

CS 398: Application Development

Week 03 Lecture: Analysis & Design 2

Architectural Patterns

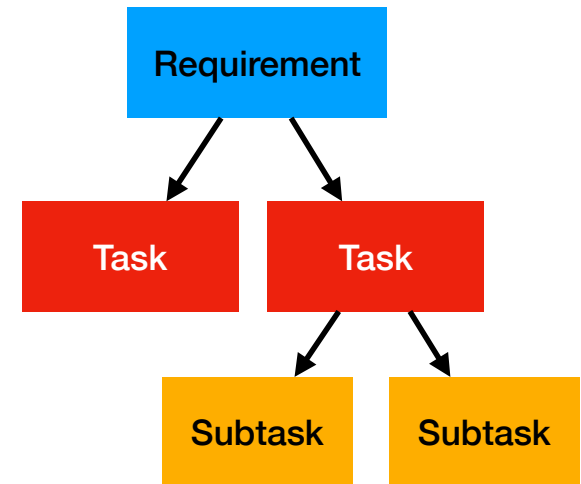
Q & A

Requirements :: Tasks

- Requirements represent functionality, or features that you wish to consider implementing. Requirements are what you **communicate** and **discuss**.
- Tasks represent the work that needs to be done to realize your requirements. Tasks are what you **do**.

e.g.

- Requirement:
 - Users should be able to sort the list of notes.
- Tasks:
 - Make the existing list sort by default.
 - Add a menu item that lets the user choose how to sort it.
 - Add a toolbar button that lets them reserve the sort order.
 - Write unit tests, etc.



What to include?



Best Practices

IntelliJ project; build scripts;
unit testing framework



Tasks include requirements, plus ALL other work that is done in a sprint.

Credit to <https://dilbert.com/strip/1995-11-17>

How to Log Requirements

- What do we log in GitLab?
- How many requirements should we have? Tasks vs. Requirements...
- How do we decide what to implement?

Sprint 1

First sprint.

Issues 4 Merge requests 0 Participants 1 Labels 2

Unstarted Issues (open and unassigned) 1	Ongoing Issues (open and assigned) 1	Completed Issues (closed) 2
Edit an existing note #5 Medium	Create a new note #4 Medium	Create the main application window #2 High
		Setup infrastructure #1 High

What can we assume about our app?

- What is specified in the “Project specification”.
 - You should have **basic functionality for your platform**.
 - If you build a desktop app, you should have menus, hotkeys, resizable window etc.
 - If you build a mobile app, you should use conventions there.
 - You decide what platform as part of your requirements.
- What about the “cloud functionality” statement?
 - I wish I hadn’t included this...
 - For now, **assume a single platform, monolithic application**.
 - Around sprint 4, we’ll pivot to store data in a service - we’ll discuss at that time.

Can we use third-party libraries?

Project Specification - <https://student.cs.uwaterloo.ca/~cs398/01-syllabus/4-project-specification/#faq>

4. **Can we use third-party libraries**, or are we supposed to build everything from scratch?

You can, and should, use libraries where appropriate - that's why they exist! They're often well-designed and tested, and using them can result in much a higher quality product (plus, they save time). You can even use third-party source code, within limits - see the course [policies](#), specifically around what constitutes [plagiarism](#). TL;dr. You can external source code as long as a single source doesn't constitute more than 10% of your project. More than that, and it's considered a potential academic integrity offense. If you have questions about this, please [just ask](#).

Course Policies - <https://student.cs.uwaterloo.ca/~cs398/01-syllabus/6-policies/#plagiarism>

Students are expected to either work on their own (in the case of quizzes), or work within a project team (for the remaining deliverables in the course). All work submitted should either be their own or created by the team for use in their project. However, we realize that it is common practice to use third-party libraries and sources found online to solve programming problems. For this reason, **the team is allowed to use third-party source or libraries for their project provided that (a) they document the source of this contribution in source code, typically as a comment, and in their README file, and (b) no single source constitutes more than 10% of their project.** Failure to acknowledge a source will result in a significant penalty (10% or more) of your final project grade, depending on the severity of the infraction. Note that MOSS will be used to compare student assignments, and that this rule also applies to copying from other student projects.

