Lecture 1

September 9, 2020 5:04 PM

Five essential tasks in software engineering:
Specification/Requirements:

- Specification is a description of a software system to be developed
    - Functional: use cases, interactions the software must provide
    - non-functional
- Requirement

Design:

- Different aspects
- Guides the development team in building a software product

Implementation:

- Converting the design into an executable system
- Must address some fundamental principles (requirements)

Verification & Validation

- More than testing
- Make sure that a system conforms to the specification and meets the requirements
- Verification: does the software meet the specification?
- Validation: does the specification capture the customer's needs? Is it what the customer wants?

Maintenance & Evolution (most of the time)

- Modification of a software product after delivery
- Four main types:
    - Corrective: fixing errors
    - Perfective: implementing new/changed user requirements
    - Adaptive: modifying the system to cope with changes in environment
    - Preventive: increasing maintainability or reliability

A software process is who is doing what, when, and how in the development of a software system

Process models:

- Code-and-fix
    - Good for small projects and short-lived prototypes
    - Hard to accommodate changes
    - No good way for assessing risks
- Waterfall
    - Following the steps
    - Advantages
        - Suitable for projects that are well understood but complex
    - Disadvantages
        - Requires much planning up-front (not easy)
        - No sense of progress until the end
        - Delivered product may not match need
- Staged Delivery
    - procedure
        - Waterfall-like beginnings (requirements and design upfront)
        - Short release cycles (plan, code, test, release, repeat)
        - Delivery possible at the end of any cycle
    - Advantages
        - Intermediate deliveries can have feedback
        - Can ship at the end of any release cycle
        - Integration problems are visible early
    - disadvantages

        - Requirements must be known up-front
- Evolutionary prototyping
    - Similar to staged delivery
    - Requirements are not know up-front but discovered by feedback
    - Advantages
        - Participatory design
        - Useful feedback loops
        - Practical and widely used
    - Disadvantages
        - Spec must be flexible
        - Requires customer involvement
        - Planning, schedule, feature set are hard to estimate
- Spiral
    - Risk-oriented variation of evolutionary prototyping
    - Need to identify and solve problems with the highest risk at each iteration
    - Advantages
        - Early indication of problems
        - Decrease risks
        - Appropriate at the beginning
    - Disadvantages
        - Must assess risk
        - Tasks are changed frequently
- Agile
    - Customer collaboration
    - Responding to change
    - Value individuals and interactions
    - Practices
        - Continuous integration (CI)
        - Scrum
            - Scrum member rotate through roles (especially product owner) each iteration.
            - Sprint (iteration) is the basic unit of development in scrum
                - It is restricted to a specific duration (usually two weeks)
                - Sprint planning: communicate the scope of work for the sprint
                    - What have you completed
                    - What is blocking in your way
                    - What will you do next
            - Challenges
                - Team members are geographically dispersed or part-time
                - Members have very specialized skills
                - Products with many external dependencies
                - Products with regulated quality control
        - Test-driven development
        - Pair programming

Keeping track of progress

- Task board (GitHub has an native solution)
    - Can have priority points based on difficulty
- Burndown chart
    - Time-work remaining chart
- Sprint review and retrospective
    - Review:
        - Review the work that was completed and planned but not completed
        - Present completed work to the stakeholders
        - Team and stakeholder collaborate on what to work next
    - Retrospective

- What went well during the sprint?
- What could be improved?

Lecture 2

September 16, 2020 5:07 PM

Processes: Main Message

- Customize the processes depending on the product, organizational culture, team structure, needs, etc.
- Follow processes, but do not over-emphasize process over product

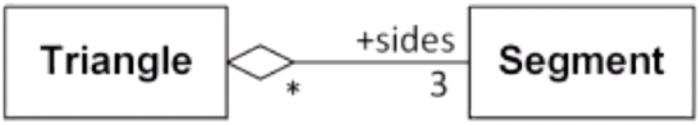
This is how we should develop a software. (SDLC)

UML: Unified Modeling Language
It's a common standard of software development, independent of development process and programming language.

