CPSC 310 - Introduction to Software Engineering

Jan. 7th - Lecture 1: Introduction

- What is Software Engineering?
  - Cross-cutting concerns
    - Information Security: protect sensitive information from unauthorized activities - inspection, modification, recording and disruptions
    - Ethics: making decisions that respect privacy, promote fairness, benefit society; adhering to professional codes, navigating ethical dilemmas, understanding the impact of unethical practices
  - 210 View: The process of transforming a mental plan of desired actions for a computer into a representation that can be understood by the computer
  - 310 View: The **socio-technical** process of transforming...
  - SE: Invent → Design → Build → Validate → Maintain → Research → Improve → Deploy → Invent
- Course Motivation
  - Modern software development
  - 210: Construction; 310: Design; 319: Team Dev; 410: Special topics
  - Objectives
    - Evaluate SE processes
    - Elicit, deconstruct, and refine functional requirements and quality attributes
    - Devise and Justify high- and low-level designs
    - Iteratively derive implementations of a design of reasonable complexity

        - Carry out the implementation of a design incorporating ethical and security
        - Independently acquire and apply modern and unfamiliar technology
        - Validate systems using both black-box and glass-box approaches
- Project
    - Deliberate Practice: practice an intentionally designed set of skills
        - Goal of practice is mastery
        - Skills are guided to emphasize specific abilities
    - Project Design
        - Experiential software development experience
        - Fully based around functional correctness
            - non-trivial requirements, data and algorithm heavy
            - assigned requirements
            - specified environmental restrictions
        - Integral quality focus
        - Team-based development
        - Modern software development stack
    - Expectations
        - 6 hrs/wk * 2 people * 12 wks
        - C1 & C2 probably most time consuming
        - Failure cases: ghosting partner + starting the day before
    - Assessment
        - Two main test paths
            - Unit testing within project
            - rate-limited integrating testing
    - AutoTest
        - Formative assessment
        - Feedback will be communicated with a bucket evaluation
    - Deliverables
        - C0: Jan 16 @ 1800

        - C1: Feb 14 @ 1800
        - C2: Mar 14 @ 1800
        - C3: April 4 @ 1800
- Assessment proportions
    - 25%: Project implementations
    - 30%: 4 * Quizzes
    - 40%: Final Exam
    - 5%: 10 * Journals (1% each = 5% bonus for both quizzes and project if all are completed

Jan. 9th - Lecture 2: Black-box Testing

- "Testing shows the presence not the absence of bugs."
- Why test software?
    - Uncaught exception because of a single line of code (code designed for Ariane 4 led to integer overflow because Ariane 5 could go faster), caused the explosion of the Ariane 5
    - Technical debt: choosing easy solution as of right now but in the future there might be complicated features that need to be implemented and the easy solution will require substantial amount of rework
    - Tight deadlines encourage bad decisions, Technical debt, Poor communication across reporting lines, Interpreting the results of tests
    - Giving a finite amount of time and resources, testing enables validating that a system has an acceptable risk of costly or dangerous defects.
- Why not test?
    - Good reason: do not how; legacy code
    - Bad reason: bad design; does not catch bugs (now); slow; boring; hard to change; QA's job
- Common testing assumptions
    - "The cost of fixing faults rises exponentially with how late they are detected"
        - commonly stated but based on evidence from 20+ years ago
        - does not seem to hold for modern processes, tools, and languages

- Test-driven development
    - Step 1: Write a failing test
    - Step 2: Make the test pass
    - Step 3: Refactor your code
    - In the project: C0 - Step 1; C1 - Step 2 (implementation); C2 - Refactor
- Testing terminology
    - SUT/CUT: system/code under test
    - Glass-box: tests that consider internals of CUT
    - Black-box: tests that are oblivious of internals of CUT
    - Effectiveness: the probability of detecting a bug per unit of effort
    - Higher testability: more effective tests, same cost
    - Lower testability: fewer weaker tests, same cost
    - Repeatability: the likelihood that running the same test twice will yield the same result
    - Flaky test: tests that pass/fail non-deterministically
- Each test has
    - a meaningful name
    - setup: **`@Before`**
    - execution
    - validation
    - teardown: **`@After`**
- Unit tests: test behavior of a single unit
- Integration tests: test how parts of the system work together
- End-to-end tests → Flaky tests: as scope increases, tests are more likely to become flaky
- Kinds of tests by purpose (Test Suites)
    - Smoke
        - Captures core functionality
        - Stability of the build
        - Very reliable as they need to catch major issues, very fast to run, small test suite
        - run to find any major issues

    - Regression
        - captures all existing functionality (no new tests)
        - quality of the build
        - slower to run as the test suite is typically large
        - run to ensure no existing behavior has changed (e.g. after a refactor)
    - Acceptance
        - Captures new functionality added by feature (required functionality from users perspective)
        - Run to ensure end product solves the problem it was intended to solve
- Black Box Testing
    - Steps: read the specification; write tests; go to step 1
    - On test failure, three options: fix system; fix specification; fix test
    - Advantages
        - finds high severity bugs: act like users
        - easy to get started: more clear what to test
        - less overfitting to implementation: emphasis on specification
        - parallelize development and testing: no need to access implementation
    - Disadvantages
        - need well-defined specifications
        - poor explainability
        - intentional handicap
    - When reading a method specification to write tests, we look for
        - Equivalence Class Partitioning (ECP)
            - a systematic way of covering the (possibly unbounded) space of inputs/outputs
            - help you to avoid redundant tests
        - Boundary Value Analysis (BVA)
            - help to find most common errors hiding in edge cases
    - ECP

        - Failures are sparse in the space of possible inputs, but dense in certain parts
        - IO partitioning
            - Partitioning all possible inputs to CUT into equivalence classes which characterize sets of **inputs** with **something in common**
            - we do not consider intentions/actions/outputs of the CUT, except to assert on the correctness of an output for an input
            - Partitioning all possible outputs to CUT into equivalence classes which characterize sets of **outputs** with **something in common**
            - Relating Input and Output partitioning
    - BVA
    - Summary
        - The two techniques create a highly effective test suite, but they rely on well-defined specifications.
        - Require individual judgements

Jan. 14th - Lecture 3: Languages and Async

- Async
    - Concurrency
        - Executing multiple program parts at the same time
        - Actual ordering is indeterminate
        - Shared state makes it hard: W before R different than R before W
        - Problem: Deadlock etc
        - We want it because it may provide performance benefits
    - For long running processes
        - `setTimeout()`
        - File Input and Output
        - Network requests
    - `setTimeout(...)`

    -
      ```
      setTimeout(function() {
          console.log("Hello");
      }, 1000);
      ```

    - Anatomy of a callback

      ```
      const onComplete = function() {
          //
      };
      setTimeout(onComplete, 1000);
      ```

    - Error-first callback idiom

      ```
      foo.action(msg, function(err, data) {
          if (err) {
              // handle
              return;
          } else {
              // worked, use data
          }
      });
      ```

- Example program: determine the output

  ```
  function start() { doIt(); }
  function doIt() {
      console.log("start");
      setTimeout(function() {
          console.log("done");
      }, 2000);
      console.log("end");
  }

  start();
  ```

    - the output will be: start, end, done

- Why would we do...?

```
function makeItSo() {
    // ...
    setTimeout(function() {
        // do something that takes a long time
    }, 0);
    // ...
}
```

- Non-trivial example: read first file in a directory

```
const files = fs.readdir(source);
if (fs.stat(files[0]).isFile()) {
    const data = fs.readFile(files[0]);
}
```

