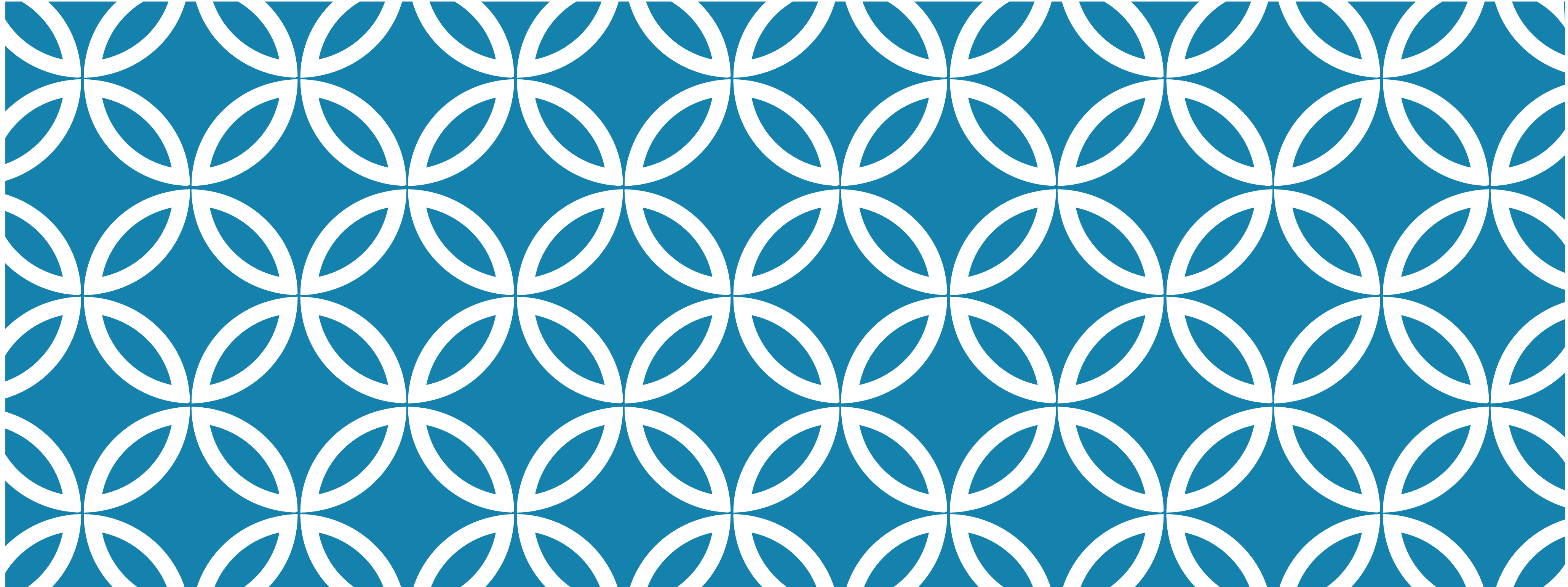
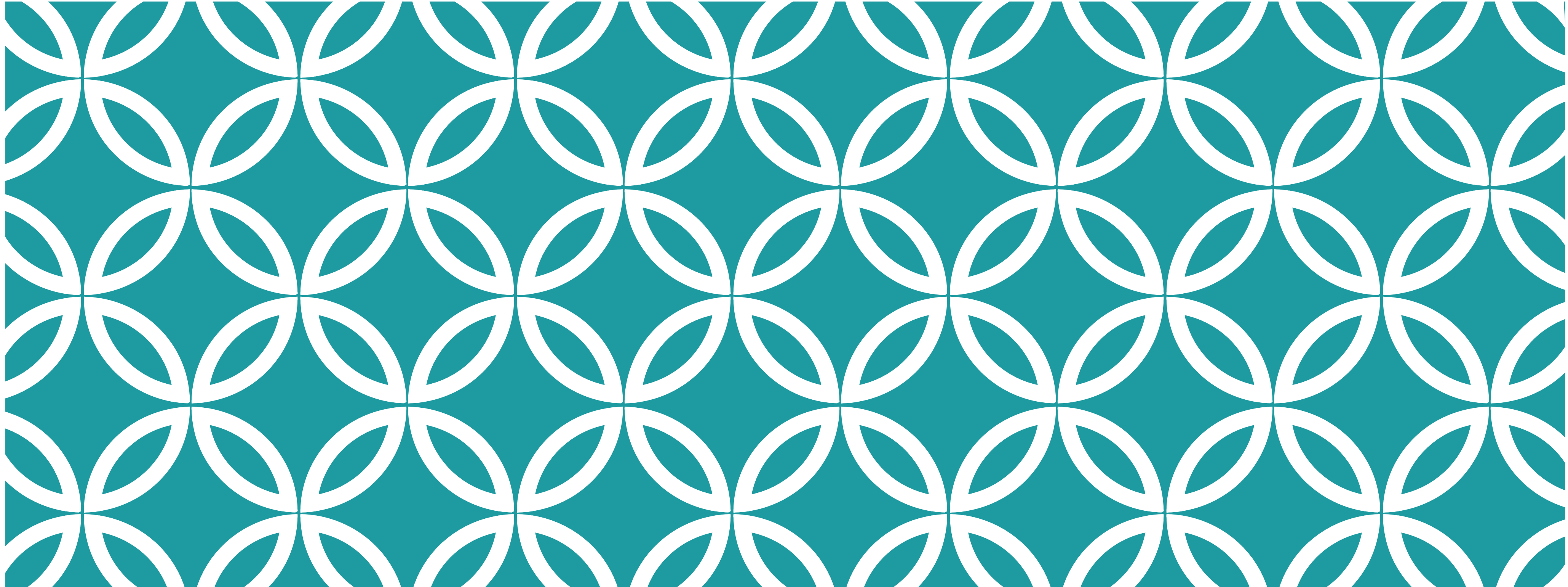


WELCOME TO CS 346!

CS 346: Application Development



OUTLINE

CS 346: Application Development

INTRODUCTIONS

Dr. Jeffery Avery

Associate Professor, Teaching Stream

Cheriton School of Computer Science

Caroline Kierstead

Instructional Support Coordinator

Teaching Assistants

Yiwen Dong, Favour Kio, Aniruddhan Murali , Gareema Ranjan, Hauton Tsang, Amber Wang

https://student.cs.uwaterloo.ca/~cs346/1249/course-outline/contact-us/

aka “Jeff” or Prof Avery
please not Jeffery

WHAT IS THIS COURSE ABOUT?