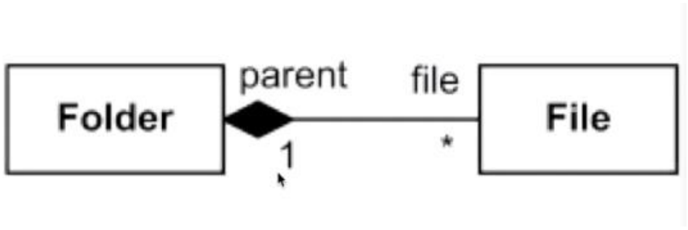
UML diagrams are used for capturing different aspects of design:

- Requirements
- Systems architecture
- Program design

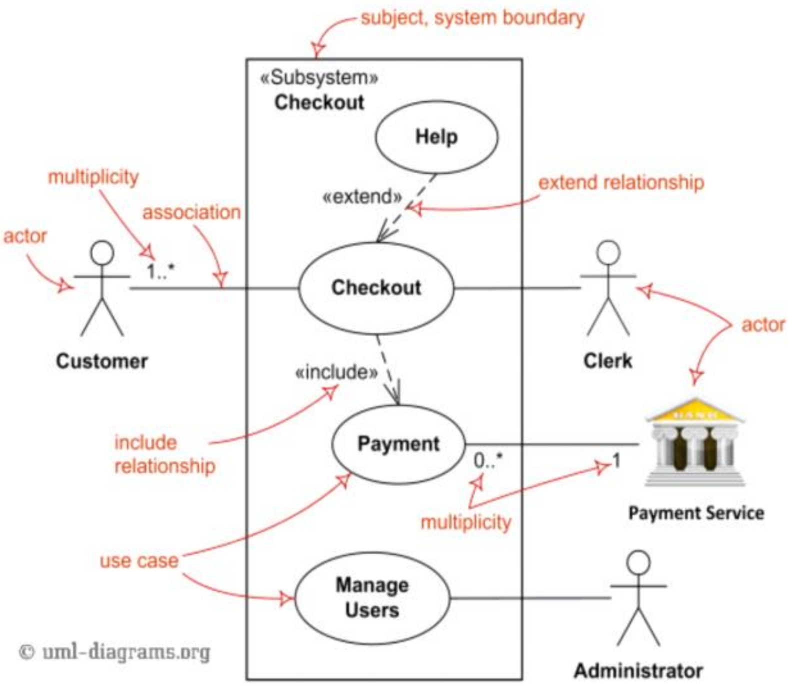
UML diagrams types

- Class:
  - use rectangle showing the name of the class, data structure, attributes and operations
  - Shows the relationships between classes in a system
  - Visibility symbols
    - +: public
    - -: private
    - #: protected
    - ~: package
  - Object:
    - An instance of a class, can optionally contain values of fields
    - Written in a rectangle [object name; class name](not necessary to have both)
  - Interfaces:
    - Specifies a contract
  - In UML, both Classes and Interfaces are instances of an abstract class called Classifier
  - Relations
    - Generalization
      - Relationship between a more general class (super class, parent) and a more specific class (subclass, child)
    - Association
      - Role: association end name
      - Multiplicity: multiple class can associate to one class, and one class can associate to multiple classes
      - Types:
        - Binary
          - Aggregation: a weak form of whole/part
            - 