  - This does not work as files IO is asynchronous, so function calls like **`readdir, stat, readFile`** are asynchronous
- 3 ways to resolve asynchrony: callbacks, promises, **`async`** and **`await`**
  - Callbacks add too many brackets, reducing readability
  - Promise has two execution states: pending, or settled (fulfilled + rejected)
    - Fulfilled: **`then(...)`**
    - Rejected: **`catch(...)`**
  - **`async`** & **`await`**: syntactic sugar (we can use try-catch structure)

```
async function readDirectory() {
    try {
        await ...
    } catch () {

    }
}
```

- Promises
  - **`.all`**
  - **`.allSettled`**
  - **`.race`**

- Languages
    - Static vs. Dynamic Types
        - Variables have types in statically typed languages
        - Values have types in dynamically typed languages
    - ???
    - Value comparison
    - Syntax & Semantics
    - Static analysis
        - any programmatic analysis of a program that only inputs the program text itself and does not need to execute the program
        - performed in compilation, and also: ???
    - Linters help developers

Jan. 16th - Lecture 4

Jan. 21st - Lecture 5: Glass Box Testing

- State Machine Diagram
    - Used to model complex behaviors and models
    - Extremely useful for event-based modelling
        - essentially every UI
    - Can be used to reduce the complexity of event systems
        - e.g. reasons about when events are applicable
        - transitions run to completion (like JS methods)
- Glass Box Testing
    - These are tests written against implementation
    - Checks the implementation itself
    - Can evaluate the quality of test suite through **mutation testing**
    - Easy to overfit test suite to the implementation, rather than the spec
    - Fuzzing can decrease over-fitting
- Test Coverage
    - Line/Statement coverage

    - Branch coverage
    - Path coverage
- Line/Statement coverage

    -
      ```
      # line coverage
      eval (x: number, c1: bool, c2: bool): number {
          if (c1)
              x++;
          if (c2)
              x--;
          return x;
      }

      # eval(0, true, true) covers all line, as each line
      is executed
      ```

    -
      ```
      # statement coverage
      eval (x: number, c1: bool, c2: bool): number {
          if (c1 || someOtherThing())
              x++;
          if (c2)
              x--;
          return x;
      }

      # eval(0, false, true) covers all statements
      ```
- Branch coverage

    -
      ```
      # Branch coverage
      eval (x: number, c1: bool, c2: bool): number {
          if (c1)
              x++;
          if (c2)
              x--;
          return x;
      }

      # Case 1: eval(0, true, true)
      # Case 2: eval(0, false, false)
      ```

- Path coverage
    - often isn't feasible in practice because it would just take way too long.
    - for example, path coverage for **`for`** loops means evaluating every number of iterations the loop could take
- Coverage Advantages/Disadvantages
    - Advantages: cheap to compute, intuitive, actionable
    - Disadvantages: artificial thresholds, does not imply correctness
- Mutation Testing
    - The coupling hypothesis on which mutation testing relies
        - states that mutant versions of programs are similar to the kinds of mistakes developers make, and that these mistakes alias to real faults
    - it is appealing because it provides a means to simulate developer mistakes, which could lead to real bugs
- Automated Testing: Fuzzing
    - Automated testing $\neq$ Test automation
    - It is about test-input generation (generate test inputs that expose bugs in a program); or about test case/suite generation (generate test suites that expose bugs in a program)
    - Test-input generation
        - Assume a program P that takes in input i.
        - Goal: given P, generate the inputs i that expose bugs
    - Fuzzing
        - they are test-input generation algorithms where:

          it has some elements of randomness;

          it may use feedback from program execution: $P(i)$ or **`analyze`**$(P(i))$ to guide the generation of next input
        - Simplest: Random fuzzing (generate i randomly) - also **black-box fuzzing** as feedback from program is not used to guide new generation
        - To have less random inputs: generate inputs based on specifications

Jan. 23rd - Lecture 6: Testability, Assertions, and Intro to Ethics

- From execution to validated behavior
    - Just because code can be executed, does not imply correctness (only exception is crashing bugs)
    - Assertions bridge this gap
        - assert on the behavior of the execution
        - want to consider both valid and invalid behaviors
    - Two common strategies (four phase test, given-when-then)
- Given-when-then
    - Motivation: testing code is hard to read/reason about
    - Goal: "executable specifications" (BDD)
- Equality
    - Strength of assertion
        - `expect().to.be.defined` (most broad)
        - `.to.be.an()`
        - `.to.equal()` (most narrow)
    - Assertions that are stronger usually closely follow the specifications
    - Weaker assertions are still useful to help with debugging (use to short circuiting the code)
- Testability
    - Observability - extent to which the response of the CUT to a test can be verified
        - What do we have to do to **identify** pass/fail?
        - How **expensive** is it to do this?
        - Can we **extract** the result from the SUT (system under test)?
        - Do we **know enough to identify** pass/fail?

        -
          ```
          // Given
          const a = new Animal();
          expect(a.isHungry()).to.be.true;
          expect(a.amountEaten()).to.equal(0);

          // When
          a.eat()

          // Then
          expect(a.isHungry()).to.be.false;
          expect(a.amountEaten()).to.be.greaterThan(0);
          ```

- Controllability - extent to which the CUT can be made to perform specific actions of interest
    - Can we control the SUT during a test?
    - How expensive is it?
    - Does the system make running a test **impractical**?
    - Given a test goal, do we have enough **information** to create an adequate suite?
    - How much tooling can we afford?

    -
      ```
      foo() {
          // chunk 1
          // chunk 2
          // ...
      }
      ```
        - This type of function is hard to test as it does many things all at once. We wish to refactor each chunk so that we can test them independently.
- Automatability - ability to execute the test programmatically
    - Can tests be executed without human intervention?
    - What is the cost of automated infrastructure?
    - What is the benefit of using a test infrastructure?
        - Executions can be batched

            - Run on same configuration / hardware
            - Global visibility of results
            - Enables regression testing and integration testing
        - Huge economic advantages
        - Tests are code too (subject to their own fault)
        - Not all test failures uncover faults (defect in itself, flaky test due to non-determinism, REQUIRED shortcoming - undefined behavior)
        - How to recognize a "true" failure?
            - Developer changed source, but test passed on the next iteration
    - Isolate-ability - degree to which the element under test can be validated on its own
        - Can the component being tested be isolated?
        - Isolated components are: simpler to reason about; less prone to non-determinism; faster
        - What is the cause of this?
        - If an element cannot be naturally isolated, can we simulate it?
            - Simulated dependencies can also enable validating unusual states.
        - **Being able to isolate where the error occurs**
    - Mocking Dependencies
        - Often we rely on portions of the system which are non-deterministic, slow, user-driven, not yet built
        - **Mocks** adhere to the contract (interface) but simulate behavior
        - real implementation is substituted in production
        - Uses Mocks for Isolation for **unit test**, for larger scale tests, we do not rely on Mocks
        - Need to think about "how to set up the hierarchy", "how to declare", "how to instantiate", and "what to implement"
- Ethics
    - Ethics in SE
        - Provide a moral compass to **guide** professional developers

        - project design: identify and communicate benefit/impact
        - testing: ensure accuracy, security, and fairness
        - maintenance: address vulnerabilities and ongoing conformance
    - Professional Ethics (ACM, IEEE)
        - Codifies standards and values for SEs
        - Obligations that must be upheld to improve people's lives
- Ethics in Testing
    - Poorly validated software can clearly cause direct harm
    - Software validation can also cause unanticipated harm
    - During testing, we consider not just whether a behavior is correct, but also if the "correct" behavior is "right"

