Architecture

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

Defining Success

“It doesn’t take a huge amount of knowledge and skill to get a program working. Kids in high school do it all the time. The code they produce may not be pretty; but it works. Getting something to work once just isn’t that hard.

Getting software “right” is hard. When software is done right, it requires a fraction of the human resources to create and maintain. Changes are simple and rapid. Defects are few and far between. Effort is minimized, and functionality and flexibility are maximized.”

– Robert C. Martin [1]

Defining Success

“It doesn’t take a huge amount of knowledge and skill to get a program working. Kids in high school do it all the time. The code they produce may not be pretty; but it works. Getting something to work once just isn’t that hard.

Getting software “right” is hard. When software is done right, it requires a fraction of the human resources to create and maintain. Changes are simple and rapid. Defects are few and far between. Effort is minimized, and functionality and flexibility are maximized.”

– Robert C. Martin [1]

- What does it mean to “get software right”?

Defining Success

“It doesn’t take a huge amount of knowledge and skill to get a program working. Kids in high school do it all the time. The code they produce may not be pretty; but it works. Getting something to work once just isn’t that hard.

Getting software “right” is hard. When software is done right, it requires a fraction of the human resources to create and maintain. Changes are simple and rapid. Defects are few and far between. Effort is minimized, and functionality and flexibility are maximized.”

– Robert C. Martin [1]

- What does it mean to “get software right”?
- What kind of software do we want to produce?

Software Qualities

As users, we can think of features that we want in all of our software [2].

- **Effectiveness**: Our software should be “fit for purpose”
- **Reliability**: It should work as expected most (all) of the time.

As developers, we can add to this list:

- **Extensibility**: we should be able to extend existing functionality or add new functionality. The design should accommodate this.
- **Scalability**: it should be scalable to handle increased demand e.g., more users, more transactions, greater throughput.
- **Reusability**: We should reuse design/code whenever possible and design our solution in a way that fosters reuse. (However, beware YAGNI).

How do we design to accomodate these requirements?

- Theses are more than just “features in a list”.

Software Qualities

`Functional requirements` are related to the functionality of your application. These are often what users are talking about in user stories. e.g., “save a file”.

`Non-functional requirements` refer to the qualities of our software. e.g., scalability, robustness, power-consumption, speed, efficiency. These often reflect the architectural decisions that you make.

Both types of requirements can “fall out” of user scenarios or user stories.

- You can log and track them like any other requirement.

> **Tip**
>
> Non-functional requirements are often referred to indirectly by users
>
> - e.g., “Retrieving the report I need always take too long, so I keep a spreadsheet open and just copy the data myself.“”

Architecture & Design

Why It Matters

What is it?

Software architecture is the “fundamental organization of a software system, encompassing its components, their relationships, and the principles guiding its design and evolution” [2].

In other words, the architecture is the *structure* of your software: what components exist, how they are arranged and how they communicate with one another.

Why is this important?

A solid architecture ensures a system is robust, testable, and adaptable to future changes. Without proper architecture, systems often become difficult to maintain and expensive to upgrade [3]

Architectural decisions are critical decisions that help us build and maintain software successfully, over the life of the software.

Anti-Patterns

Anti-patterns are software design patterns that at-first appear to be beneficial, but result in poor outcomes. You don't want to repeat these.

Big Ball of Mud

A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. [4].