Architectural Patterns

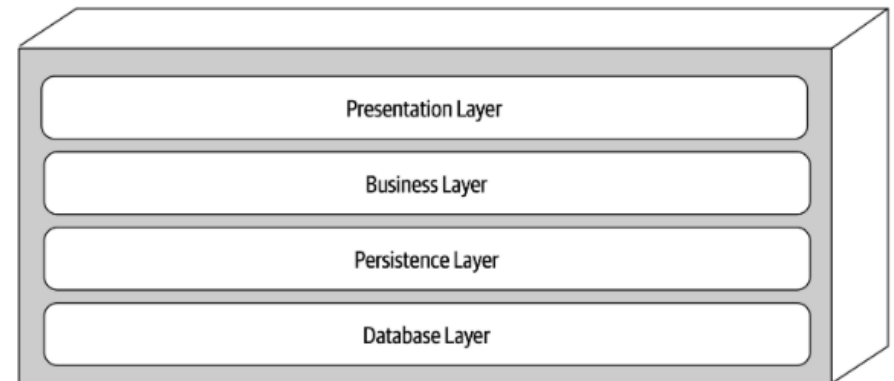
Monolithic Patterns

Layered Architecture

A layered or n-tier architecture is a very common architectural style that organizes software into horizontal layers, where each layer represents some logical functionality.

Standard layers in this style of architecture include:

- **Presentation:** UI layer that the user interacts with.
- **Business Layer:** application logic, or “business rules”.
- **Persistence Layer:** describes how to manage and save application data.
- (Optional) **Database Layer:** the underlying data store that actually stores the data.



Note: these can be logical tiers (i.e. modules in the same system)

The major characteristic of a layered architecture is that it enforces a clear **separation of concerns** between layers.

Monolithic Patterns

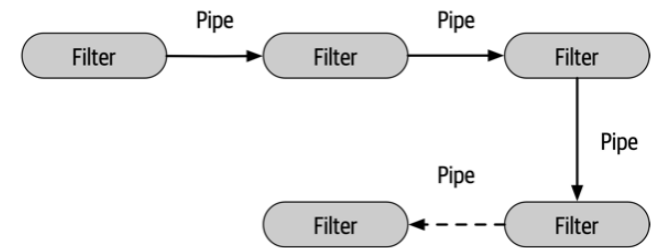
Pipeline Architecture

A pipeline (or pipes and filters) architecture is appropriate when we want to transform data in a sequential manner. It consists of pipes and filters:

Pipes form the communication channel between filters. Each pipe is unidirectional, accepting input on one end, and producing output.

Filters are entities that perform operation on data that they are fed. Each filter performs a single operation, and they are stateless. There are different types of filters:

- **Producer:** The outbound starting point (also called a source).
- **Transformer:** Accepts input, optionally transforms it, and then forwards to a filter (this resembles a *map* operation).
- **Tester:** Accepts input, optionally transforms it based on the results of a test, and then forwards to a filter (this resembles a *reduce* operation).
- **Consumer:** The termination point, where the data can be saved, displayed.



These abstractions may appear familiar, as they are used in **shell programming**. It's broadly applicable anytime you want to process data sequentially according to fixed rules.

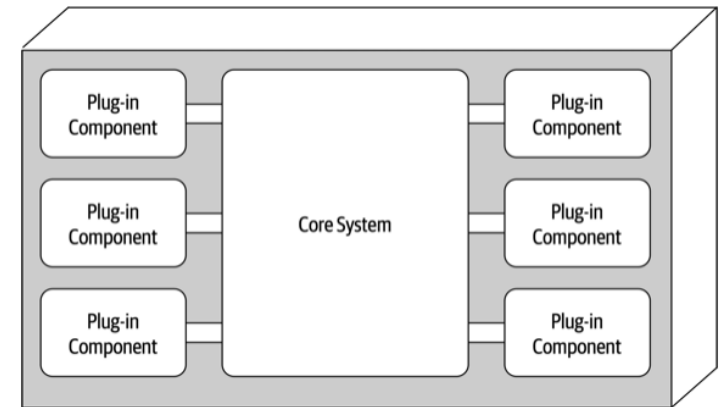
Monolithic Patterns

Microkernel Architecture

A microkernel architecture (also called plugin architecture) is a popular pattern that provides the ability to easily extend application logic to external, pluggable components.

This architecture works by focusing the primary functionality into the core system, and providing extensibility through the plugin system.

