

Designing Large Systems

AKA: Putting It All Together

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Monday, April 24, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 JavaFX App Window Structure
- 2 Technologies Overview
- 3 JavaFX Scene Builder
- 4 Building the Demo GUI

Goals

- ▶ High level review
- ▶ Connecting topics
- ▶ View the larger picture

Context

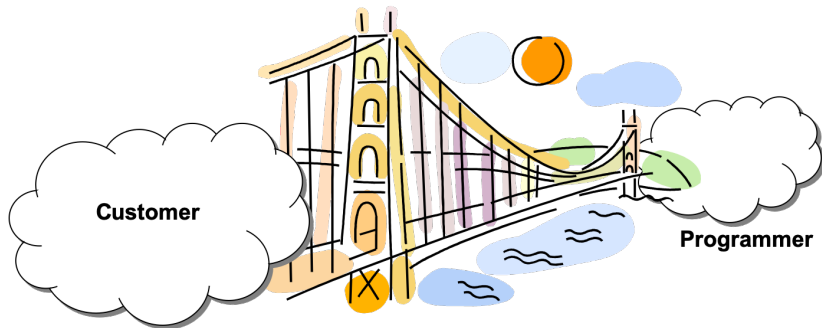
- ▶ Small-Scale Programming
 - ▶ Language Skills
 - ▶ Object-Oriented Basics
 - ▶ 1060/1070 & 1010/1020
- ▶ Medium-Scale Programming
 - ▶ Advanced Object-Oriented Concepts
 - ▶ Abstract Data Types
 - ▶ 2150 & 2120
- ▶ Large-Scale Programming
 - ▶ Software Requirements Analysis
 - ▶ Agile Project Management
 - ▶ Teamwork
 - ▶ 3720 & 4910

“The hardest single part of building a software system is deciding what to build. No part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.”

— Fred Brooks

Role of the Software Engineer¹

A bridge from customer needs to programming implementation



First law of software engineering

**Software engineer is willing to learn the problem domain
(problem cannot be solved without understanding it first)**

¹ "You can't solve it, unless you understand it."

Second Law of Software Engineering

- ▶ **Software should be written for people first**
 - ▶ Computers run software, but hardware quickly becomes outdated
 - ▶ Useful + good software have a long life
 - ▶ For software to be useful and grow, people (aka other software engineers) must be able to understand it

Requirements Drift Is A Huge Problem

- ▶ **Requirements Drift:** When the code diverges from the customer's actual requirements
 - ▶ It usually a slow, hidden process
 - ▶ End up with a final product that does not meet the customers needs!
- ▶ Missing requirements could cause things to fail

User Stories

- ▶ User-centric communication of requirements
- ▶ Partition work by functionality
- ▶ Build understanding of
 - ▶ User roles
 - ▶ Functionality
 - ▶ Motivation
- ▶ Focuses design on the user's needs and priorities

Intentional Design

- ▶ We need to be careful with our design
- ▶ Previously, assignments have been simple assignments
 - ▶ Worked on your own
 - ▶ A few hours
 - ▶ Bad habits developed
- ▶ As a professional, your projects will be much more complex
 - ▶ Will work on a large team
 - ▶ Projects can span years, with multiple incremental releases
 - ▶ Code does not go away
- ▶ Major projects don't rewrite the system with new versions
 - ▶ They just change parts of the system (the systems grow)

Encapsulation

- ▶ Proper encapsulation is key for large systems
- ▶ Where to look for functionality and data should be obvious
- ▶ **Key Question:** What class does this belong to?
- ▶ **Example:**
 - ▶ Size of the game board
 - ▶ `GameScreen` will use
 - ▶ But it belongs to the game board itself

Information Hiding

- ▶ Information hiding is how we control access to encapsulated data and operations
- ▶ **Key Questions:**
 - ▶ Who should see this data?
 - ▶ How can this data be accessed?
- ▶ Private data is protected
- ▶ Only expose what is needed to use the class
- ▶ **Benefits:**
 - ▶ Debugging – Where can this data change?
 - ▶ Security – What code can access this?
 - ▶ Ease of use – clients don't need to know implementation details
- ▶ **Example:**
 - ▶ `IGameBoard` vs 2D array