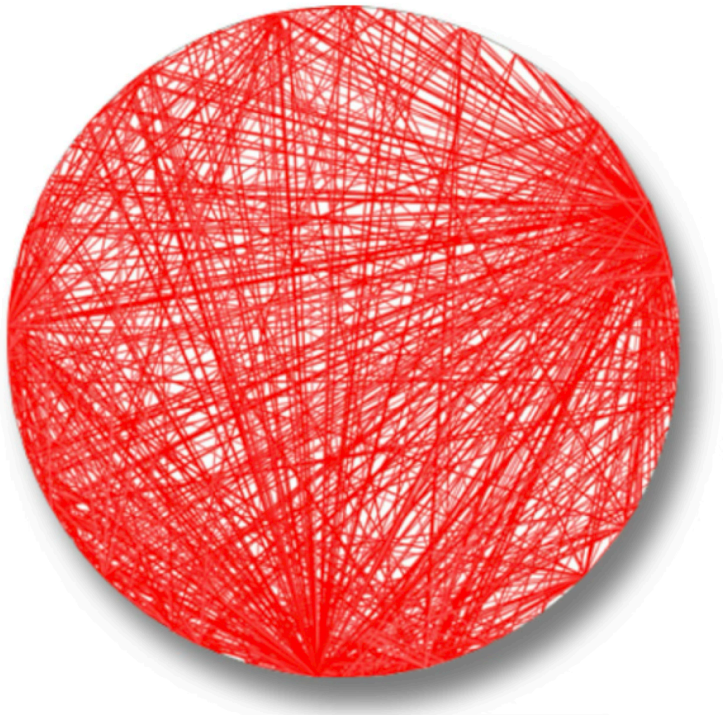


Figure 1: Big Ball of Mud is a tangled mess of function calls.

Anti-Patterns

God Objects

A God Object represents the opposite problem; we have no issues with object sharing because most functionality is contained in a *single monolithic object*!

```
class GameManager {
private:
  vector<string> players;
  int score = 0;
  bool isRunning = false;

public:
  GameManager() = default;
  ~GameManager() { /* ... */ }

  void addPlayer(const string& name) {
    players.push_back(name);
    custom_print("Added player: ");
  }

  void startGame() {
    isRunning = true;
    score = 0;
    custom_print("Game started!");
  }

  void updateGame() {
    if (isRunning) {
      score += 10;
      custom_print("Score updated: ");
    }
  }

  void endGame() {
    isRunning = false;
  }

  void draw() {
    custom_print("[Rendering]");
  }

  void handleInput(const string& input) {
    if (input == "quit") {
      endGame();
    }
  }
}
```

Architectural Styles

A Software Architecture Style refers to a high-level structural organization that defines the overall system organization, specifying how components are organized, how they interact, and the constraints on those interactions [2], [5].

Common styles include:

- Pipes and filters architecture: sequential processing of data.
- Layered architecture: tiered system, with a clear separation of concerns.
- Event-driven architecture: designed around managing events.
- Service-oriented architecture: system consists of independent services.
- Microservices architecture: smaller services that coordinate actions.

We'll discuss a few of these in detail.

Architectural Styles

Pipes-and-Filters

A pipeline (or pipes and filters) transforms data in a sequential manner. It has a number of connected components.

- **Pipes**: form the communication channel between filters. Each pipe is unidirectional, accepting input, and producing output.
- **Filters**: perform operations on data that they are fed.
  - **Producer**: The outbound starting point (also called a source).
  - **Transformer**: Accepts and transforms input, and then forwards to a filter.
  - **Tester**: Accepts input, optionally transforms it based on the results of a test, and then forwards to a filter.
- **Consumer**: The termination point, where the data can be saved.

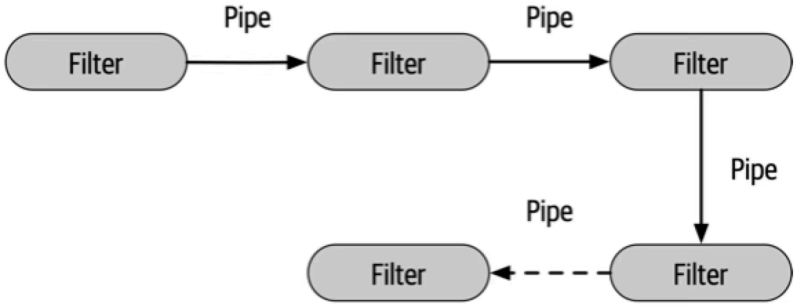


Figure 2: https://architectural-patterns.net

Architectural Styles

Microservice Architecture

A microservices architecture arranges an application as a collection of loosely coupled services, which communicate using a lightweight protocol.