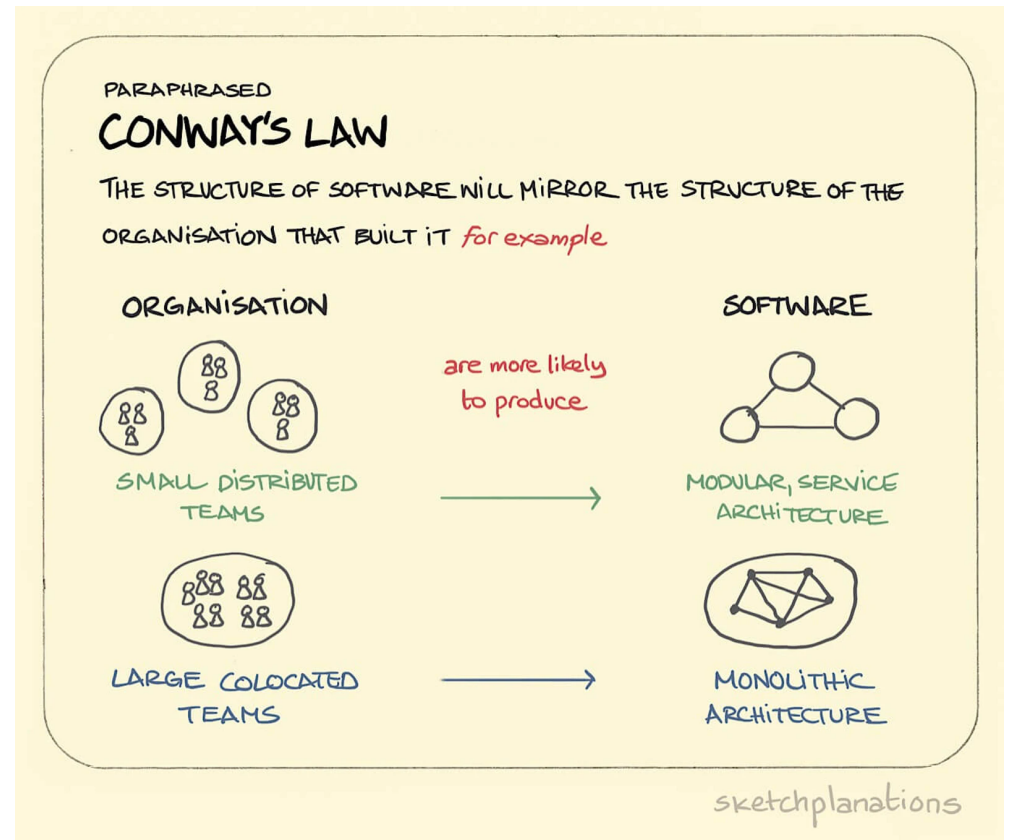
This allows the developer, for instance, to invoke functionality in a plugin when the plugin is present, using a defined interface that describes how to invoke it (without need to understand the underlying code).



Examples of this architecture include web browsers (which support extensions), and IDEA (which support plugins for various programming languages).

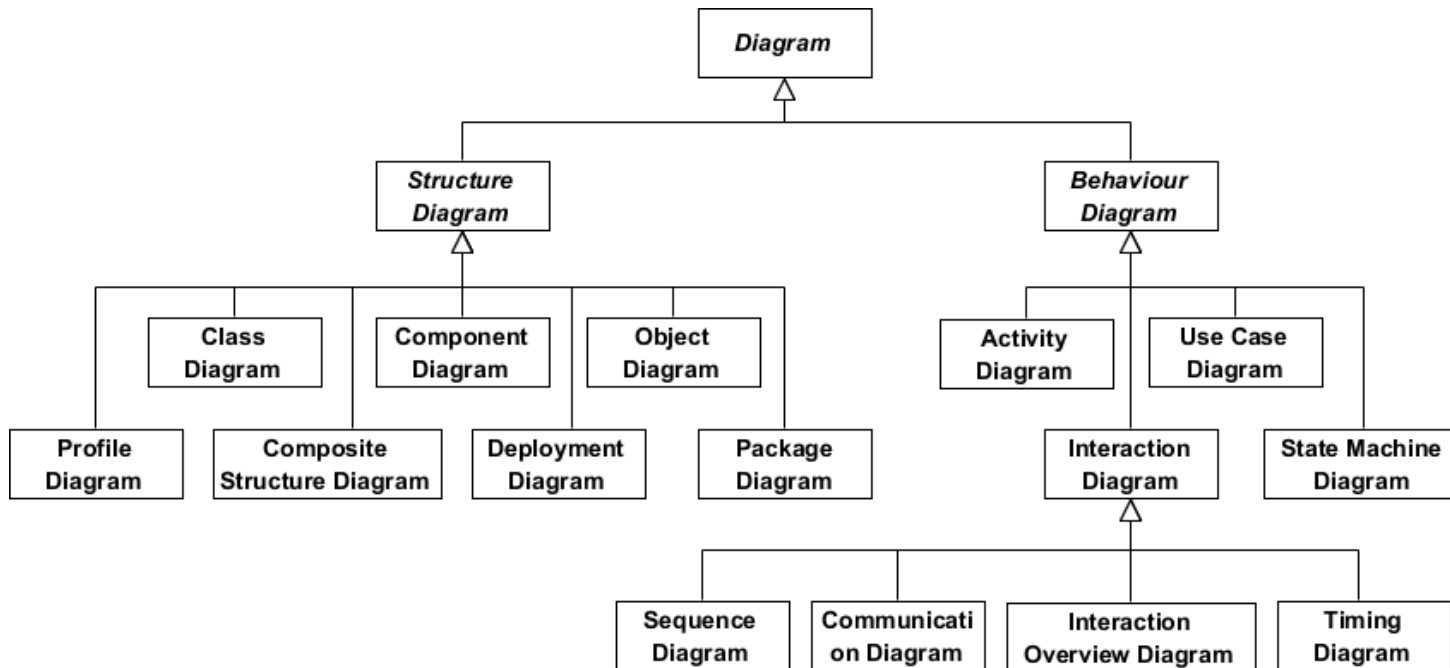
What Architectural Style to Pick?

