeping](image)
Visualizing Designs

- You have seen several UML diagrams in CMPT 276 that help communicate
  - Be careful. UML is only a tool for communication. It is not design.

- 2 Common hurdles prevent visualizing & discussing design well
  - Hierarchy / Abstraction
  - Perspective

<table>
<thead>
<tr>
<th>Person</th>
<th>state $\in {\text{running, sleeping}}$</th>
</tr>
</thead>
</table>

```
<table>
<thead>
<tr>
<th>Person</th>
<th>state $\in {\text{running, sleeping}}$</th>
</tr>
</thead>
</table>
```

```
Person

state $\in \{\text{running, sleeping}\}$
```

```
Running

sleep()

run()

Sleeping
```

```
Requester

request()

tick()

Handler

respond()

Scheduler
```

```
Requester

request()

tick()

Handler

respond()

Scheduler
```
Visualizing Designs

- You have seen several UML diagrams in CMPT 276 that help communicate
  - Be careful. UML is only a tool for communication. It is not design.

- 2 Common hurdles prevent visualizing & discussing design well
  - Hierarchy / Abstraction
  - Perspective

- Consider a system from multiple hierarchies to avoid missing the big picture
Visualizing Designs

• You have seen several UML diagrams in CMPT 276 that help communicate
  – Be careful. UML is only a tool for communication. It is not design.

• 2 Common hurdles prevent visualizing & discussing design well
  – Hierarchy / Abstraction
  – Perspective

• Consider a system from multiple hierarchies to avoid missing the big picture

• **Consider both static & dynamic contexts**
Tips

- Prefer to reduce the number of boundary crossings and the # of places they happen
Tips

- Prefer to reduce the number of boundary crossings and the # of places they happen
Tips

- Prefer to reduce the number of boundary crossings and the # of places they happen

```
main()
  foo()
  bar()
  baz()

io()  io()  io()
  io()  io()  io()
```

```
main()
  read()
  foo()
  bar()
  baz()
  write()
```
Tips

- Prefer to reduce the number of boundary crossings and the # of places they happen
- **Prefer batch processing unless incrementality is required**
Tips

- Prefer to reduce the number of boundary crossings and the # of places they happen

- **Prefer batch processing unless incrementality is required**
  - Operating at Google scale can require incrementality
  - Batch processing is clearer & groups related code if you can use it
Tips

- Prefer to reduce the number of boundary crossings and the # of places they happen
- Prefer batch processing unless incrementality is required
- Prefer to keep your in-flight data immutable
Tips

- Prefer to reduce the number of boundary crossings and the # of places they happen
- Prefer batch processing unless incrementality is required
- **Prefer to keep your in-flight data immutable**
  - It is easier to see where a bad object was created than when it was corrupted
Tips

- Prefer to reduce the number of boundary crossings and the # of places they happen
- Prefer batch processing unless incrementality is required
- Prefer to keep your in-flight data immutable
- Start by following a user story through the system. Follow the data.
  - Where is data created?
  - Where is data transformed or consumed?
  - Where is new data made observable?

  All of these indicate components.
The Hidden Challenge

We have looked at many different architectural issues, but they have focused on the abstract & left something missing:
The Hidden Challenge

We have looked at many different architectural issues, but they have focused on the abstract & left something missing:

How do we decide the boundaries of a component?
Summary

- There are several architectural idioms that can be useful in creating a flexible program
Summary

- There are several architectural idioms that can be useful in creating a flexible program
- **Cleanly separating out layers & interfaces is crucial in modern designs**
Summary

- There are several architectural idioms that can be useful in creating a flexible program.
- Cleanly separating out layers & interfaces is crucial in modern designs.
- When first designing, follow the data of a user story.