CS 346 Application Development
LAB, LEC, TST 0.50

Introduction to **full-stack application design and development**. Students will work in **project teams** to design and build complete, working applications and services using standard tools. Topics include **best practices** in design, development, testing, and deployment.

Prerequisites: CS 246; Computer Science students only.

https://student.cs.uwaterloo.ca/~cs346/1249/

WHAT WILL YOU DO?

You will design + build an application!

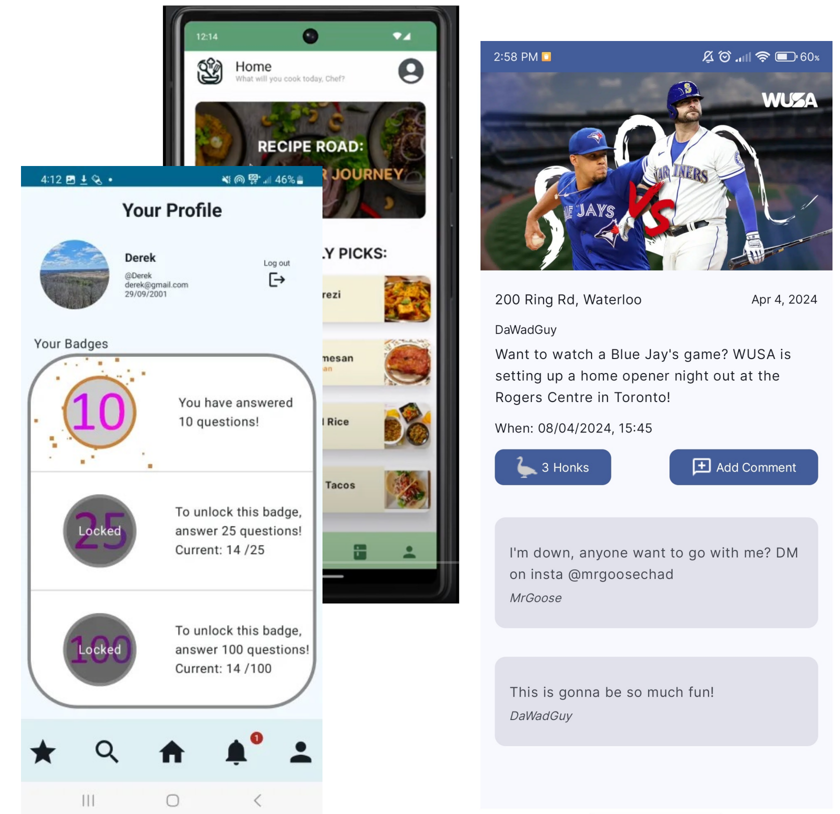
Teams of 4 people

Produce a well-designed, robust application

- You choose what application to build!
- Mobile or desktop using our technology stack.
- Basic requirements (e.g., graphical, saves data, uses cloud services).
- You and your team pick suitable advanced features.

Bi-weekly releases

- Demo to your TA and submit product releases.



https://student.cs.uwaterloo.ca/~cs346/1249/course-project/gallery/

WHAT WILL YOU LEARN?

Iterative, team-based development

- Work on a project team, where you need to collaborate and coordinate work.
- Learn an interesting and useful tech stack, with a modern programming language.
- Learn mobile development, graphical user interfaces, database connectivity.
- Apply relevant design practices i.e., design principles, patterns.

Best practices

- Build software the way that you would in industry. This includes software development practices. e.g., code branching/merges, issue tracking, unit testing, software releases.
- Just like real-life, you will demo your progress!

Teamwork

- Practice communication, teamwork, collaboration skills.
- It will be ~~fun~~ ~~challenging~~ ~~frustrating~~ rewarding!

COURSE WEBSITE (LINK)

CS 346 F24

Schedule Blog Teams GitLab Piazza Learn

Filter

- Home
- Course Outline
  - Learning Objectives
  - Course Structure
  - Assessment
  - Course Policies
  - Required Resources
  - Contacting Us
- Getting Started
- Teamwork

CS 346 Application Development

Course Description

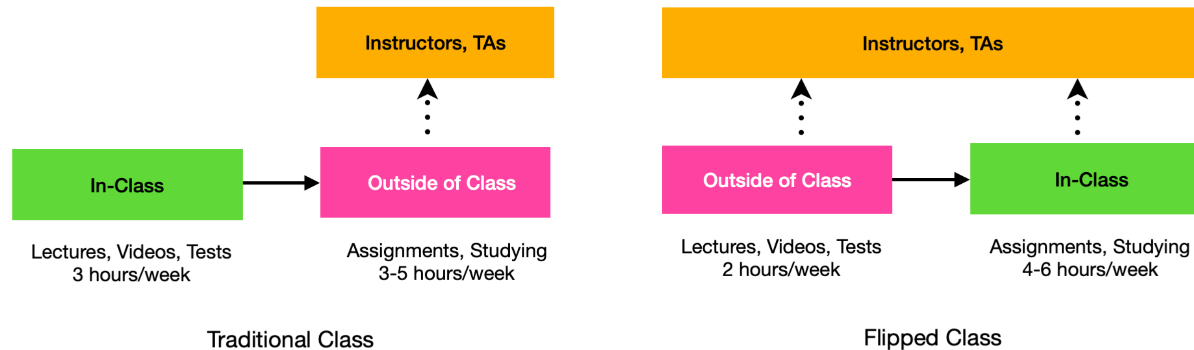
CS 346 Application Development
LAB, LEC, TST 0.50

Introduction to full-stack application design and development. Students will work
applications and services using standard tools. Topics include best practices in

Prerequisites: CS 246; Computer Science students only.

https://student.cs.uwaterloo.ca/~cs346/1249/

COURSE STRUCTURE (LINK)



Mon: The instructor will post videos & slides on Mon morning, which you should review before Wed.

Wed (In-class): The instructor will review, show demos etc. Most of the class is time for your project.

Fri (In-class): Friday is dedicated to working on your project. Instructor + TAs will be present.

SCHEDULE (LINK)

Schedule | Blog | Teams | GitLab | Piazza | Learn

Things we'll discuss/demo in-class (not testable)

Everything due end of the week!