- Look at the characteristics of the specific architectural style.
- What are the tradeoffs? There are positive and negative characteristics of each.
- Conway's Law
 - Organizations will proceed a design that mirrors their communication structure.
 - e.g. layered architecture, with a common technical database team.
 - e.g. domain focused teams building reusable business services.

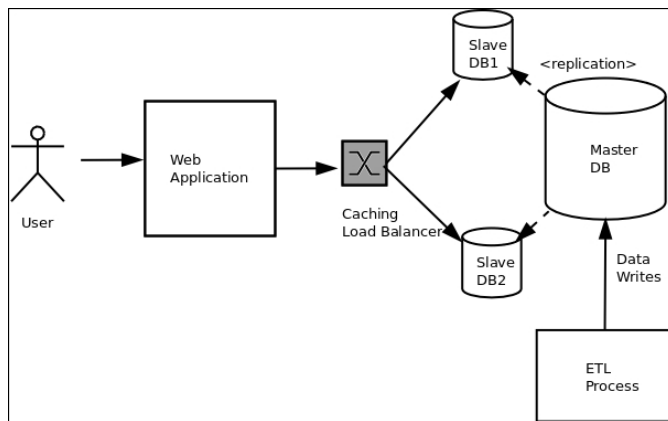


Diagramming your Architecture

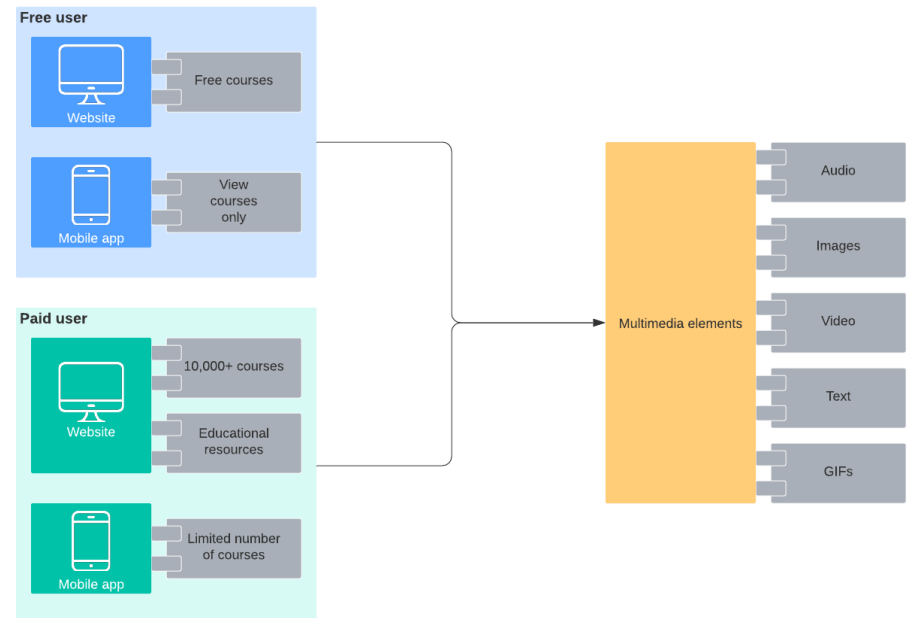
UML is a standard for diagramming systems. It contains both structure and behaviour diagrams. Consider how it might help you communicate your structure to your team, or stakeholders.



IMPORTANT: it's more important to have useful and meaningful diagrams, than to adhere to the UML standard. There are lots of examples of useful system diagrams that are less formal.



Current state







Activities

TODO Today



Planning

1. Create project plan 

Requirements

1. Pick users, (optional) create personas 
2. Interview people that fall into your role 
3. Identify requirements, (affinity diagram) 
4. Document requirements in GitLab 

Analysis & Design

1. Determine technical impact 
2. Choose architectural style 
3. System diagram

Quiz in LEARN this week! Opens now, due by Fri 11:59 PM.