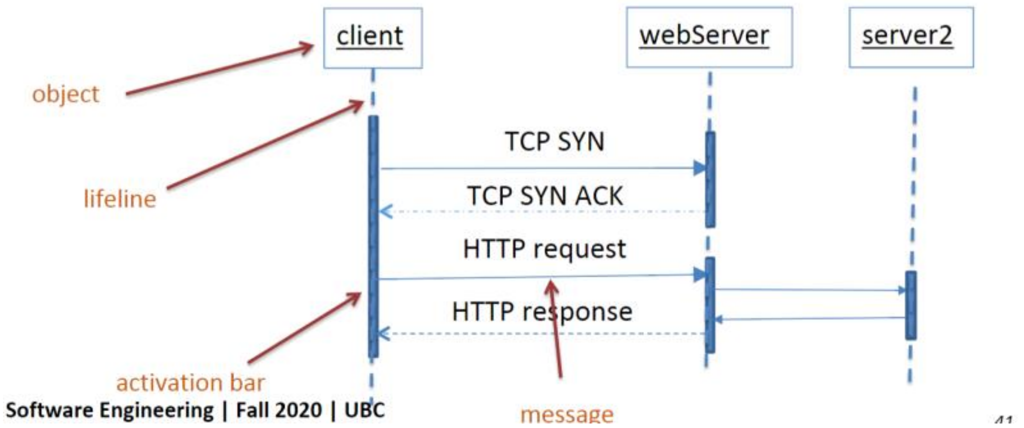
            - One segment can belong to multiple triangles
            - Triangles must have 3 segments
          - Composition: a strong form of whole/part

- 

- One File belongs to one folder, and doesn't have the right to live by itself, and one file can only live in one folder.
- Folders can have multiple files
- Only one end of association can be marked as aggregation/composition
- They should form an acyclic graph, since no instance should be part of itself directly or indirectly.
- N-ary
- Dependencies
- Case diagram:

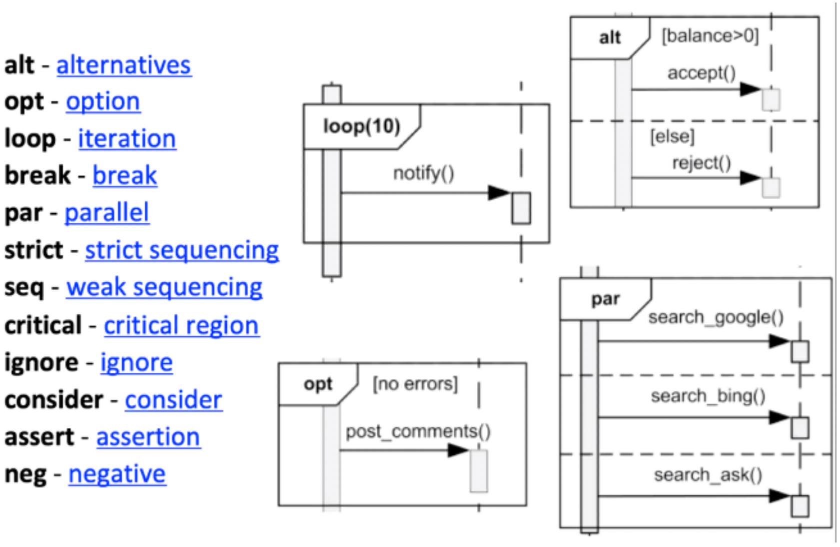


  - Represents the user's interaction with the system (use cases)
  - Subject: boundaries of the system
  - Actors: shapes with names (nouns)
  - Use cases: ellipses with names (verbs)
  - Line associations: connect actors to use cases
    - Multiplicity
  - Relationships
    - users
      - Generalization
    - Use cases
      - Include: A includes B, then B must be executed in/with A
      - Extend: A extends B, A may/may not be executed before B
      - Generalization
- Sequence diagram:

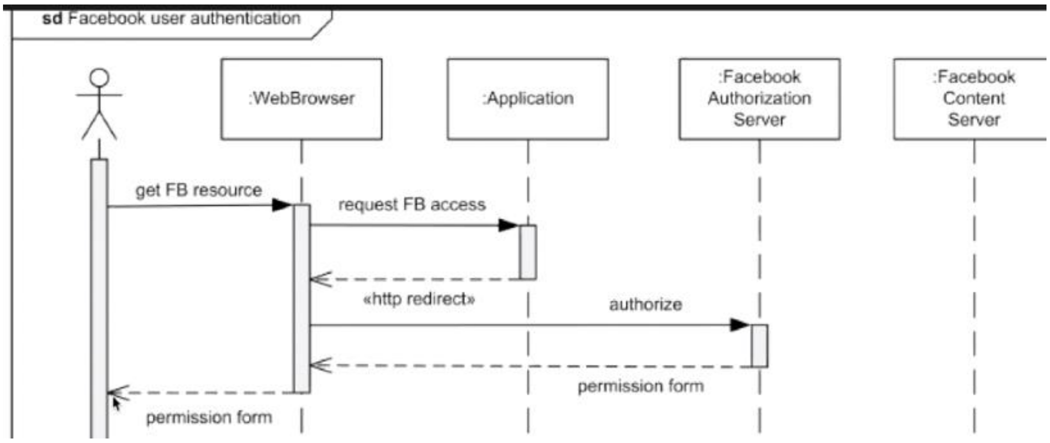


Software Engineering | Fall 2020 | UBC

- Represents the interactions of the objects in a system
- Consider small, discrete pieces of systems
- Messages
  - Synchronous call sends a message and wait for the response
  - Asynchronous call sends a message and proceeds immediately without waiting for a return value
- Execution specification represents a period in the participant's lifetime
  - Can be overlapped
- Interaction fragments:
  - Allows to call another interaction
  - Good for simplifying large and complex systems, and reusing interactions



alt - alternatives
opt - option
loop - iteration
break - break
par - parallel
strict - strict sequencing
seq - weak sequencing
critical - critical region
ignore - ignore
consider - consider
assert - assertion
neg - negative

- Eg.
  - 

  - User tries to get FB resource by web browser (synchronous)
  - Web browser requests FB access (synchronous)
  - The application sends an http redirect back to the web browser (async)
  - Web browser tries to authorize on the FB server (sync)
  - FB send back the permission form to web browser (async)
  - Web browser shows the form to the user (async)

Lecture 3

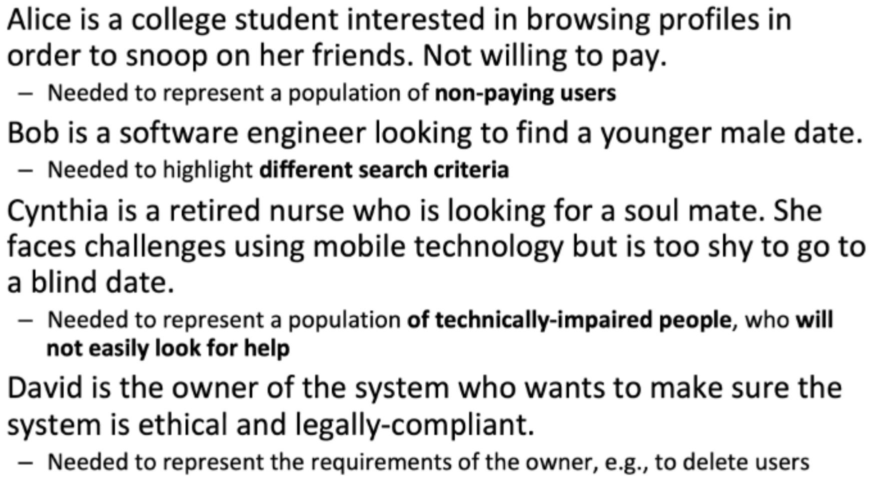
September 21, 2020 2:59 PM

Requirements specify what to build (not how to build it)

- Functional: actors and actions
    - What the users can do
- Non-functional: performance, safety, security, scalability, dependability, reusability, portability

How to build requirements:

- Access to users is important
    - Talk to users
    - Ask questions to dig for requirements
    - think about why, not just what
    - Allow requirements to change later
- Personas
    - Think about typical users of a system
    - Personas should be different from each other
    - Pros:
        - Help understand the customers and satisfy customer problems
        - Align the stakeholder in the entire company
    - Cons:
        - May lead to false sense of understanding
        - Biases on the developer perception
    - Example (online dating system)



Alice is a college student interested in browsing profiles in order to snoop on her friends. Not willing to pay.

- Needed to represent a population of **non-paying users**

Bob is a software engineer looking to find a younger male date.

- Needed to highlight **different search criteria**

Cynthia is a retired nurse who is looking for a soul mate. She faces challenges using mobile technology but is too shy to go to a blind date.