Link to weekly agenda

Videos and slides under weekly agenda (testable)

Quizzes and project components due

Week	Lecture Posted (Mon online)	In-Class Activities (Wed & Fri)	Due (Fri 11:59 PM)
Week 1: Sept 3-6	-	■ Introduction ■ Teamwork ■ Development Process ■ Setup GitLab Project	-
Week 2: Sept 9-13	Kotlin, Gradle	■ Install toolchain ■ Create a Gradle project	■ W2 Quiz ■ M1: Project Setup

WEEKLY AGENDA (LINK)

Blog

BLOG

Week 01 - Introduction

This first week, we will have lectures on both Wed and Fri; you can expect this material to be spread across both days.

2024-09-03

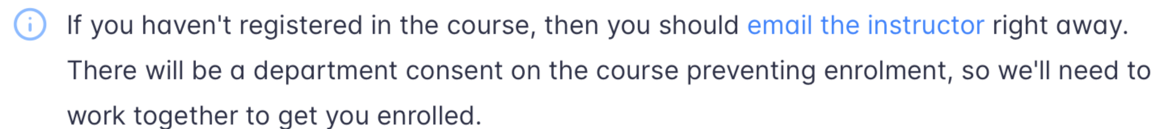
Link to weekly agenda

WEEKLY AGENDA (LINK)

Week 01 - Introduction

Welcome to the course!

This first week, we will have lectures on both Wed and Fri; you can expect this material to be spread across both days.


If you haven't registered in the course, then you should email the instructor right away. There will be a department consent on the course preventing enrolment, so we'll need to work together to get you enrolled.

Topics

Also has links to videos, slides each week.

Posted Mon 10 AM.

ASSESSMENT (LINK)

Individual

- Quizzes (weeks 2-11): 10 x 2% = 20%
- Weekly attendance (weeks 2-13): 12 weeks x 1% = 12%   // I'll pass around an attendance sheet

Team Project

- M1. Project setup: 2%   // see schedule; something due approximately every 2 weeks
- M2. Project proposal: 5%
- M3. Design proposal: 6%
- M4-M6. Product releases (including demo): 3 x 10% = 30%
- M7. Final release (release + documentation): 25%

POLICIES (LINK)

Grading

- Quizzes are auto-graded and returned the next Monday.
- Everything else is graded and returned ~1 week after submission. Grades in Learn.
- Regrade policy: 1week from the time they are returned.

Group Participation

- You must form teams by the end of week 2.
- We will help, but we reserve the right to remove you if you don't cooperate.
- You must participate during the term! In unusual circumstances, we may adjust grades downward if you do not adequately contribute.

Has not been used previously.

Code "Sharing"

- You are allowed to share code (up to 25 lines) with appropriate citation. No AI/LLM!

WEEK 01: WHAT TO DO?

Contact me after class if you

1. Aren't registered yet.
2. Need to switch sections.
3. Have any other questions!

Attend lectures

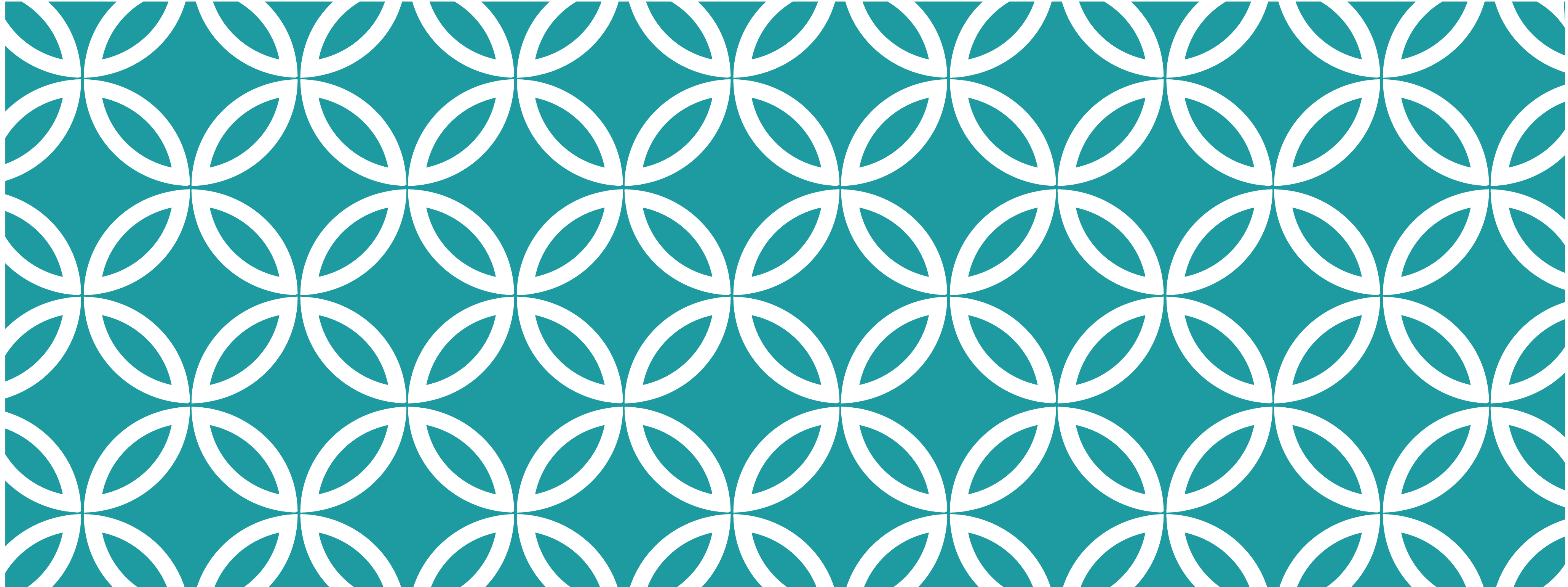
- Wed: Introduction; Software Process
- Fri: Teamwork; Development Best-Practices.

Register for the course!

- Talk to the instructor (i.e., me) if you are not registered.
- You must be in all morning or all afternoon sections; you cannot mix/match.
- You must attend in-person. You may also be removed if you fail to participate.

Form teams

- Four people
- All team members must be in the same sections. Talk to me if you need to switch sections.
- A mix of skills and interests is desirable! There's lots to do.



WHAT IS AN APPLICATION?