Jan. 28th - Lecture 7: Ethics and Intellectual Property

- Names
    - We have all created some kind of user object at some point, and users have names. They have first names and last names.
    - Here are false assumptions about names
        - Names do not change, names are set at birth, all names consist of more than one letter, names are not in ALL CAPS, etc.
- Dealing with data
    - Vast amounts of data are often collected, analyzed or sold to third parties
    - Anonymization is frequently used to avoid reputational harm to the collecting organization
    - Anonymization is simple to perform, but has many shortcomings
    - k-anonymity ensures information for a person cannot be distinguished from at least $k-1$ other people in a dataset
- Algorithmic Bias
    - systematic and repeatable errors in a system that create "unfair" outcomes, such as "privileging" one category over another in ways different from the intended function of the algorithm

    - Why can't the law take care of this?
        - A wide variety of behaviors are socially negative, but not all of these can be legislated
        - Only legal restrictions is a low bar
        - the SE community needs to consider legal, moral and societal impact of the work
- Engineers need to consider ethics
    - We use common-sense before considering code of ethics, but that is problematic
- Intellectual Property (IP)
    - As an IP user, I need to **honor creators' rights** and have limited rights to use existing work
    - As an IP creator, I have the **right to profit** and must honor rights of other creators when I use their work
    - As an employee, have responsibility to **comply with IP laws** and to protect trade secrets
- IP protection
    - All modern tech businesses rely on a comprehensive set of approaches to protect their IP
        - Trademark: Distinguished feature that differentiates a product or organization in the marketplace
        - Copyright: mechanism to protect creative work
        - Trade secrets: protects information using confidentiality agreements
        - Patents: state-conferred rights to an inventor for their invention
        - Licenses: Used with the other options to capitalize on IP
- Individuals and Organizations need protection
    - Creating valuable products entails risk and investment
    - Ideas themselves are not inherently valuable
    - Protections exist to reward your risk by enabling you to profit from your product without having someone initiate it
- IP: Balancing creative and public rights
    - IP are creations of human intellect
    - Need to balance the rights of creator with the public

    - Want to stimulate, not inhibit, the creation of new work
    - Creator: time-bound exclusive rights
    - Public: Restricted rights for use of existing work in new creations
- Derivative works
    - IP policies strive to **not inhibit** future innovation
    - This allows: improvements to existing inventions, new works that include portions of existing work, innovative interpretations of existing work
- Licenses
    - are widely applied in the software space and clarify how code can be used and whether it can be modified and distributed by others
    - Restrictive/Copyleft: generally require the same rights for derivative work
    - Weak Copyleft: enable portions of a system to be released under non-copyleft licenses - specific exceptions for linking to libraries
    - Permissive: require only attribution, allowing non-derivative portions to remain proprietary
    - e.g. GPLv3 (General Public License), LGPLv3 (Lesser General Public License), MPLv2

Jan. 30th - Lecture 8: Kinds of Process

- Software is built by teams of people
- Software Project Risks
    - Users
    - Requirements
    - Project Complexity
    - Planning and Control
    - Team
    - Organizational Environment
- Projects need good plans
    - A software process is a **structured** set of activities for developing a software system
    - It defines who does what, when, and how, to reach a goal
    - Processes have descriptions that discuss

        - products: the outcome of a process activity
        - stakeholders: people who care about the outcome
- Process phases
    - Many different software processes - each with their own strengths and weaknesses
    - All include - requirements gathering; architectural design; detailed design/specification; implementation; integration; testing; deployment; maintenance
    - Goal is to - mark out clear steps; produce tangible items; allow for review of work; specify actions to perform next
- Efficiency progress affected process
- Waterfall scenario
    - Requirements: 18 months
    - Design: 12 months
    - Implementation: 18 months
    - Verification: 12 months
    - Maintenance: 15+ years after version 1
- Dawn of Extreme Programming
    - "Communication, Simplicity, Feedback, Courage"
- Evolution into Agile Development
    - Individuals and Interactions over processes and tools
    - Working software over comprehensive documentation
    - Customer Collaboration over contract negotiation
    - Responding to Change over following a plan
    - New approaches: Users stories, Test Driven Development
- Agile Software Process
    - Emergent design: start at the **Minimum Viable Product** (MVP), then grow from there, identifying duplication and introducing abstractions as needed to solve duplication
    - Refactor code: rather than doing **big** design first (make pragmatic choices along the way -- this is not a license to write terrible code) AND the architectural spike must come first
- Spikes

    - These are short, intense, activities preceding development iterations (or sometimes fit in between, but not typically).
    - Architectural spike: when product is being devised, decide on high- and medium-level architectures
    - User Interface spike: before any UI development, decide on look and feel, UI framework, plan for UI expansion.
- Scrum - Agile methodology
    - Product backlog - sprint backlog - daily scrum meeting - potentially shippable product increment
    - Roles
        - Product Owner: defines features of the product; prioritizes features according to market value; adjust features and priorities every iteration, as needed
        - Scrum Lead: facilitates scrum process; helps resolve problems; shields team from external interferences; NOT the manager
        - Team: self-organizing, self-managing, cross-functional; developers, designers, managers, clients, etc.; 7 ± 2 people
    - Artifacts
        - Product backlog - prioritized list of backlog items (PBIs); PBIs specify a customer-centric feature (User Story form); Effort estimated by Team, priority estimated by Product Owner
        - Spring backlog - Contains list of user stories that are negotiated by team and product owner from the product backlog; negotiated PBIs broken down into specific tasks
        - Burndown chart
    - Ceremony
        - Team presents what is being accomplished during the sprint
        - Typically takes the form of a demo of new features or underlying architecture
        - Informal
        - Review and retrospective can be one or separate meetings
- Kanban - another Agile process methodology
    - No sprints but a continuous process without a sprint backlog
    - Each column on the board has a **work-in-progress limit** related to the team's capacity

    - No specific release dates, up to the team for decision

Feb. 6th - Lecture 10: Automation and DevOps

- Automation
    - Automation in the project
        - 310 bots are examples of continuous integration and rely on automation to check and evaluate
        - the cornerstone to these analyses is automation: run automatically and continually provide feedback
    - Coding and Deploying
        - Deployment: code must be built into a release and deployed into production for users to use it
    - Continuous Integration Loop (**CI**)
        - Change code → Get code and dependencies → Build → Run tests → . . .
        - Actions goes in the forward direction, Feedback goes in the backward direction
        - This loop is **repeatable, and reliable**
        - Automated pipeline: triggered by code change events
        - Ensures absence of obvious build issues and configuration issues
        - Ensures tests are executed
        - May encourage more tests
        - Can run checks on different platforms, security checks
    - From Integration to Release
        - Release management: versioning and branches
        - Goal: continuous releases - every merge-to-main should be a release
    - Semantic versioning
        - A version number MAJOR.MINOR.PATCH, increment
            - MAJOR when make incompatible API changes
            - MINOR when you add functionality in a backwards-compatible manner

        - PATCH when you make backwards-compatible bug fixes
    - Additional labels for pre-release and build metadata are available as extensions
        - $\tilde{}1.2.3 \to 1.2.*$
        - $\hat{}5.4.3 \to 5.*.*$
- Why semantic versioning?
    - Consider a library `Firetruck` that requires package `Ladder`
    - At the time of library creation, `Ladder` is at version 3.1.0. Since the library uses functionality from 3.1.0 version of `Ladder`, it is also safe to use `Ladder` that are versions no less than 3.1.0 and no greater than 4.0.0
- From **release** to **continuous release**
    - Traditional view (Boxed software)
        - Working toward fixed release date, QA heavy before release
        - Release and move on
        - Fix post-release defects in next release or through expensive patches
    - Frequent releases
        - Incremental updates delivered frequently (weeks, days, ...)
        - Automated updates ("patch culture", "update done? ship it")
    - Hosted software (Software as a Service - SaaS)
        - Frequent incremental releases, hot patches, different versions for different customers, customer may not even notice update
- Continuous **Delivery** vs. **Deployment**
    - Continuous delivery: one manual step - deploy to production
    - Continuous deployment: every step is automatic
    - Unit Test → Platform Test → Deliver to Staging → Application Acceptance Tests → Deploy to Production → Post Deploy Tests
- Automation facilitates rapid progress