- Needed to represent a population **of technically-impaired people**, who **will not easily look for help**

David is the owner of the system who wants to make sure the system is ethical and legally-compliant.

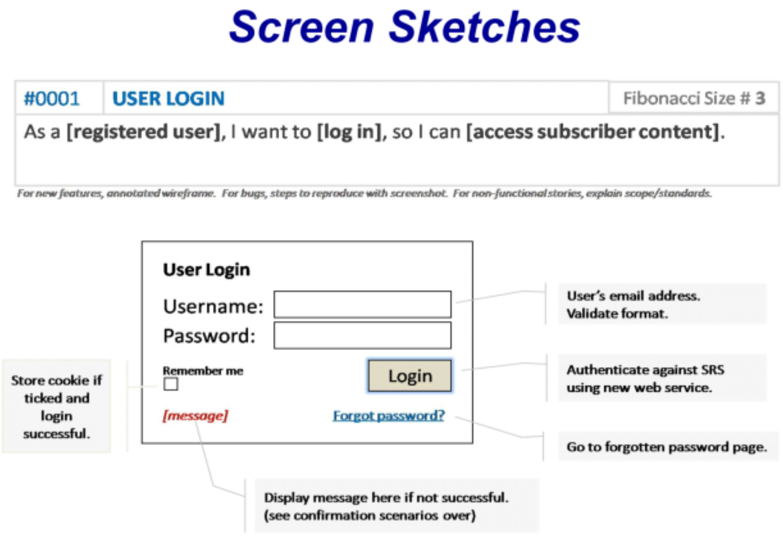
- Needed to represent the requirements of the owner, e.g., to delete users

How to document requirements:

- Non-functional requirements
    - Specific and measurable
    - Write down in a list
    - Can vary for different devices to fit users
- Functional requirements
    - Document
        - Detailed and long (rigid)
        - Includes: preface, introduction, glossary, user requirements definition, system architecture…
    - Prototype
        - Evolutionary prototype
            - Will become deliverable system
        - Throw-away prototyping
            - Just used for defining the specification and thrown away
            - Throw away because the system is poorly structured and difficult to

            maintain
      - □ Pros:
        - ◆ Clear and easy to understand
        - ◆ Appealing to the users
        - ◆ Useful for parts of systems that's hard to describe
      - □ Cons
        - ◆ Non-functional requirements are hard to express
        - ◆ Some functional requirements are difficult to prototype
        - ◆ Has no legal standing as a contract
        - ◆ Time consuming
  - ○ User stories:
    - ■ High level definition of a requirement
    - ■ Contains just enough info so that the developers can produce a reasonable estimate of the effort to implement it
    - ■ Format:
      - □ As persona (a role), I want sth, so that benefit
  - ○ Use cases:
    - ■ Focus on behaviors to meet the user's needs
    - ■ Actors are not personas
      - □ Multiple personas can be a single user
    - ■ Add more info (relationships between actors, use cases)

Screen sketches



Screen Sketches

#0001	USER LOGIN	Fibonacci Size # 3
As a **[registered user]**, I want to **[log in]**, so I can **[access subscriber content]**.		

For new features, annotated wireframe. For bugs, steps to reproduce with screenshot. For non-functional stories, explain scope/standards.

User's email address. Validate format.

Authenticate against SRS using new web service.

Go to forgotten password page.

Store cookie if ticked and login successful.

Display message here if not successful. (see confirmation scenarios over)

Dating system example
Use case: register, setup profile, make some of the profile to be private, search based on specific requirements/filter, direct messages, Like/dislike other users, ban users, upgrade the membership.
Actors: users (register, setup profile, search based on filter, direct message, like/dislike users), system owner (ban, browse users)

Lecture 4

September 23, 2020 5:45 PM

System modules:

- Use nouns, not verbs
- Break a large system down into progressively smaller components or classes that are responsible for some part of the problem domain