CS 346: Application Development

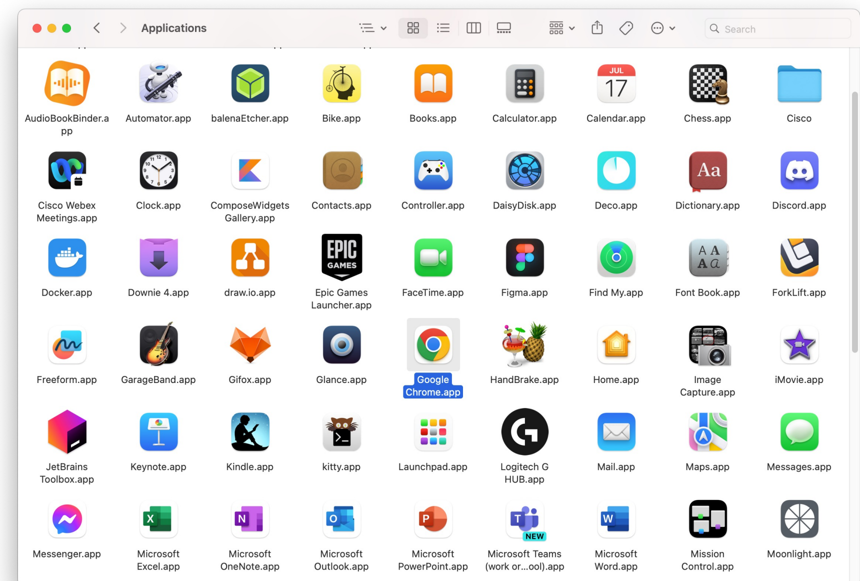
WHAT IS AN APPLICATION?

An application is “software designed to solve a problem for users”.

- Tends to be task-oriented.
- Term covers mobile, desktop, web apps.
- Distinct from system software, which is software that provides services for other software e.g., drivers.

What is a full-stack application?

- Division of application into front-end (user interface) and back-end (part that processes data).
- Doesn’t really apply to all applications (but applies to your project!)



1. COMMAND-LINE

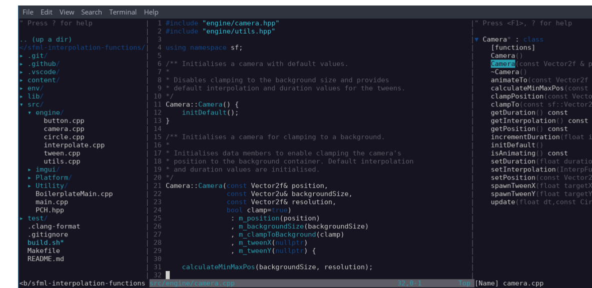
The first interactive applications were text-based and designed to be executed from a terminal or console. e.g., ls, grep, vim.

Characteristics of command-line applications:

- **Keyboard driven.** Not intended to support mice, trackpads.
- **Non-graphical**. Intended to be run from a shell/terminal, with limited character graphics.
- **Standard text I/O**. Read/write from the terminal, or file system.
- **Scriptable**. Very suitable for automation or chaining tasks.

User pro/con:

- Efficient. Small executables.
- Targeted at experts. Usually a high learning curve.
- Not discoverable. Not graphical.



2. DESKTOP

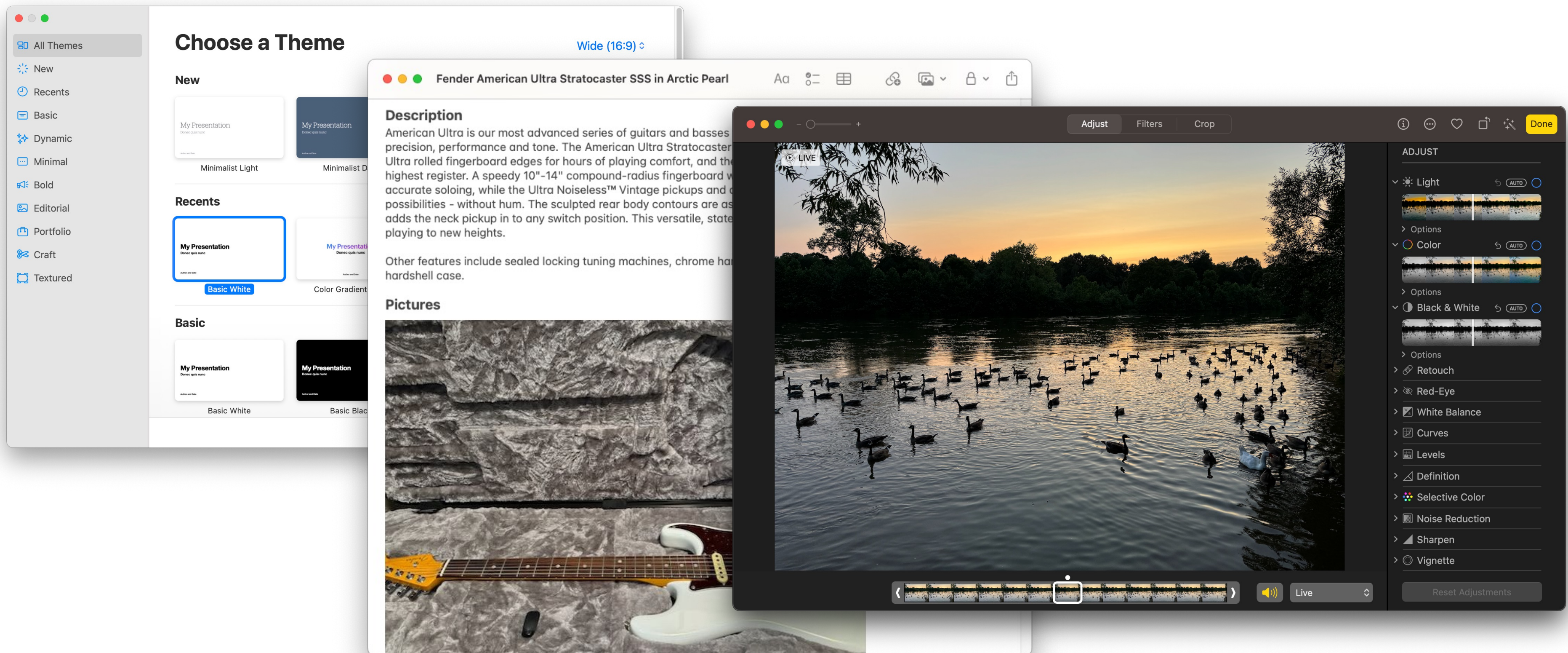
Desktop applications run on a desktop operating system e.g. Windows or macOS.