        - De-risks development through continual evaluation
        - Facilitates rapid problem identification
        - Eases rollback: past states are well understood
        - Decreases resistance to release (individual steps well understood, possibility of rollback known)
- DevOps - Development → Operations
    - Software developers and the operators are not on the same teams
        - Poor communication, slow resolution of production issues, lengthened feedback loop, higher deployment cost/time
    - Goals
        - Collaboration: better coordinate between developers and operations
        - Efficiency: reduce friction in bringing changes from development into production
        - Holistic: Consider the entire tool chain from development to operation
        - Configurations as code: documentation and versioning of all dependencies and configurations and environments
        - Automation: enable continuous delivery, monitoring
        - Quick: small iterations, incremental and continuous releases
    - QA does not stop in dev
        - ensuring product builds correctly
        - ensuring scalability under real-world loads
        - supporting environment constraints from real systems
        - efficiency with given infrastructure
        - monitoring
        - bottlenecks, crash-prone components
    - Tooling/automation
        - Infrastructure as Code (e.g. Terraform, CloudFormation), CI/CD (e.g. Travis, Jenkins), Test Automation, Containerization (e.g. Docker), Orchestration (e.g. Kubernetes), Software Deployment, Measurement/Monitoring (e.g. Prometheus, Grafana)
    - Docker

  - Lightweight virtualization
  - sub-second boot time
  - shareable virtual images with full setup and configuration
  - used in development and deployment
  - separate docker images for separate services (web server, business logic, database)
  - **weaker isolation than virtual machines: less secure**
- Configuration management, Infrastructure as Code (IaC)
  - scripts to change system configurations (configuration files, install packages, versions, ...); declarative vs. imperative
  - ...
- Kubernetes container orchestration
  - manages which container to deploy to which machine, launches and kills containers depending on load
  - manage updates and routing
  - ...
- Monitoring
  - Monitor server health, service health
  - collect and analyze measures or log files
  - dashboards and triggering automated decisions
  - Tools (e.g. Grafana as dashboard, Prometheus for metrics, Loki + ElasticSearch for logs)
  - push and pull models
- Testing in Production
- Crash telemetry - Sending error reports to developers
- What if...
  - we had plenty of subjects for experiments?
  - we could randomly assign subjects to treatment and control group without them knowing
  - we could analyze small individual changes and keep everything else constant
  - These are ideal conditions for controlled experiments
- A/B Testing

        - Experiment size - with enough subjects (users), we can run many experiments; even very small experiments become feasible
        - Implementing alternative versions of the system - using feature flags, separate deployments
        - Map users to treatment group - randomly from distribution
        - ...
        - Feature flags
            - boolean options
            - features are tracked explicitly and documented, localized and independent
            - external mapping of flags to customers
            - BMW uses feature flags as paywalls: subscription for heated seats, more power
    - Canary Releases
        - Testing releases in production
        - Incrementally deploy a new release to users, not all at once
        - Monitor difference in outcomes (e.g. crash rates, performance, user engagement)
        - Automatically roll back bad releases
        - Simple version of A/B testing
        - Telemetry essential
    - Chaos experiments
        - It allows to measure a system's traffic, patterns of use, resource utilization, and practical ???

Feb. 11th - Lecture 11: Requirements

- Requirements
    - A "wicked problem" is one that can only be defined while solving the problem - one must solve a problem once to define it and then solve it again to create a solution that works
    - Curriculum design as a wicked problem
        - Four main stakeholders: students, TAs, instructors, UBC

        - Differing goals
        - wicked characteristics: no definitive formulation, no stopping rule, solutions not true-or-false (just good-or-bad), no ultimate test of a solution
    - Software will always be hard because of complexity, conformity, changeability, and invisibility
    - Requirement engineering lifecycle
        - Elicitation
            - The process by which requirements are gathered
            - Using whatever sources of information are available: client, users, observation, videos, documents, interviews
        - Validation
            - Have we elicited and documented the right requirements?
            - Are they consistent?
    - Two main kinds of requirements
        - Functional requirements: specifies what the system should do
        - Quality attributes: properties that the product must have, usually described using adjectives, often strongly impact system success
            - Security, Reliability, Performance, Legal, Usability, etc
    - Measurable quality attributes
        - Security: intercepting the email and gain info from it; percentage of block phishing emails; clients getting private info
        - Reliability: uptime percentage
        - Performance: time to deliver an email
        - Usability: how many times people access help
- Requirements Evolution
    - Size alignment
        - Large requirements (comprehensive reqs / formal specs)
        - Medium requirements (use cases)
        - Small requirements (user stories)
    - Starting point

        - Spending a lot of time planning, lead to large documentations that involve many stakeholders
        - very difficult for a client to play out the behavior of the software with large specifications
    - Medium requirements - Use cases
        - Still interconnected - they would refer to one another
    - Smaller requirements - smaller Use cases
    - Pictorial requirements
        - use case diagrams show packaging and decomposition of use cases not their content
        - each ellipse is a use case, actors can be other systems
    - Problems with use cases
        - Still difficult for a client to play out the behaviour based on the description because it is so in-depth
        - Contributes to a mismatch between client expectations and what the developer does

Feb. 13th - Lecture 12: User Stories

- Components
    - Role Goal Benefit (sometimes just called the "user story"), Definitions of Done (sometimes called Acceptance Criteria, solution to Role-Goal-Benefit), Engineering Tasks
- Scrum Timeline
    - Product Backlog → Sprint Backlog → Spring (2-4 weeks, with daily scrum meeting) → Potentially Shippable Product Increment
    - Always iterating from working product, to working product, even if only some of the features are present
- Connecting user stories with software engineering
    - Role → Goal → Benefit (Helps prioritize user stories) → Definitions of Done → Engineering Tasks → Story Points
    - Role: which users to test with
    - Goal: what behaviour to build
    - The benefit informs what the definitions of done are

    - Definitions of Done: what tests to specify
- Role Goal Benefit statement
    - Have a role (specific type of user expressing the need), have a goal (the desired behaviour), have a benefit (the outcome of the behaviour)
    - "As a customer, I want to be able to buy something, and then get it"
- Example of Definitions of Done
    - "As a shopper, I want to be able to buy something and then view past purchases so that I can track spending on the site"
    - DoD: User clicks the button buy, and it appears in their purchased items, and is shipped to the user
    - It is user-level (should not mention code), it is client-oriented solution domain
- Good DoD
    - Contracts with clients, know you have completed a user story, and can mark it resolved
    - Like a sequence diagram that explains how the features work
- Estimating story points
    - A story point $\approx$ an hour of developer work
    - Estimation is important to ensure team can complete work
    - widely used to see if ahead/behind schedule
    - enables greater responsiveness or team awareness
    - Estimation traditionally was made by a developer, guessing how long a story would take them; moving to more ontological approaches
    - Case of local estimation approach: 1, 2, 3, 5, 8 (Fibonacci sequence of story points)
- From user stories to epics to themes
    - User stories are often related, but independent
    - Epics group together related user stories (usually delivered over multiple sprints, grouping of stories that share an overall goal)
    - Themes group epics and describe even higher-level objectives
- Good things about small requirements
    - Clear linkage between problem domain and solution domain
    - Decrease risk