You can effectively think of these services as independent applications that have specific responsibilities, and coordinate work through some communication mechanism.

Some of the defining characteristics of microservices:

- Services usually communicate over a network.
- Services are organized around business capabilities i.e. they provide specialized, domain-specific services to applications.
- Service are not tied to any one programming language or platform.
- Services are small, decentralized, and independently deployable.

Each micro-service is expected to operate independently.

Architectural Styles

Layered Architecture

A layered architecture is an *application pattern* that groups components into horizontal layers, where each layer represents a logical division of functionality [2].

The **Presentation Layer** handles the UI and all input and output.

The **Domain Layer** handles “Business Logic” i.e. functionality related to your problem domain.

The **Persistence Layer** handles saving data to a file, database or other service.

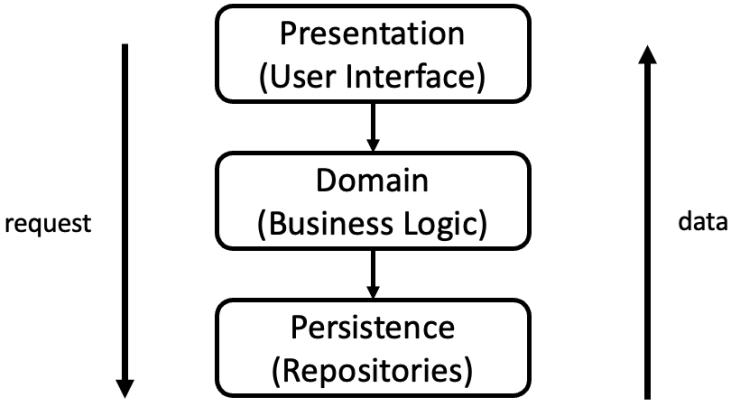


Figure 3: https://architectural-patterns.net/layers

Architectural Styles

Layering our software into distinct tiers provids a clear separation of concerns. Layers communicate with the layers immediately below them.

- Separation of concerns
  - I/O requests are always from the presentation layer
    - Requests flow top-down
    - Data flows bottom-up.
  - External communication is managed by the Persistance layer.
- We avoid circular dependencies:
  - Presentation and Persistance use the domain classes.
  - One-way or weak dependencies.

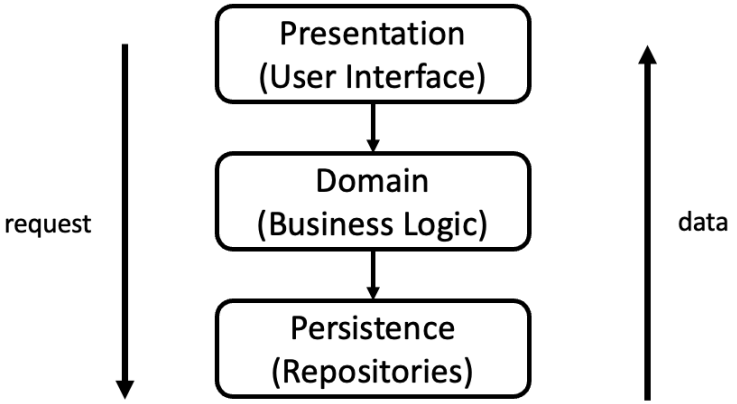


Figure 4: https://architectural-patterns.net/layers

Architectural Styles

Dependency Rule

MVC and MV-style patterns require a strict data flow model.

1. Requests flow down.
2. Data changes flow back up, usually in response to requests.

The dependencies between layers also mirror this:

- dependencies flow inward.
- layers can only communicate with the immediately adjacent layers.

> **Note**
>
> We'll focus on a layered architecture in this course, since it's a common application pattern. We'll return to the topics of data and control soon.