Characteristics of desktop applications include:

- **Keyboard and mouse** are used for input. Possibly other pointing devices e.g., trackpad.
- **Windows on a desktop** is the primary abstraction. Application input/output is contained within a window, which can be moved, resized, and stacked on one another.
- **Multiple-window support**. Applications may have multiple windows, each with their own content.
- **Multi-tasking** by design. Multiple windows supports multiple applications running side-by-side.
- **Rich user experience**. Applications can use graphics, animations, and other visual effects to make the user experience more engaging.

User pro/con:

- Handles richer input and output.
- Features are more ”discoverable” for new users.



Desktop applications are suitable for tasks that require a lot of screen real-estate, or that need to be used for long periods of time. They also support mouse/trackpad and other input devices which makes them suitable for precise input as well.

3. MOBILE

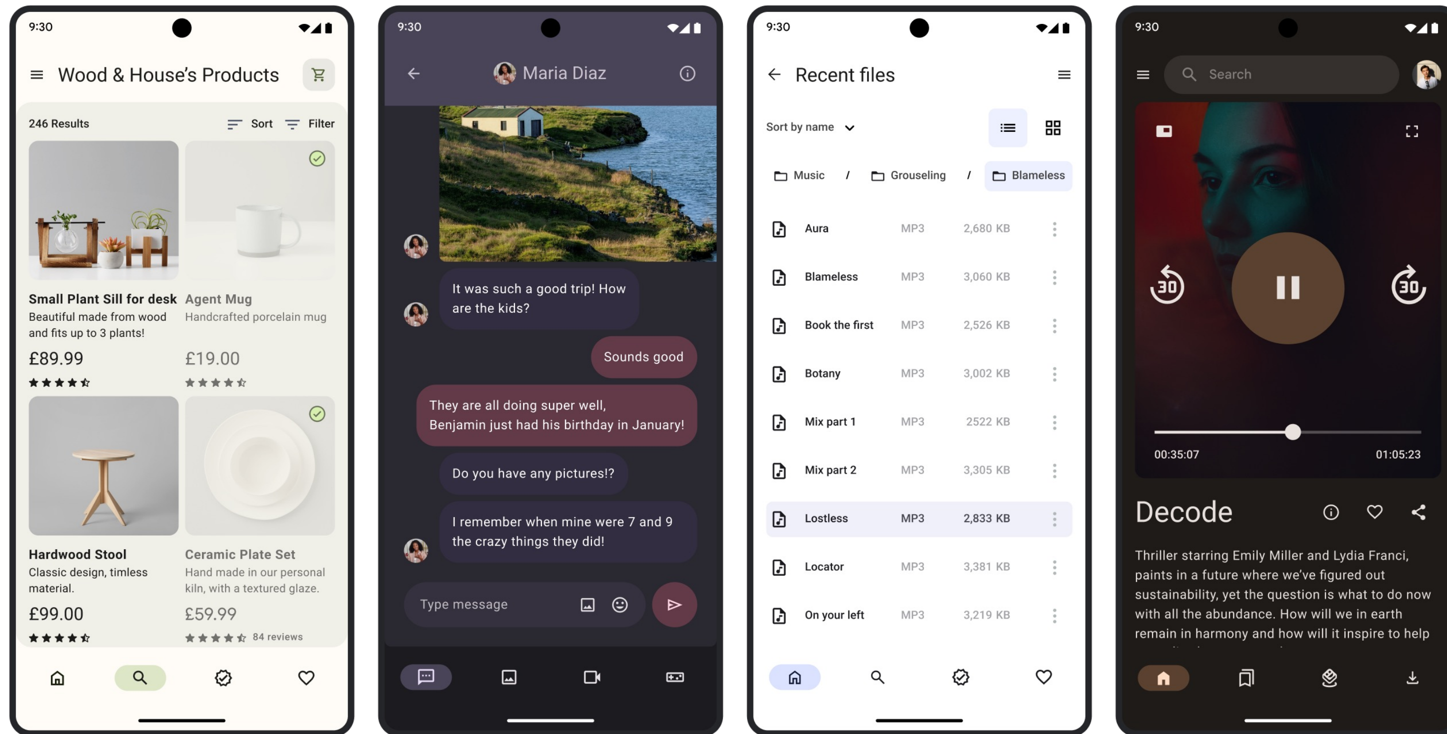
Mobile applications are designed for use on a smartphone, tablet or similar device.

Characteristics of mobile applications include:

- **Touch-enabled**. Users interact using gestures e.g., tapping, swiping, and pinching.
- **Small-screen enabled**. Applications are designed to be used on a small screen and are often optimized for portrait mode (due to hand orientation).
- **Single-tasking**. Due to the screen size, applications tend to be single tasking and are not designed for long periods of use.
- **Rich-user experience**. Like desktop applications, mobile applications can use graphics, animations, and other visual effects to make the user experience more engaging.

User pro/con:

- **Lack of precision**. Small devices and touch-interfaces make precision interaction difficult.
- **Small screen**. Smaller interaction surface is less suitable for "large" output. See desktop.



Mobile applications are usually designed for casual, on-the-go use. They also tend to favor content consumption vs. creation, where touch-input isn't a significant restriction. Otherwise, they are functionally very similar to desktop applications.