    - Still legal documents, do not have hierarchy the way prior requirements did
- Assessing user stories: INVEST
    - Agile approaches depend on user stories
    - I - Independent, N - Negotiable, V - Valuable to users or customers, E - estimatable, S - small, T - testable
    - Independent - user stories should be independent of one another, but obviously they are not implemented in a vacuum
    - Negotiable - clients have to be able to fully understand and critique how a feature will work
    - Valuable - best if it is providing value to a customer; customers are the ones paying for the development effort
    - Estimatable - user story should be written precisely enough that a developer can estimate how long it will take
    - Small
    - Testable - how you know it is done, and you have built what you said you were going to build; need to know how to test, what it means when all tests pass
        - some things are not testable
- INVEST walkthrough
    - Architectural context: Company email system with a database of contacts
    - User story: As an employee, I want to search for contacts by name so I can message them
    - DoD: Employee can enter a name or partial name into a text box, click search, will see the list of contacts matching the search with check box next to each one, and they can then press a message button once at least one is selected and then a composed email with contacts included appear
    - Independence? Negotiable. Valuable. Estimatable? Testable.
- Specs
    - From requirements to specifications
        - Requirements: describe what functionality is needed, must be comprehensible by the client

        - Specifications: are the technical details about how to achieve the functionality, are written for the engineering team
        - Communicate the same things but to different audiences
        - Key challenge: translating informal (often vague) requirements into the Software Engineering domain
        - Specs can vary in their degree of formalism (many tools)
    - They matter
    - They are hard
        - Need specifications are complete, consistent, concise, and precise

Feb. 25th - Lecture 15: Refactoring

- Agile software process
    - Emergent design: start at the **minimum viable product**, then grow from there, identifying duplication and introducing abstractions as needed to solve duplication
    - Refactor code: rather than doing **big** design first (make pragmatic choices along the way -- this is not a license to write terrible code) AND the architectural spike must come first
    - Technical Debt: poor extensible code
- "Listening" to code
    - Code smell $\rightarrow$violates$\rightarrow$ Design principles
    - Code smell $\rightarrow$motivates$\rightarrow$ Design patterns
    - Design patterns exemplifies design principles
- Refactoring timeline
    - NOT: 2 weeks of every 6 months; when the tests are failing; when you should just rewrite the code; when you fix a bug
    - Opportunistically: recognize a warning sign; before starting a new function; after finishing a new function; ...
- Refactoring
    - predictable and meaning-preserving code transformations
    - meaning = semantic = behavior
- Craftsmanship Manifesto

    - Not only working software, but also well-crafted software
    - Not only responding to change, but also steadily adding value
- Broken Code
    - Code must be able to do 3 things:
        - Its job (execute according to its purpose)
        - Afford change
        - Be understandable
- Technical Debt
    - Design choices that were made in the interest of **time** or **budget**, rather than **technical reasons**
    - Accrue over time and often require broad system **restructuring** to decrease debt introduced by past poor decisions
    - Needs to be allocated on a different "budget" or put into an engineering task and discussed with the customer so that they can decide about its value
    - Some technical debt related user stories can exist, view "users" as a subsystem
    - Deliberate vs. Inadvertent of adding a technical debt; Reckless vs. Prudent of adding a technical debt
        - Deliberate and Reckless (Irresponsible): we don't have time to think about it
        - Deliberate and Prudent (Intentional): deadline is firm, but the **risk** is worth it
        - Inadvertent and Reckless (Incompetent): It **will work** in production without any testing
        - Inadvertent and Prudent (Accidental): We were **lucky**, but next time can do better
- 2 Tricks of Agile
    - Tests: Write tests that ensure that you know that your code is meeting specifications
    - ...
- Motivation for refactoring
    - As the code changes, code issues emerge
    - if it's larger scale, it's called **Technical Debt**

        - if it's a cosmetic issue, it's called a **Code Smell**
        - They are expected in Agile methodologies
        - They make code hard to evolve, difficult to add new features
- Emergent Abstraction
    - Any time trying to make a change, it is difficult because
        - make that change in more than one place
        - changes will result in merge conflicts
        - code is too **obfuscated** to be clearly understood
    - run the risk of introducing bugs
    - To avoid these, we refactor before making changes by introducing an abstraction to either
        - Solve duplication/clone/scattering that causes the multiple change locations
        - Reduce tangling between the pieces of functionality by splitting them up
        - Increase readability to make the code more understandable
- How to refactor
    - Make sure all tests pass, examine the code smell, determine how to refactor this code, apply the refactoring, run tests to make sure nothing breaks, repeat until the smell is gone
    - Magic numbers: any use of an actual number in the code

Feb. 27th - Lecture 16: Code Smell

- Long Method
    - Practically impossible to understand what's going on
    - Tests cannot help with fault localization because code blocks are not isolatable
    - Does not facilitate convenient extension by subtypes
    - Extract clusters of behaviour into new methods
    - Method Design Aesthetics
        - All the same level of abstraction ✓
        - Differing levels of abstraction ×

    - Methods should just do one thing at the **right** abstraction level
- Long Parameter List
    - Create parameter object to replace the parameters with an object
- Comment Misuse
    - Good - Comment technology: necessary for specification and documentation (requires, modifies, effects)
    - Bad - Comment that explain code: these are those that do not refer to specification or documentation (normally anti-patterns); special exception for comments that describe **why**
    - Fixing explanatory comments
        - extract method and give the method a **useful name**
        - named elements **become** the documentation
- Switch on type
    - A conditional varying behaviour based on an object's type
    - Use polymorphism to remove conditionals
    - Each switch-case gets its own class, each class gets its own version of the method, each method gets the contents of the case body
- Identical Methods in Multiple Classes
    - Pull up method refactoring by creating method in superclass, then remove from the subclasses, and add extension points in subclasses
    - Identify the unique and common parts of the methods, extract the common parts, leave the unique parts
- Almost Identical Methods
    - Template Method Refactoring
        - Pull up the duplicate code (or duplicated logic) into the super method
        - The super method calls an abstract helper method that inserts the contents
        - The subclasses provide the implementations for the content method
- Feature Envy
    - Wishing to have access to lots of fields from another class, the method being implemented should probably live in that class

    - Law of Demeter: too many dereferences mean the method is probably in the wrong spot
- Experiential Code Smell Symptoms (issues noticed while changing code)
    - Shotgun surgery: have to make a change in more than one location
    - Divergent changes: changes will result in development collisions
        - when one class is commonly changed in different ways for different reasons
    - Need to extract behaviors into their own classes
- Underpinning Principle of Design
    - Classes not **just** be considered in terms of their apparent responsibility, but must also be centered around their empirically observed relevance

Mar. 4th - Lecture 17: Information Security

- Attacks are increasingly frequent, effective, targeted, sophisticated, profitable, persistent, elusive
- Understanding, Analysis, Mitigation, Validation
- Understanding
    - Terminology
        - Assets: what is being secured
        - Subject: who are the people using the system
        - Policies: the rules associated with the system
        - Threats: The reason we need policies
        - People: Users, ops, dev, malicious actors
        - Process: Architecture, design implementation
        - ...
    - Security Requirements: Confidentiality, Integrity, Availability, Accountability
    - Physical security: keyloggers; System security: denial of service; Network security: man-in-the-middle; Human security: social engineering
- Analysis
    - Threat modelling
        - Who are they?