Design Principles

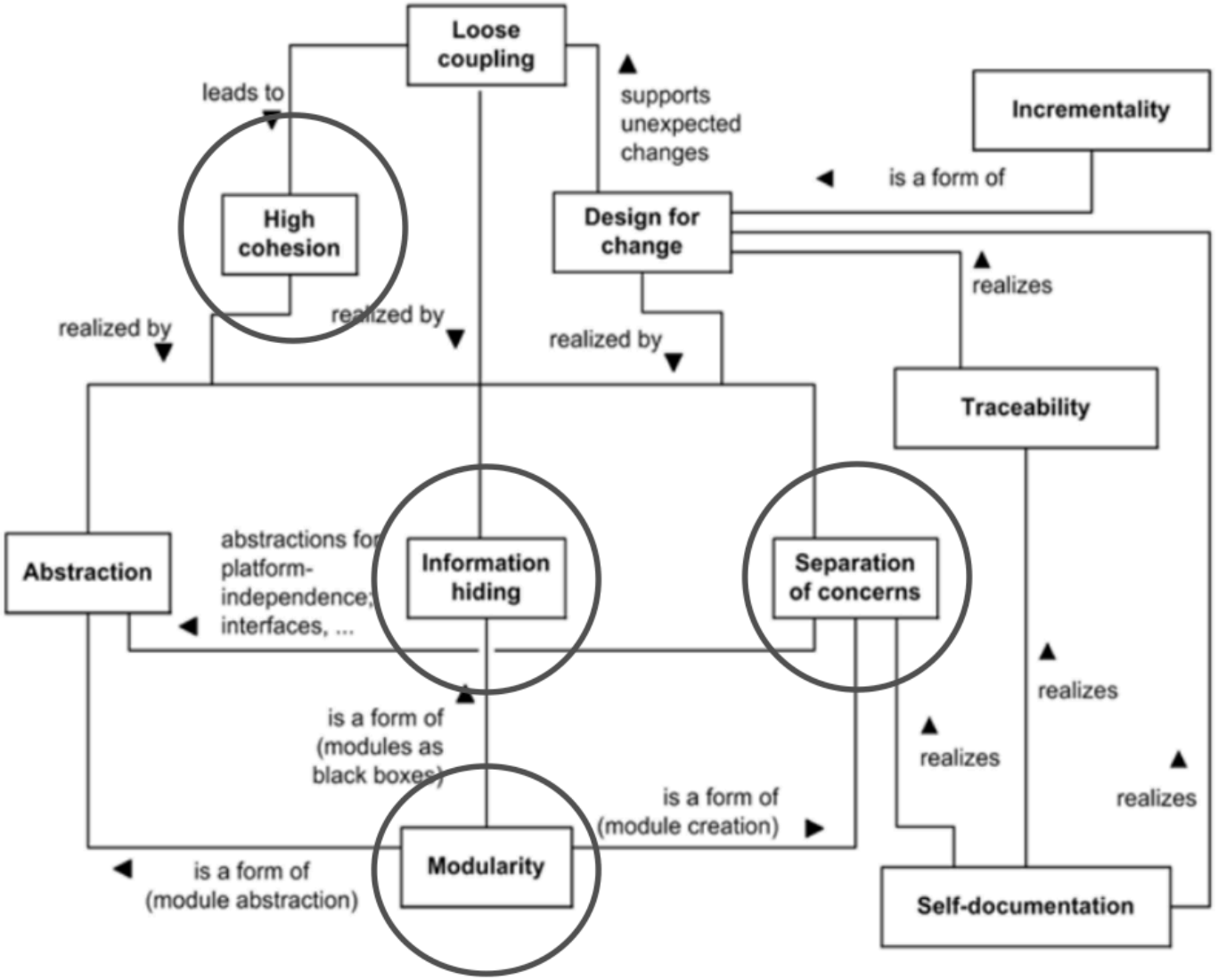


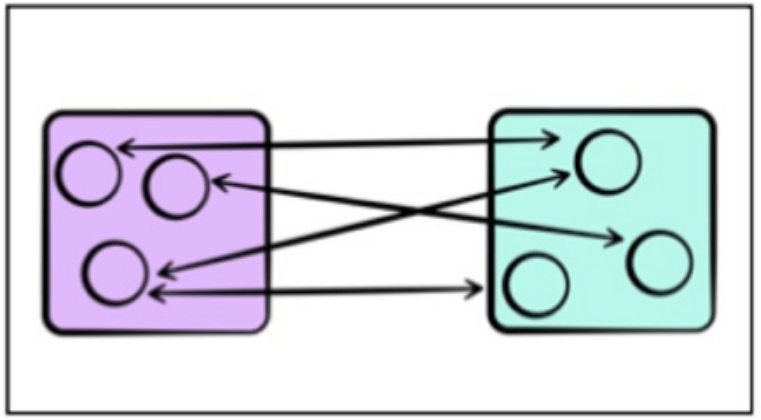
Figure 5: Principles to address flexibility, scalability and robustness.

Design Principles

Separation of Concerns

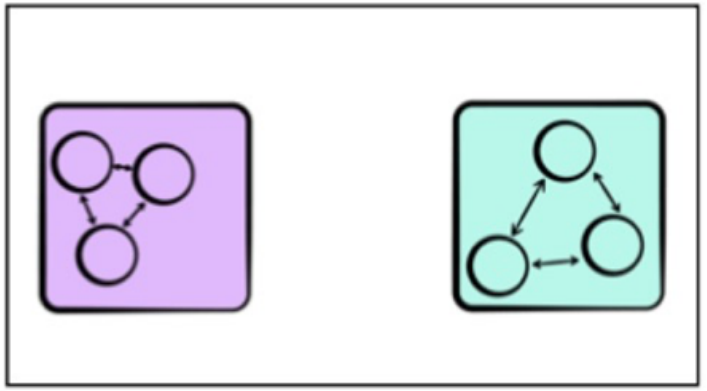
Your components should be independent of one another.

- **High cohesion**: each component is self-contained with clear responsibilities.
- **Loose coupling**: components should be self-reliant and there should be few dependencies between them.



Coupling

Coupling refers to how closely linked components or modules are to each other.



Cohesion

Cohesion is a measure of how closely related the parts of a module are.

- Clear separation leads to stable APIs. It's also easier to make changes.

Design Principles

Single Responsibility

Your architecture should consist of components.

Each component is a relatively independent software entity that implements a coherent set of functionality e.g., a class or library with a specific purpose.

Components can be classes, or functions, or module. It's "loosely defined" on purpose (but in practice often means "classes").

Benefits

- Clearly defined roles results in "cleaner" code that is easier to read, maintain and test. Remember `cohesion`.

Design Principles

Information Hiding

The internal state of a `function` | `class` | `module` should not be visible outside of that entity.

Multiple benefits

- High cohesion: entities are responsible for their own state.
- Minimized coupling: entities only communicate through well-defined interfaces.
- Avoids side-effects and helps avoid ”accidental mutation”.
- Makes entities more reusable if implementation can be changed with minimum impact on the rest of a system.

Design Principles

Modularity

Modularity refers to the logical grouping of source code into related groups. It's the high-level component structure that we want when we talk about separation of concerns e.g., namespaces in C++, packages in Java or Kotlin.

Modularity provides:

- Clear division of responsibilities, easier to manage code.
- Opportunities for code reuse e.g., exporting a distinct module.
- Improved ability to test our code!

Design Principles

Packaging

A package is a mechanism for grouping related classes, interfaces and other related code together. It's similar to a namespace in C++.

Use the `package` keywork at the top of a file to associate it to a package.

- All contents of that file are affected i.e., classes, functions.