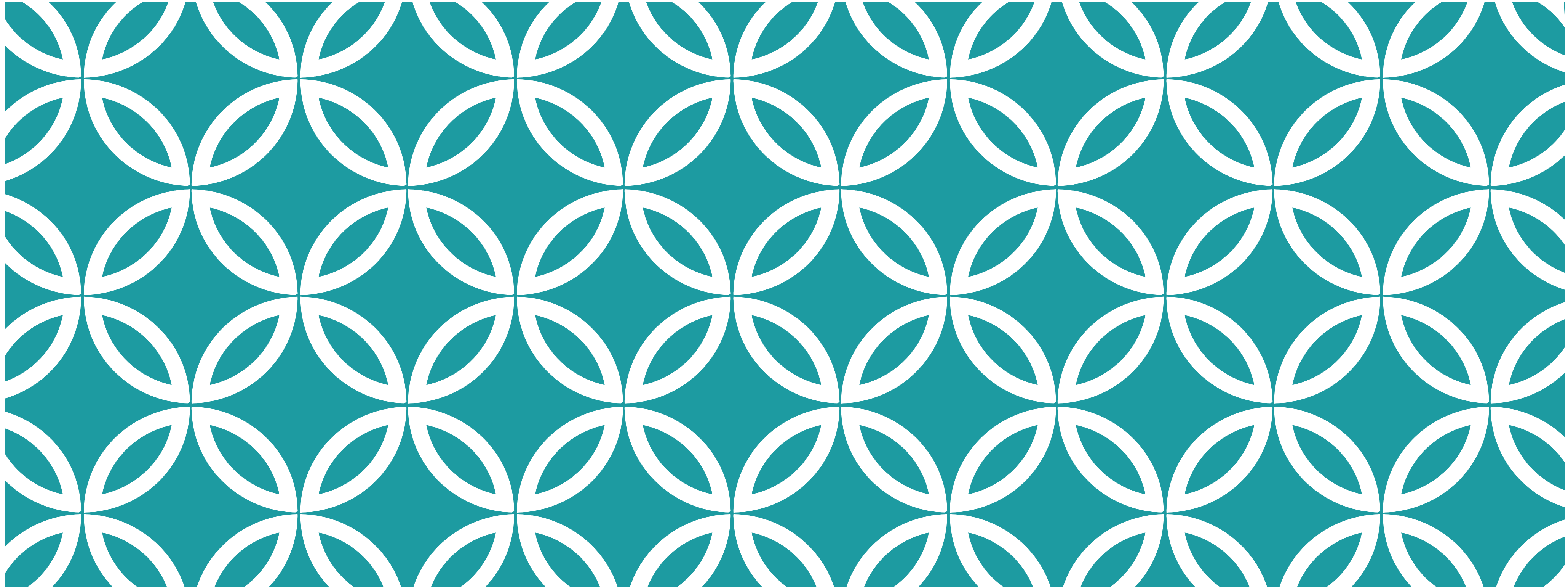
YOUR PROJECT?

Can be any of:

- Console application (with some graphics and rich functionality)
- Desktop application (standard features e.g., cut/copy/paste and undo/redo)
- Android application (standard features, e.g., device orientation)

Tip

- Pick what interests you!
- Pick an application that suits this style of application.



DEVELOPMENT THEMES

CS 346: Application Development

WE WANT TO BUILD SOFTWARE "CORRECTLY"

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. It works because **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, Clean Architecture (2016).

Recognize that we requirements and our software **will change over time**.

- We will need to add new features or extend existing features.
- We will inevitably have bugs and defects to address.

GOAL: ROBUSTNESS

Software rarely works in a vacuum.

- Your operating environment may change (OS, libr
- You are probably getting inputs from many sourc Sometimes they are in a format you don't expect.

These things can result in expected behaviour.

Robustness means that your software needs t faced with unintended inputs, or changes to th

- It cannot crash. Ever. Manage faults and exceptio
- Performance should not degrade over time.
- Data should never, ever get lost.

BUSINESS

Delta's CEO says the CrowdStrike outage cost the airline $500 million in 5 days

JULY 31, 2024 · 12:38 PM ET

By Jason Breslow

A Delta Air Lines jet takes off at the Los Angeles International Airport in April. Ed Bastian, the airline's CEO, says the CrowdStrike outage has cost the carrier $500 million.

Damian Dovarganes/AP

Robust means this doesn't happen. Ever.

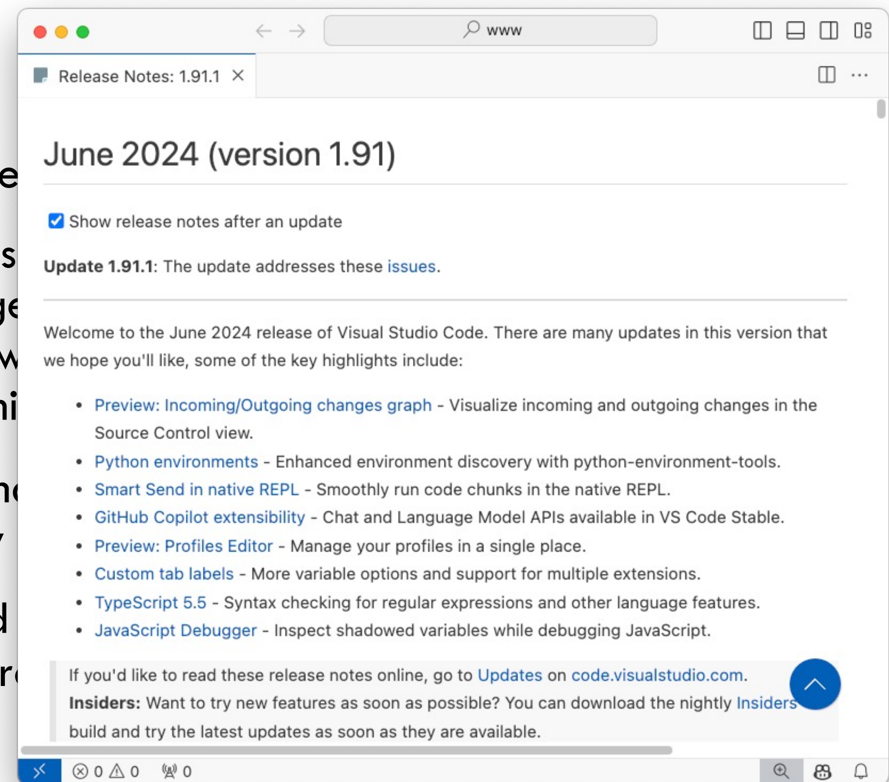
GOAL: FLEXIBILITY

Our needs change over time, and software ne

Flexibility implies that you can make changes the opposite of “brittle code”). This also sugge breaking changes, like adding drastically new adding support for code fences and syntax hi

Extensibility implies the ability to expand an image file format is released, and you easily

These properties are enabled by careful and application properly, you can anticipate futur