        - What are they trying to do?
        - Which vulnerabilities are they using?
    - Attack tree
        - Attacker - Goal - Attack
        - Who is doing it? - What are they trying to do?- How are they going to do it?
    - DREAD
        - Damage, Reproducibility, Exploitability, Affected users, Discoverability
    - Attack surface
        - The "size" of your I/O mechanisms
            - The fewer ways in, the easier it is to secure them
        - OWASP definition
            - The number of I/O mechanisms
            - The code securing those mechanisms
            - The data being transmitted
            - The code & mechanisms securing the data
        - Other ...
    - STRIDE
        - Spoofing: pretend to be someone else
        - Tampering: Modifying data
        - Repudiation: Denying performing a task
        - Information Disclosure: Access private information
        - Denial of Service: Halt service
        - Elevation of Privilege: Gaining unauthorized access
- Mitigation
    - Defense in depth
        - A small breach shouldn't become a big one. Need to minimize access within the system
        - Action
            - Don't rely on border security alone
            - Vary mechanism

    - Avoid failure propagation
  - Downside: Adds complexity, redundancy
- Least privilege
  - Coarse-grained privileges allow
    - Malicious access
    - Accidental access
  - Action
    - Finer-grained privileges, roles, access control lists
    - Microservices
  - Downside: More complicated (in code and in usage)
- Separation of duties
  - A single bad actor can cause great harm in a secure system
    - High-impact tasks should be validated
    - Also good for protecting against accidents
  - Action
    - Introduce multiple steps/people for sensitive tasks
    - ...
  - Downside: ...
- Strong authentication
  - If it is easy for an adversary to gain access they will
  - Action: Multi-factor authentication
  - Downside: velocity and complexity
- Non-repudiation
  - System users must be held-accountable
  - Action: atomic, write-only, auditing
  - Downside: audit logs are also sensitive information that could be compromised

- Validation
  - Google security layers
    - Operational security, Internet communication, Storage devices, User identity, Service Deployment, Hardware infrastructure

Mar. 6th - Lecture 18: High Level Design

- Design Principles
    - Pragmatic programmer: eliminate effects between unrelated things by designing components that are self-contained, independent, and have a single, well defined purpose.
    - Principled code arises and emerges
    - Why principled code?
        - Flexible: should handle change (like new features)
        - Understandable: code is shared
        - Maintainable: endure over time (easily find and fix bugs)
- Design Guidance
    - Abstraction - high level concept at the root of all that follows
        - Manage complexity by focusing on key aspects for given task and stakeholder
        - Both too much abstraction or too little inhibit understanding
        - Useful for discussing viewpoints that makes sense for individual stakeholders
    - Decomposition - mechanism for ideating about abstractions
        - Mechanism for breaking down complex description into more manageable pieces
        - Goal is to make common tasks simple while not prohibiting exceptional tasks
        - Can be done both top down or bottom up
    - Information hiding - how programs leverage abstraction
        - Hides implementation details from high level interfaces
        - Separate that which varies from that which stays the same
        - Separate names from implementations (aka API from impl)
    - Encapsulation - language approach for implementing information hiding
        - Mechanism for implementing abstractions in a program
        - Usually through the use of language interfaces
        - Captures data and behavior and separates these from their implementation
- Coupling and Cohesion

- Coupling intuition
    - Coupling is the relationship between modules
    - If we make a change in one module, what will the impact be on other modules within the system
    - Modules should not depend on each other's internals
    - Easy to couple modules in a way that violates architecture
- Kinds of coupling
    - Data coupling: coupled via data, primitive types
    - Stamp coupling: coupled via data, data structures
    - Control coupling: coupled via control flags
    - Global coupling: coupled via data, global variables
    - Content coupling: coupled via internal access
- Cohesion intuition
    - Cohesion is the relationship between elements within a module
    - Modules should only contain functions that belong together
    - Often observable by the data **modified** by the functions
    - ...
- Kinds of cohesion
    - Function cohesion: function does one thing
    - Sequential cohesion: doesn't do one thing, but the tasks are related by data, the sequence is important (output from one step used as input in next step)
    - Communicational cohesion: the sequence is not important, but the tasks are still related by data (tasks grouped in method only because of data)
    - Procedural cohesion: the tasks are related by control flow, and the sequence is important
    - Temporal cohesion: the tasks are related by control flow, but the sequence is not important (functionality grouped because of execution timing)
    - Logical cohesion: the tasks are not related, but they are similar (functionality combined for task, regardless of intent)
    - Coincidental cohesion: the tasks are not related, and they are not similar (functionality just arbitrarily grouped)

        - Reduce coupling, increase cohesion
- SOLID - modular design principles and heuristics
    - S - Single Responsibility Principle

      O - Open/Closed Principle

      L - Liskov Substitution Principle

      I - Interface Segregation Principle

      D - Dependency Inversion Principle
    - S Principle
        - A class should have only a single responsibility. Specifically, only one potential change in the system's specification should be able to affect the implementation of the class
        - Divergent changes often signal SRP violations because they suggest that one class is doing the work of more

Mar. 11th - Lecture 19: SOLID

- Review
    - Coupling - between classes
        - e.g. fields, inheritance/composition, method parameters
        - want **low coupling** - low interdependence between classes
        - in practice: a change in one place does not trigger a change in many (shotgun surgery)
        - code smells: feature envy, shotgun surgery
    - Cohesion - within a class between elements, all methods relate to one area of the specification
        - want **high cohesion**
        - in practice: multiple parts of the spec aren't in class (divergent changes)
- SOLID
    - O - Open/Closed Principle
        - A class must be **closed** for **internal change** (no need to change its code)
        - It also must be **open** for **extension** (subclasses can specialize straightforwardly)

        - When designing classes, do not plan for brand new functionality to be added by modifying the core of the class
        - Instead, design the class so that extensions can be made in a modular way, to provide new functionality by leveraging the power of the inheritance facilities of the language, or through pre-accommodated addition of methods
    - L - Liskov Substitution Principle
        - Subtype requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T
        - If S is a subtype of T, then objects of type T (supertype) in a program may be replaced with objects of type S (subtype) without altering any of the desirable properties of that program.
        - Substitutability: a subclass should not break the expectations set by its superclass
        - Precondition rule: preconditions should not be strengthened (for the same inputs), OK to widen

          Postcondition rule: postconditions should not be weakened (for the same inputs), OK to narrow

Mar. 13th - Lecture 20: SOLID (Continued)

- SOLID
    - I - Interface Segregation Principle
        - Many client-specific interfaces are better than one general-purpose interface
        - Clients should not have to depend on unnecessary methods
        - No implementation should be forced to provide methods that do not fit into their abstraction (classic pathway to SRP and LSP violations)
        - Solution: move towards role-based interfaces
            - clients need only know about the methods that are of interest to them
            - relates strongly to the concept of cohesion
            - extract interface refactoring

- D - Dependency Inversion Principle
    - Depend upon means **declaring** an entity of that type
    - Depend upon **abstractions**, do not depend upon implementations
    - high-level modules **should not** depend on low-level modules. Both **should** depend on abstractions
    - Abstractions **should not** depend on **details**, details should depend on **abstractions**
- Design Principle Recap
    - eliminate effects between unrelated things by designing components that are self-contained, independent, and have a single, well-defined purpose

Mar. 18th - Lecture 21: Microservices and API Design

- API Design
    - Application Programming Interface

      a way for two or more computer programs to **communicate** with each other
    - Information hiding
        - only expose necessary functions
        - abstraction hides complexity by emphasizing essential characteristics and suppressing details
        - caller should not assume anything about how an interface is implemented
        - effects of internal changes are localized

        -
          ```
          interface BarometricElevation {
              getElevationInMeters(pressureInMB:
          number): number;
          }
          ```
    - Principles
        - Do one thing and do it well