Separation of Concerns

- ▶ AKA Modular design
- ▶ Every module has a specific role and concerns
- ▶ Modules work together to solve more complicated problems
- ▶ Can have a higher initial development overhead
- ▶ **Benefits:**
 - ▶ Each module is simpler
 - ▶ Reuse
 - ▶ Unit testing
 - ▶ Verification of smaller pieces
 - ▶ Divide and conquer
 - ▶ Teamwork

Design By Contract

- ▶ **Benefits:**

- ▶ Separates responsibilities
 - ▶ Code is not always error checking
- ▶ Simpler design
 - ▶ Design a module and its contracts
 - ▶ Wait for the implementation
- ▶ Communication (aka documentation)
- ▶ Verification

- ▶ Common communication tool
- ▶ Easy documentation
 - ▶ Class diagrams
 - ▶ Documentation of Design Patterns
- ▶ Simplified problem solving
 - ▶ Activity diagrams
 - ▶ Could ask a customer to help with the flow!
- ▶ BUT... any documentation you create... must be maintained (kept up to date)... just like the code

Interfaces

- ▶ Provide a way to describe functionality without implementation
- ▶ Design process, design the interfaces not implementations
 - ▶ See how the interfaces will interact
 - ▶ Design the whole system
 - ▶ Make implementations later
 - ▶ Keep functionality and methods abstract
- ▶ Communication tool
 - ▶ I don't need the details, just the interface
- ▶ Interfaces force abstraction, information hiding & separation of concerns in our designs

Constants

```
public static final
```

- ▶ Constants (`public static final` variables) provide a way to expose information to clients
- ▶ Clients need to know max and min values of an object to use it
- ▶ That data still belongs to the correct object
- ▶ **Example:**
 - ▶ `IGameBoard.MAX_ROWS`

- ▶ Factor out common code (aka abstraction)
 - ▶ Can override for efficiency
- ▶ Add new methods to an existing `interface`
 - ▶ Doesn't break existing implementations

Coding to the Interface

- ▶ The interface says what the object can do
 - ▶ Which is **ALL** the information the client code needs
- ▶ Make all variable's declared type an **interface**
 - ▶ Recall, compiler checks declared type to determine if the method is available
 - ▶ Any implementation of the **interface** *has to* have those methods
 - ▶ Any implementation can be used for the dynamic type
- ▶ Client code is more flexible
- ▶ Methods are more flexible

Inheritance & Abstract Classes

- ▶ Abstract classes are a way to create **interfaces** in languages that don't have them
- ▶ Not as common in Java
- ▶ Still useful for providing implementations of methods inherited from **Object**
- ▶ Inheritance just allows us to re-use code
 - ▶ by extending the functionality of working code

- ▶ Parameterize a data type that exists in our class
- ▶ Very useful for data structures
- ▶ Really just manipulating pointers
- ▶ The declared type is `Object`, so we can call those methods

Design Patterns

- ▶ Considered common knowledge
- ▶ Effective/Elegant solutions to common problems
- ▶ Often implemented in libraries
- ▶ Rely heavily on OOP principles like inheritance
- ▶ A lot more out there:

https://www.tutorialspoint.com/design_pattern/index.htm

Model-View-Controller

- ▶ Architectural pattern
- ▶ Separation of concerns based on layer
- ▶ Simplifies designing a system
- ▶ Makes it easier to swap out views
 - ▶ Still use `interfaces` to make changes easier

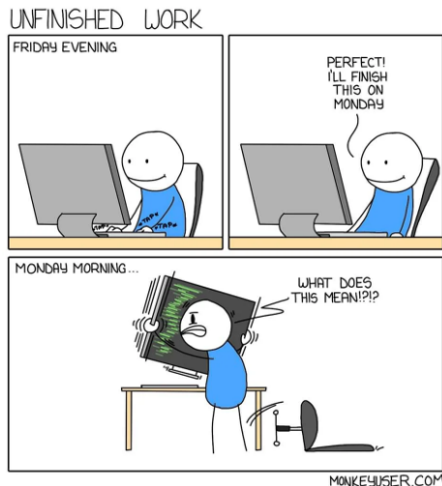
- ▶ Relies on the **Observer Pattern**
 - ▶ Often already partially implemented by the library
- ▶ GUIs take a large number of classes to build
 - ▶ One for each screen we want to display
 - ▶ Each with a corresponding controller
 - ▶ View layer becomes much larger
 - ▶ One for each widget on the screen
 - ▶ Multiple layout related objects
 - ▶ **interfaces** simplify the design and implementation
- ▶ **Example:**
 - ▶ Your project 5 starter code had a setup view and controller, as well as game view and controller