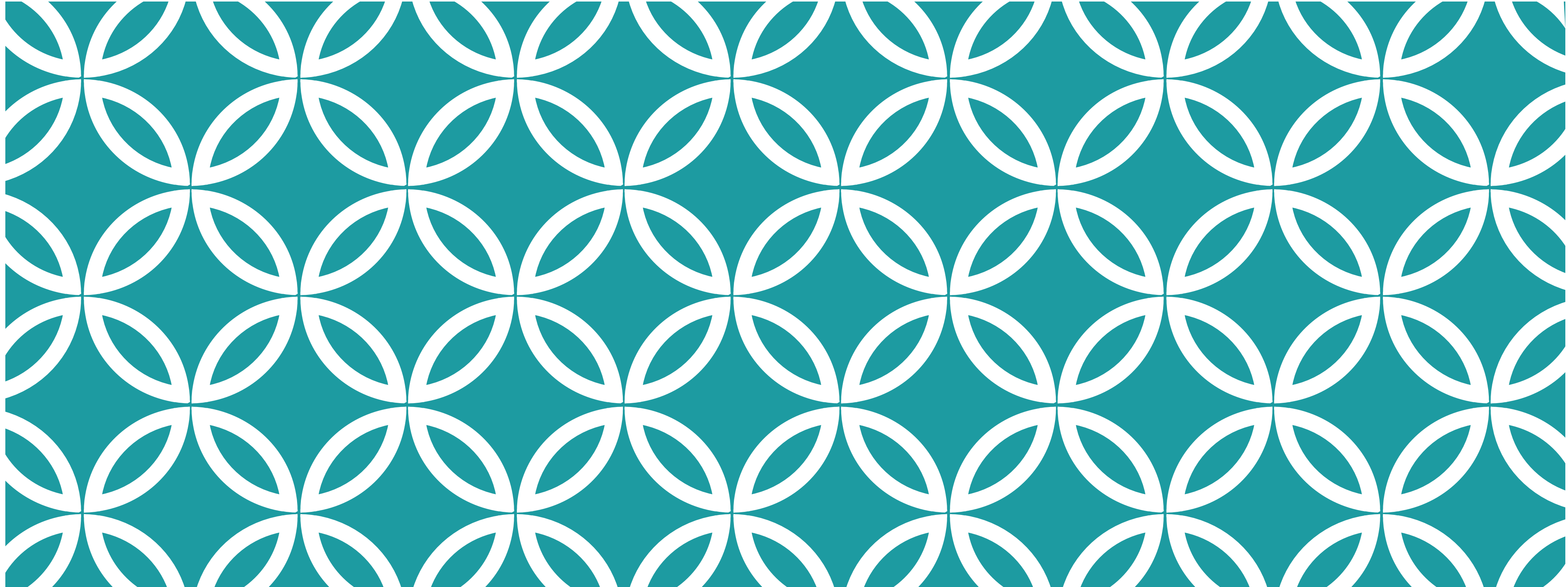
GOAL: REUSABILITY

Software is expensive and time-consuming to produce. Code reusability is literally reusing code that you've written across more than one project.

Code reusability helps to reduce cost and time to delivery. **It also reduces risks,** since you are reusing well-tested code, instead of writing new and potentially defective code.

There are different levels of reuse.

- At the lowest level, you reuse **classes**: class libraries, containers, maybe some class "teams" like container/iterator.
- At the highest level, you have **frameworks.** They identify the key abstractions for solving a problem, represent them by classes and define relationships between them.
- There also is a middle level. This is where I see patterns: **design patterns** are both smaller and more abstract than frameworks. They're really a description about how a couple of classes can relate to and interact with each other (Eric Gamma, 2005)



MODERN DEVELOPMENT

CS 346: Application Development

PICKING A PLATFORM

What should we develop – desktop applications or mobile applications?

1. Your first consideration should be **the user experience**. What application style is most suitable for your user and their requirements?

2. Consider the market share of the **platform** (desktop vs mobile). Mobile is arguably a more common platform i.e., "everyone has a phone" but desktop is still very important. Consider reach (# of users) vs. profitability of a platform.

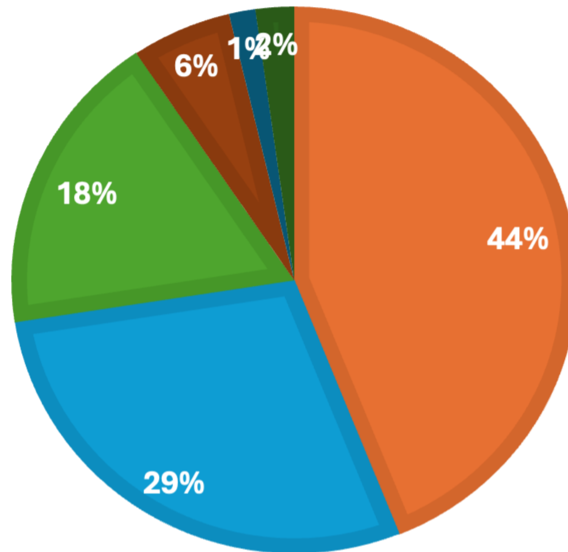
3. Finally, consider the market share of the **operating system** for your specific scenario. Ultimately you need to support the platform + OS combination that works.

MARKET SHARE?

80.0%
70.0%
60.0%
50.0%
40.0%
30.0%
20.0%
10.0%
0.0%

COMBINED MARKET SHARE

Android Windows iOS macOS Linux Unknown



Command-line applications account for < 1% of commercial software, which is why they aren't included here. Similarly for watches and other wearables.

What's the most popular software platform in the world?

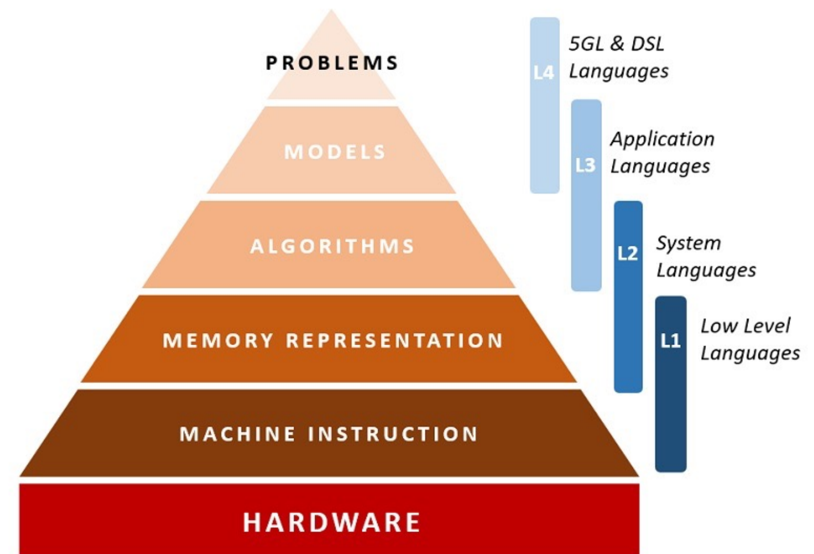
PICK A PROGRAMMING LANGUAGE

Your choice of programming language needs to align with the software that you are developing.