          ```
          getPaidUsersAndSortByName() -> getUsers(paid:
          boolean, sortIndex: ...)
          ```

    - APIs should be as small as possible but no smaller
        - when in doubt, leave it out

        -
          ```
          // Bad
          getUsers(paidOnly: boolean, sortKey:
          Users.Keys)

          // Best
          getPaidUsers()  // sortKeys not
          needed, client sorts
          ```
    - Implementation should not impact API
        - leaking API details is a fundamental mistake
    - Minimize access, increases independence
        - API should **never** expose internal implementation details
    - names matter
    - documentation matters
- Two main questions

  What is the goal, Who will be using
- Technical considerations: easy to use, hard to misuse
- How to write an API
    - start with one-page spec to create use cases
    - talk to as many stakeholders as possible
    - ...
- Designing API methods
    - Do not make client do anything that could be internal
        - look for clones in sample client systems
        - provide basic type access
    - Users should not be surprised by API behaviour
        - use consistent params and types
        - avoid returns that demand exceptional handling
    - Fail fast: report errors as soon as possible
        - compiler better than runtime

            - first error rather than last at runtime
        - Designing for constant change
            - think concretely about what parts of the system are likely to change in the short and medium terms
            - while APIs are forever, their internals will change more than you can usually imagine in advance
    - Programmatic Usability
        - APIs are affordances used by developers for understanding
        - API design forces you to think from a client's point of view
        - Good APIs encourage reusability
        - Broad valuable APIs can be widely used
    - Principles of programmatic usability
        - Good visibility: possible actions and states must be exposed
        - Good model: helpful, consistent, and complete abstractions clarify the correct model of the system
        - Good mapping: natural mapping between actions and results
- Microservices
    - Big interface into smaller separate APIs
    - Microservice architecture:

      multiple **loosely coupled** services work together, each service focuses on a **single purpose**, and has a **high cohesion** of related behaviors and data
    - When we **model** microservices, we should be **disciplined** across all three design principles. It is the only way to achieve the **full potential** of the microservice architecture. Missing any one of them would become an anti-pattern
    - "Syndromes"
        - **Poorly modelled** microservices do more harm than good (especially when having more than a couple)
        - Lack of **observability**, making it difficult to triage performance issues or failures
        - When facing problems, team tend to create new service instead of **fixing existing one** (even though the latter may be better option)
        - even though services are **loosely coupled**, lack of **holistic picture** of the whole system could be problematic

Mar. 20th - Lecture 22: REST APIs

- REST - architectural style, a set of guidelines or principles for a Web API
- REpresentational State Transfer API (REsource State Transfer)
- RESTful Design Principles
    - Localization of reasoning
    - Reduction in the propagation of change
- Pervasive REST - favoring **reuse**
    - as a means to reduce complexity
    - as a means to improve productivity
    - as a means to increase reliability
    - as a means to encourage / enable reuse
- Library Scaling
    - two extremes: large, feature-laden, components; small, simple, components
    - large components are hard to adapt
    - small components have complex cost/benefit implications
- SOAP - Simple Object Access Protocol

  RPC - Remote Procedure Calls

  Pro: Simple to program, behaves like local calls

  Con: The protocol must be pre-defined. The client and the server must agree on methods/data types

  Delocalization of changes - server changes propagate to client
- REST: Nouns and Verbs
    - A HTTP request = HTTP method (Verbs) + URI (Nouns)
    - Verbs: the actions that can be taken on the system's resources (nouns)
        - Verbs are HTTP methods: GET, POST, PUT, DELETE
        - every resource must support all four verbs
        - Idempotency: will performing a VERB multiple times have the same result for server resources as performing it once?
            - POST: is not idempotent, always makes new resources
            - Other: idempotent, resources are consistent

        - Nouns: the resources (or objects) of our system, contained in the URI (or URL)
            - Provide abstraction of resources and data
            - Shows the relationship between different nouns
            - in the Request URI/URL
            - are the application's resources
            - are always plural (issues, not issue)
            - order of the nouns describes the relationship between different resources
- Design constraints
    - apply principled design process to commonly occurring challenges facing online applications
    - needed to support networked sets of loosely coupled services
    - constraint: replication needed to support caching
    - constraint: uniform interface allowed simple mechanisms
    - constraint: statelessness required for reliability
    - constraint: connectedness supports discoverability
- REST Design Principles 2
    - Resource orientation: emphasizes **separation of concerns** between a resource and the actions a system might perform on the action
    - Uniform interface: having a consistent, predictable, and simple mechanism for marshalling requests and responses allows for clients and servers to independently evolve
    - Statelessness: all client requests need to contain all of the information needed to process a response
    - Self-descriptive messages: messages should contain information to help clients reason about how to process a response; enable follow-up actions for a resource; enable correctedness, where clients do not need to guess the APIs
- Design Flexibility
    - URIs define resources, but there are many ways to pass additional data to the server
        - PUT: /segments/{id}/starred

          what *user* is applying the star

        - DEL
            - Is the *user* performing the action permitted to delete this
        - Potential options

          headers, explicit parameters, request bodies
- Enabling API evolutionary robustness
    - imagine response object, {name: `cpsc autobot`}
    - in the future, need to differentiate first and last names, one possible schema: {first: `cpsc`, last: `autobot`, name: `cpsc autobot`}
    - REST resources as much as possible strive for backwards compatibility since frontend and backends are loosely coupled and may not be simultaneously updated

Mar. 25th - Lecture 23: REST API +

- REST API
    - Versioning
        - Encode date of API version
        - Encode version in request
        - Encode version in parameters
        - Specify the version in the header
- Design Patterns - Introduction
    - Design pattern is a tried and true solution to a commonly encountered problem - generalizable solutions to common problems
    - These were architectural idioms, to guide architectural design
    - Why not design patterns?
        - easy to blindly "force" a pattern into a context where it does not make sense
        - abstraction layers have costs (mainly understandability) that must be balanced with benefits
        - abstraction layer required for a pattern ...
    - Why design patterns
        - **leverage** existing design knowledge
        - enhance **flexibility** for future change

        - ease communication by using a shared **vocabulary**
    - Design Pattern categories
        - Creational design: object creation mechanisms
        - Structural design: ...
        - Behavioral design: ...
    - 2 important OO foundational technologies
        - abstraction: creating a more generic classes/interfaces
        - polymorphism: using dynamic substitution to make for code that is more reusable and/or obvious ...
- Observer pattern
    - Multiple watchers, one watchee
    - abstract the watchers to subclasses of Observer, abstract the watchee to subclass of Subject/Observable
- Composite pattern - structural pattern

Mar. 27th - Lecture 24: Design Patterns

- Factory
    - working on an implementation for managing Cars, but we don't always know what kind of cars we are dealing with
    - the problem will be
        - constructor doing work outside the scope of class
        - making a car is complicated and distracts from what the car is supposed to do
        - SRP violation
        - constructor is prone to changes (divergent changes)
    - Extract CarFactory, CarFactory creates and returns Car
    - what if one doesn't know the exact type of Car one wants at any particular time? (obliviousness)
        - abstract `SmallCar` and `BigCar` into `Car`, and abstract `SmallCarFactory` and `BigCarFactory` into `CarFactory`
        - where the each car factory delivers its own type of cars
- Singleton