> **Tip**
>
> Use a package name that is unique, and describes that module. The convention is: `reverse-URL (unique) + descriptive suffix.`

```
src
├── controllers
│   └── ProductController.kt
└── views
    └── CustomerView.kt
```

Suggested package names:

```
ca.uwaterloo.cs346.prj.controllers
ca.uwaterloo.cs346.prj.views
```

Design Principles

To “use” a class or function from a different package, you import it.

```
package ca.uwaterloo.cs346.prj.views
import ca.uwaterloo.cs346.prj.controllers.ProductController

class CustomerView() {
  val pc = ProductController();
  // ...
}
```

What are the benefits of doing this?

- It’s an easy way to keep layers separate e.g., `presentation` layer should not import `database` layer directly.
- It’s resilient: you can move classes around easily.

However there are drawbacks:

- It exposes the entire source tree to your developers.
- Your build system will treat this as a single module, which can be slow.

Design Principles

Modules

Alternatively, you can decompose your application into sub-modules.

For example, you could create separate modules for `presentation`, `data` and `domain` layers of your application and let the build system manage it.

There are benefits:

- It enforces dependencies, since your have to define them for the build configuration.
- Your build system can build each separately which is faster.
- You can only expose modules that you want public.

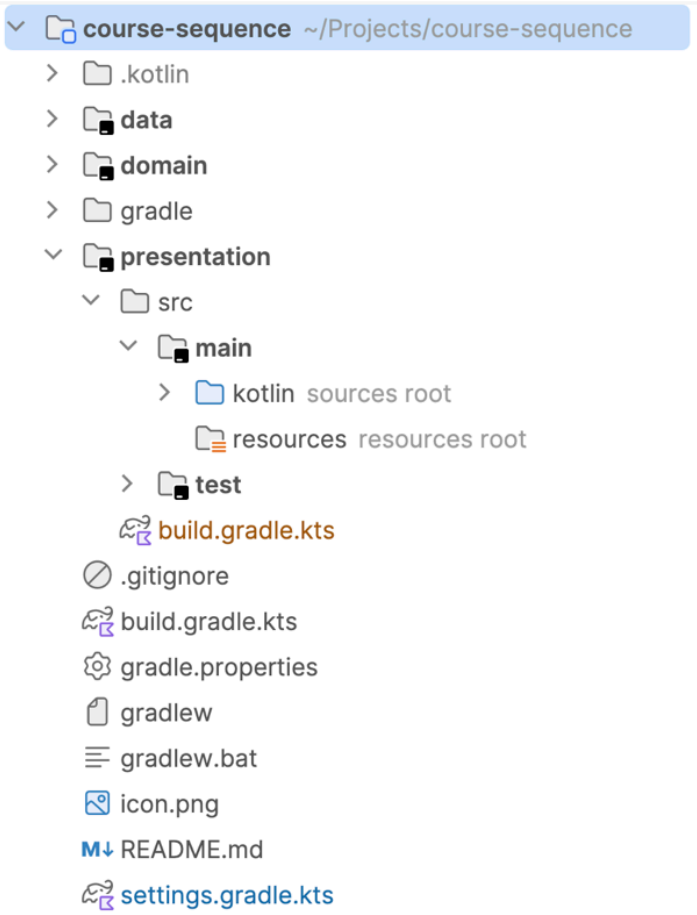


Figure 6: Multi-module projects are useful for larger code-bases.

Design Principles

Modularity has other benefits as well.

It can help is maintain independence from third-party libraries or frameworks. By keeping these entities firmly separate from our source code, they become easier to test (and/or replace).

We'll see this when we discuss the presentation and persistance layers - the "edges" of our system.

Bibliography

[1] R. C. Martin, *Clean Architecture - A Craftsman's Guide to Software Structure and Design*. Pearson, 2017.

[2] M. Richards and N. Ford, *Fundamentals of Software Architecture A Modern Engineering Approach*, 2nd ed. O'Reilly Media, 2009.

[3] M. Fowler, "Software Architecture Guide." [Online]. Available: https://martinfowler.com/architecture

[4] B. Foote and J. Yoder, "Big Ball of Mud," in *Pattern Languages of Program Design*, Addison-Wesley Professional, 1997, pp. 654–692.

[5] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.