We can (simplistically) divide programming languages into two categories: **low-level** and **high-level** languages.

- Low-level: "close to the metal", providing the most control over how your code executes.
- High-level: "more abstract", exercising less control over code execution.

Early assumptions in programming languages is that we were move away from low-level languages, and this has *sort-of* happened…



PROGRAMMING LANGUAGES

Low-level languages are suitable when you are concerned with the performance of your software, and when you need to control memory usage and other low-level resources.

- They are often most suitable for **systems programming** tasks, that are concerned with delivering code that runs as fast as possible and uses as little memory as possible.
- Examples of systems languages include: C, C++, and Rust.
- Appropriate domains include: operating systems, device drivers, and game engines.

High-level languages are suitable when you are concerned with the speed of development, extensibility, and the robustness of your solution.

- **Applications programming** leans heavily on high-level languages, making some performance concessions for more expressive programming models, and programming language features.
- Examples of application languages include Swift, Kotlin, Go, and Dart.
- Appropriate domains include: : web applications, mobile/desktop applications, and servers/services.

WHAT DOES A HIGH-LEVEL LANGUAGE PROVIDE?

- **Automatic memory management**
  - Automatic memory allocation/deallocation (via ref-counting, or GC). This eliminates the risk of accessing uninitialized memory.
- **Type inference**
  - Type inference across the type system.
- **NULL safety**
  - A type system that prevents NULL errors.
- **Concurrency**
  - More control over how async code executes.
- **Broader programming models**
  - Mix of functional, object-oriented paradigms.

Features	Low-level	High-level
Memory management	manual	garbage collection
Type system	static	static or dynamic
Runtime	fast	medium
Executables	small, fast	large, slow
Portability	low	high

These features contribute to our goal of building robust, extensible, reusable software.

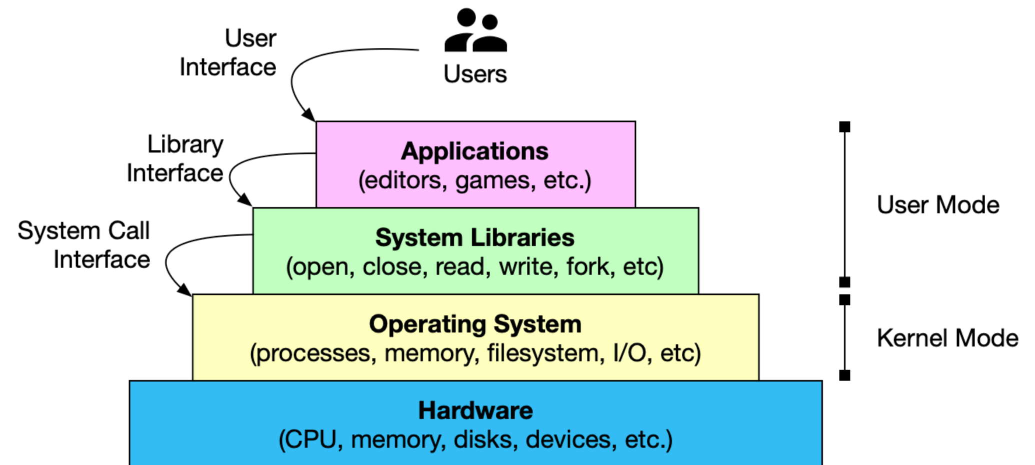
OTHER FUNCTIONALITY? CALL THE OS

System Call interface

- OS specific.
- Using this interface in your application ties you to that OS.

System Libraries interface

- System libraries provide a more generic interface, at a higher level of abstraction.
- Programming languages tend to implement standard libraries at this level. e.g., C++ standard library.
- Other "user libraries" exist in "user space", for security/stability.



LIBRARIES

You could, in theory, access a lot of your operating system functionality through syscalls, but this would be highly platform-specific code, that would not be portable across operating system. Instead we use libraries, which tend to abstract away small OS and hardware differences.

Library origin		
Programming language	stdlib, stdio	Libraries included in the programming language; guaranteed to exist everywhere
OS vendor	Win32, Cocoa	Libraries/Frameworks provided by the vendor to support syscalls/low-level access. Includes common functionality like graphics, networking.
Third-party	OpenGL, OpenCV	Libaries provided by other parties to allow additional support beyond what the OS vendors provide. e.g., database access, computer vision.

APPLICATION PROGRAMMING

A **technology stack** is the set of related technologies that you use to develop and deliver applications. We need a **programming language + libraries** that work together on your platforms.

Platform	OS	Programming Languages	Libraries?
Desktop	Windows (Microsoft)	C#	.NET, UWP (Maui)
	macOS (Apple)	Swift, Objective-C	Cocoa, UIKit, SwiftUI
	Linux (Many)	C, C++	GTK
Mobile	Android (Google)	Kotlin	Android SDK, Compose
	iOS (Apple)	Swift, Objective-C	UIKit, SwiftUI

^ Languages & Libraries are often vendor-specific !

HOW DO WE SUPPORT MULTIPLE PLATFORMS?

1. **Build a native application for each platform**. Use Microsoft's toolchain to build a Windows application, then use Apple's toolchain to build the macOS version i.e., build and maintain separate codebases for each platform that you wish to support.

2. **Use a cross-platform framework**. Find a way to leverage code and libraries across more than one platform. This can be done using a cross-platform framework like React Native, Xamarin, Flutter, or Compose Multiplatform. This approach can save time and effort, but it can result in a less polished user experience.

3. **Give up and just develop for the web**. Part of the reason for the success of the web is that you can, theoretically, build an application that runs in a web browser on each platform. This is the easiest approach but can result in a less polished user experience. You also restricted access to hardware e.g., the camera or the GPS.

WHAT **ARE** WE DOING?

We're going to use the Kotlin toolchain. It provides:

- **Native support for Android.** If you want to build a mobile application, this is the main (native) language for the most popular mobile OS. Win win.
- **Cross-platform support for Desktop.** You can also build reasonably good desktop applications for Windows, Linux and macOS with Kotlin/JVM. Libraries aren't as well developed as Kotlin/Android but certainly "good enough" for this course.

Kotlin is one of many "modern" application languages (along with Swift, Go, Dart), with features that make it extremely useful for building applications.

We'll discuss Kotlin in detail and introduce **supporting libraries** to extend its functionality i.e., graphics, user interfaces, databases and so on.