    - creating an issue tracker application where different components need to access a database. need to ensure there is only one database
    - hides the constructor, so only the singleton class can call it
    - statically provides access to that instance
    - saves an instance of the class to a static variable
        - only creating instance if needed
    - it is a design pattern that restricts the instantiation of a class to one object. this is useful when exactly one object is needed to coordinate actions across the system
    - often replaced by dependency injection
- Strategy
    - Creating an app for file compression, and want to support different compression algorithms
    - Applying strategy: separate each compression into their separate class, allow new algorithms to be added without impacting clients, expose explicit interfaces for which we can provide new algorithms as new requirements are added
    - enhanced DIP: Service does not depend on concrete Strategies
    - enhanced OCP: new Strategies can be added and Service is oblivious
    - Responsibility of knowing about concrete strategies moved to Factory
- State
    - original approach: use switch-case for Phone to handle transition - **switch-on types**
    - often used for problems that are natural to define using state machines
    - create a hierarchy of possible states
    - Store a reference to one of the state objects representing the current state
    - Delegate the state-related work ...
    - State $\approx$ extension of Strategy

Apr. 1st - Lecture 25: Design Pattern

- Adapter
    - implementing a web application for a music store, have two separate data sources for records, they store records in JSON or CSV format, however, want to return a shared format, a record
    - Intent: an "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed
    - convert the interface of a class into another interface that clients expect, adapter lets classes work together that couldn't otherwise because of incompatible interfaces, wrap an existing class with a new interface
    - One specific **adapter** for each **adaptee**
    - Assume we have a legacy interface, we would still like to segregate the interfaces, we can solve the problem by applying the Adapter pattern
- Decorator
    - decorator is similar to composition
    - Consider we have base coffee: medium-roast, dark-roast, decaf, espresso, etc
    - And we wish to add things to these base coffee: whip, mocha, soy, non-fat, etc
    - we can abstract a class of `BeverageDecorator`, where it and `Decaf`, `MediumRoast`, `DarkRoast` can extend `AbstractBeverage`
    - The `BeverageDecorator` can then be extended to each decorator `Whip` and `Mocha`, where they use `AbstractBeverage` instances (as base coffee)
    - It dynamically add responsibilities to objects
        - enable features to be added to object instances instead of **all** instances of a class
        - ...
    - analysis
        - single responsibility
        - open/close
        - using composition in place of inheritance

        - useful for layering
- Facade
    - implementing a query engine that lets clients add a dataset from an archive, remove a dataset, list all datasets, and performing a query based certain conditions
    - facade is a class that we add to expose certain actions to the public
    - design process
        - identify the main responsibilities of the subsystem (often violate SRP, try to provide a subsystem's breadth of behaviors)
        - determine if common or simple actions require complex collaboration of multiple classes within the subsystem
        - identify abstractions that would allow the subsystem to encapsulate these behaviors more directly
        - expose abstractions through the Facade to enable subsystem internals to be easily evolved (enables OCP, often via DIP, to allow evolution)

Apr. 3rd - Lecture 26: GUI Design Patterns

- GUI (Graphical User Interface)
    - A **visual way** of interacting with a computer
    - Design principles still apply: low coupling, high cohesion; SOLID; abstraction, decomposition, information hiding, encapsulation
    - Example of GUI Patterns: MVC, MVP, MVVM, React (Component based)

      From left to right, goes from thin to thick
- Start analysis with an example: writing software for a simple step-counter, the system should allow users to set and update a daily target step goal
- MVC (Model - View - Controller)
    - View: handles layout and display (HTML/CSS)
    - Model: manages data and business logic (Database)
    - Controller: routes commands to model and view (Javascript)
    - Model updates view, View sends input from user to controller, Controller manipulates model
        - Controller sometimes updates view directly

- Model
    - contains application data

      often persisted to a backing store

      defines the data structure

      contains business logic (rules of the application)
    - does not

      know how to present itself, but is able to ensure its own integrity

      care what is displayed and when
- View
    - Renders the Model for the user
    - allows the user to manipulate the data (change views, CRUD)
    - can be reused (configurable to display different data)
    - contains presentation logic (knows a bit about what it is displaying)
    - does not

      decide when to update itself

      validate data
- Controller
    - Glues Model and View together
    - Responds to User interactions
        - Updates the model when the user manipulates the View
        - sometimes updates the view when the Model changes
    - Can contain validation logic
- Separation of concerns
    - concerns are often partitioned to manage complexity

      The fidelity of a concern will vary for different levels of abstraction

      SRP & ISP are both direct derivations of the importance of identifying and isolating concerns in a design
    - MVC is a compound design pattern that is commonly used to separate concerns in user-facing systems

        MVC provides a well-understood layer of organization on a design

    - Motivation
        - separates the business logic from the display
        - UI changes more frequently than business logic
        - the same data is often displayed in different ways and the same business logic can drive both
        - designers and developers are often different people
        - testing UI code can be difficult and expensive
    - Interaction mechanism
        - View sends "changes" to Controller
        - Controller "updates state" and tell Model
        - Model "notifies" View of state changes (pull-based Observer pattern)
        - View "retrieves" state from Model
    - Benefits
        - decouple view from model
        - support multiple views
        - maintainability
        - split teams
        - testability
    - Tradeoffs
        - complexity
        - efficiency
- MVP (Model - View - Presenter)
    - Improve decoupling with explicit interfaces, enhance testability by simplifying views, further separate designers from developers
    - leveraged by Java Swing and .NET
    - Presenter: view's presentation logic
    - View sends input from user to Presenter, Presenter updates Model, Model does state-change events and notify Presenter, then Presenter updates View
    - Presenter

        - Glues `Model` and `View` together, updates `View` when `Model` changes, updates the `Model` when the user manipulates the `View`, translates between the primitives...
    - Benefits and Tradeoffs
        - Decoupling of `View` and `Model`

          split teams, testability (reduce UI testing)
        - Clearer separation of concerns between `View` and `Presenter`

          DIP used to decrease coupling between layers

          SRP and OCP better enforced by clearer roles
        - less complex than MVC (fewer events)
- MVVM (Model - View - ViewModel)
    - `View` and `ViewModel` does data binding (two-way communication)
    - Data binding links the `View` and `ViewModel`
    - Binder, framework-managed communication automation
    - Reduces coupling between the `View` and `ViewModel`
- Component Based
    - "encapsulate individual pieces of a larger user interface into self-sustaining, independent microsystems"
    - each component contains

      View: HTML Template with data binding

      Controller(ish): presentation logic necessary to render view, sometimes business rules sneak in here as well
    - Component libraries
        - Open source components (drag and drop UIs)

          easy to maintain, quick to set-up (add library), consistent: work across different browsers (less testing), accessibility, polished UI components
        - e.g. AntDesign, MaterialUI
    - Component communication
        - Parent component "Data binding (model updates)" Child component
        - Child component "User Events (callback functions)" Parent component

- Smart and Dumb components
    - Smart component: container components, contain business logic, state management, presentation logic
    - Dumb component: presentational components, only render UI based on input data
- Benefits and Tradeoffs
    - View is isolated, encapsulated, and small

      improves testing, saves development time, increased performance when updating the UI, encourages SRP, increases consistency across the application, enhanced reusability
    - adds complexity
    - easy to over-engineer (create unnecessary components)

Apr. 8th - Lecture 27: Synthesis

- REST API example
    - Workday enrolling students in classes: POST (create), PUT (update), GET, DELETE
        - **`/departments/{:id}`**: manage departments
        - **`/departments/{:id}/courses/{:id}}`**: manage courses
        - **`/departments/{:id}/courses/{:id}/students`**: manage students within a course
    - Statelessness: all the information that the server needs to handle the request is contained within the request
    - Idempotency: multiple requests will result in the same state/behavior of one request
    - Connectedness: response should contain relevant information for the client to make further/future requests
- Term coverage
    - languages and async, testing and testability, assertions and fuzz testing, process and teamwork, specs and user stories, refactoring and code smells
    - high-level design (principles), low-level design (principles), REST, Automation and DevOps, ...