Naming Conventions

- ▶ Meaningful names increase readability
 - ▶ What is this variable for?
 - ▶ What does this method do?
- ▶ Java convention
 - ▶ `packages` start with lowercase
 - ▶ Classes/`interfaces` start with capital letters
 - ▶ Variables/methods start with lower case
 - ▶ Constants all caps
- ▶ **Examples:**
 - ▶ `cpsc2150.extendedTicTacToe.IGameBoard.MAX_ROWS`
 - ▶ `IGameBoard.placeMarker(...)`

Comments

- ▶ Increase readability
- ▶ You or someone else will have to revisit this code later
- ▶ Don't rely on your memory, document!



- ▶ Project documentation, comments are all there for a reason.

USE THEM



I Am Devloper

@iamdevloper



Remember, a few hours of trial and error can save you several minutes of looking at the README.

2:11 AM · 07 Nov 18

Magic Numbers

- ▶ Bad for readability
 - ▶ Why does this loop go to 10? What does it mean?
- ▶ Bad for changes
 - ▶ Size of board changes
- ▶ Must be replaced with meaningful variables
 - ▶ Row and column vs size
- ▶ Can't just find and replace
 - ▶ Is every 10 the same?

Can't I Just Copy from Stack Overflow?

Doctors: Googling stuff online does not make you a doctor.

Programmers:



Can't I Just Copy from Stack Overflow?

- ▶ Context
 - ▶ Their context will never be your context!
 - ▶ Answers often refer to concepts
- ▶ Focus on learning problem solving skills, not just syntax
- ▶ You won't get paid to just copy code from Stack Overflow
 - ▶ especially when it may not be correct
- ▶ It's a helpful resource, like a distributed code review
 - ▶ but use it carefully!

New Languages

- ▶ Unless you took 1060 (or AP Computer Science)
 - ▶ You learned Java largely on your own, we just provided some tutorials
- ▶ New languages come and go, you will need to learn several over your career
 - ▶ Is Python a “new” language?
 - ▶ What languages do you know?
- ▶ General principles are the same



iDK @ SCR Saga
@kevinkaywho



Found a job opening that requires
8+ years of Swift experience.

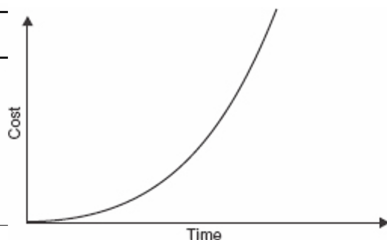
Swift is a programming language
that came out 3 years ago.

10:05 PM · 18 Aug 17

Testing

- ▶ Part of the **Software Quality Assurance** process
- ▶ Goal is to find failures (as early as possible) so we can fix the underlying faults easily and cheaply
- ▶ Lots of methods to develop test cases. . . and we need them all
- ▶ Need to automate test cases
 - ▶ Helped by programming to the interface

Stage error is found	Comparative cost
Requirements	\$1
Coding	\$10
Program testing	\$100
System testing	\$1,000
User acceptance testing	\$10,000
Live running	\$100,000



Automated Testing

- ▶ JUnit is a framework that automates unit testing
 - ▶ We can test our code after any and all changes
 - ▶ Keep the bar green!
 - ▶ Don't put any errors into the system
- ▶ Takes a lot of work to set up. . . but saves us the work of manually running all the test cases
- ▶ But they must be maintained in parallel with the code

Thorough Testing



Brenan Keller
@brenankeller

A QA engineer walks into a bar.
Orders a beer. Orders 0 beers.
Orders 999999999999 beers.
Orders a lizard. Orders -1 beers.
Orders a ueicbksjdhd.

1:21 PM · 30 Nov 18

Code Reviews

- ▶ Help improve our code
- ▶ Identifies potential faults
- ▶ Finds efficiency issues
- ▶ Builds understanding of the technology
- ▶ Enforces best practices
- ▶ Finds better design solutions
- ▶ Enforces readability of code
- ▶ Requires a committed team

Formal Reasoning

- ▶ A better version of tracing
- ▶ Shows what the code does on **ALL** values of the variables
- ▶ Avoids incorrect generalizations
- ▶ Can be used for program verification
- ▶ Forces more careful consideration of the logic
- ▶ Relies on our correct and accurate contracts
- ▶ But it's very hard to apply to larger projects

A Good Design Helps With. . .

- ▶ Reuse

- ▶ You can reuse the encapsulated objects.
- ▶ Information hiding protects our data, even for systems we aren't considering yet.
 - ▶ The documentation tells us how to use it
- ▶ By keeping the roles/concerns very specific, we can reuse it.
- ▶ Consider making a POS system for a pizzeria. If we have one module whose role is processing credit card payments, we can reuse that module in another system. If the credit card payment system needs to know information about pizzas, then it can only be used in a system for a pizzeria.

A Good Design Helps With. . .

- ▶ Reuse
- ▶ Adapting to change
 - ▶ **Separation of Concerns:** if one role changes, and there is only one module that plays that role, its only one place to make the change.
 - ▶ **Example:** Command line to GUI
 - ▶ A well designed system is easier to change when adding new functionality.

A Good Design Helps With. . .

- ▶ Reuse
- ▶ Adapting to change
- ▶ Debugging
 - ▶ Each encapsulated object can be tested individually.
 - ▶ If a test case fails, we know the bug is in that object.
- ▶ Each module has a specific role in the process.
 - ▶ What role didn't work?
 - ▶ That module contains the bug
- ▶ Hiding information limits access to the data.
 - ▶ If the data is wrong, you have a limited number of places where that change could have been made.

Design Questions to Ask

What if _____ changes?

- ▶ Any of your numbers
 - ▶ Avoid “magic” numbers
 - ▶ Use a constant instead
- ▶ User interface
 - ▶ Have all input and output be in one module (remember MVC?)
 - ▶ Boundary objects
- ▶ Order of events
 - ▶ What needs to be in what order
 - ▶ what's flexible?
 - ▶ Clear documentation
- ▶ Avoid repeated code
 - ▶ Separate into a module
 - ▶ Only one place to make changes

Design Questions to Ask

Does <object> know _____ ? Should it?

- ▶ Hide information
- ▶ If an object **A** doesn't need to know info about object **B**, then **B** can change without affecting **A**.

Design Questions to Ask

Are these related?

- ▶ To encapsulate, or not to encapsulate
- ▶ Encapsulating related things makes it easier to keep related data in sync
- ▶ Encapsulating unrelated things makes it harder to reuse

Design Questions to Ask

Can someone use _____ 5 years from now without having to ask me?

- ▶ Do they need to know the programming language?
- ▶ Do I have clear documentation and comments?
- ▶ Did I hide implementation details, and use general abstract concepts

A Final Note

- ▶ All of this takes time and practice
- ▶ Follow best practices in all your (coursework²) development
 - ▶ Saves time if you have to revisit it or debug the code
 - ▶ Builds better development skills
 - ▶ BUT you will know when to break new ground
- ▶ Make a personal project³
 - ▶ Host on GitHub
 - ▶ Add it to your resume
 - ▶ Something to show off at an interview
 - ▶ Produce high quality, clean code!

²Start assignments early. Never assume put off starting an assignment thinking: "It is easy, I will start it closer to the due date and do something else instead."

³Must be non-assignment related or you will run into under academic integrity issues

Summary

- | | | | |
|----|--------------------------------|----|-----------------------------|
| 1 | Requirements Engineering | 14 | Generics |
| 2 | Intentional Design | 15 | Testing |
| 3 | Encapsulation | 16 | Code Reviews |
| 4 | Information Hiding | 17 | Formal Reasoning |
| 5 | Separation of Concerns | 18 | Design Patterns |
| 6 | Design by Contract | 19 | MVC and GUIs |
| 7 | User Stories | 20 | Magic Numbers |
| 8 | UML | 21 | Naming Conventions |
| 9 | Interfaces | 22 | Comments and Documentation |
| 10 | Constants | 23 | A Good Design Helps With... |
| 11 | default Methods | 24 | Design Questions to Ask |
| 12 | Coding to the Interface | | |
| 13 | Inheritance & Abstract Classes | | |