Lecture 5

September 28, 2020 2:37 PM

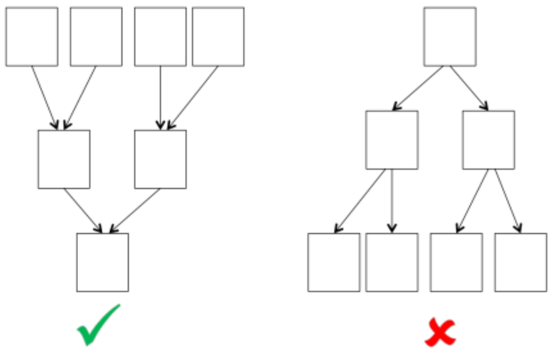
Single responsibility principle:

- Every module should have single responsibility
- Responsibility should be entirely encapsulated by the module
- All module functions should be aligned with that responsibility

Low Coupling/High cohesion principle

- Cohesion: degree to which the elements of a module belong together
  - Related code should be close to each other
- Coupling: the degree to which the different modules depend on each other
  - Modules should be independent

High Fan-in/ Low Fan-out principle

- Have a module used by many others (fan in)
- Do not use many other modules (fan out)
  - High fan-out lacks cohesion
- 

Principle of least knowledge

- Keep only the info and resources absolutely necessary for the module
- Module should assume as little as possible about the structure or properties of any other modules

Do not repeat yourself:

- Implement all functions once and only once

Keep things simple

Module interfaces:

- Only the concept with use cases, not the detail implementation
- Identify input and return value
- Return value from one method should be an input to the next method
- Collect info from multiple use cases
  - Completeness
- Meaningful and consistent names
  - Either remove or delete
- Think about single responsibility, coupling/cohesion, fan-in/fan-out

Note:

- Architecture and high-level design are interchangeable
- Low-level design: detailed design of individual modules
- Modules, subsystems, components are interchangeable

Architecture

- It is a big picture of high-level modules and their interactions
  - Interfaces and communication protocols
  - Frameworks, tools, and languages
  - Database and data structures
  - Design of the main algorithms
  - Security mechanisms
- Architectural pattern: stylized description of good design practice, based on experience
  - Often a complete system has a combination of architectural styles
  - Layered architecture
    - Android software can be layered
  - Client-server architecture
    - Android itself is not a client-server architecture.
  - Pipe-and-filter architecture
  - Model-view-controller
    - Has three layers: model (data), controller (logic), view (user representation of data)
  - Message bus
    - A software system that sends and receives messages using multiple channels

Lecture 6

September 28, 2020 9:33 PM

Patterns and principles

- Principles: guideline to follow, regardless of what patterns we are using
  - Don't repeat yourself
  - Single responsibility
  - Separation of concerns
  - Independence, high fan-in/low fan-out
  - Least knowledge
  - Make it simple (KISS)
- Patterns: something useful to follow when designing, it satisfies most principles
  - Layered architecture
  - Client-server
  - Pipe-and-filter
  - Model-view-controller
  - Message bus

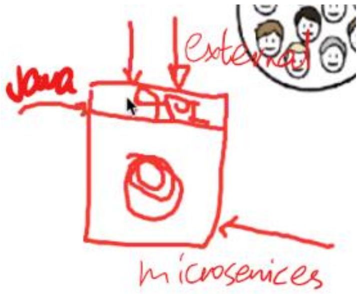
Microservices are used for backend development, split the backend into multiple independent components

- Developed, deployed, scaled independently with different languages/technology
- Communicate over lightweight interfaces
- Characteristics
  - Organized around business capabilities, one service per business capability
  - Loosely coupled (have few interfaces)
  - Owned by a small team
  - Independently deployable
  - Highly maintainable and testable
- At runtime
  - Managed by container-orchestration system
- Why do we need microservices
  - Agile development means more speed and independence
  - Cloud allows companies to scale individual services up/down
  - Technology: docker, kubernetes
- Challenges
  - Complexity shifted outside the code
  - Performance
  - Security
  - Framework diversity
  - Logging, monitoring and distributed tracing

Microservice is not library, it is an component, it can use API to talk with other microservices

API (Application Programming Interface):

- Is a style defining an interface, not a library
- It can belong to a class, a library, or a microservices
- Libraries expose API to the external world

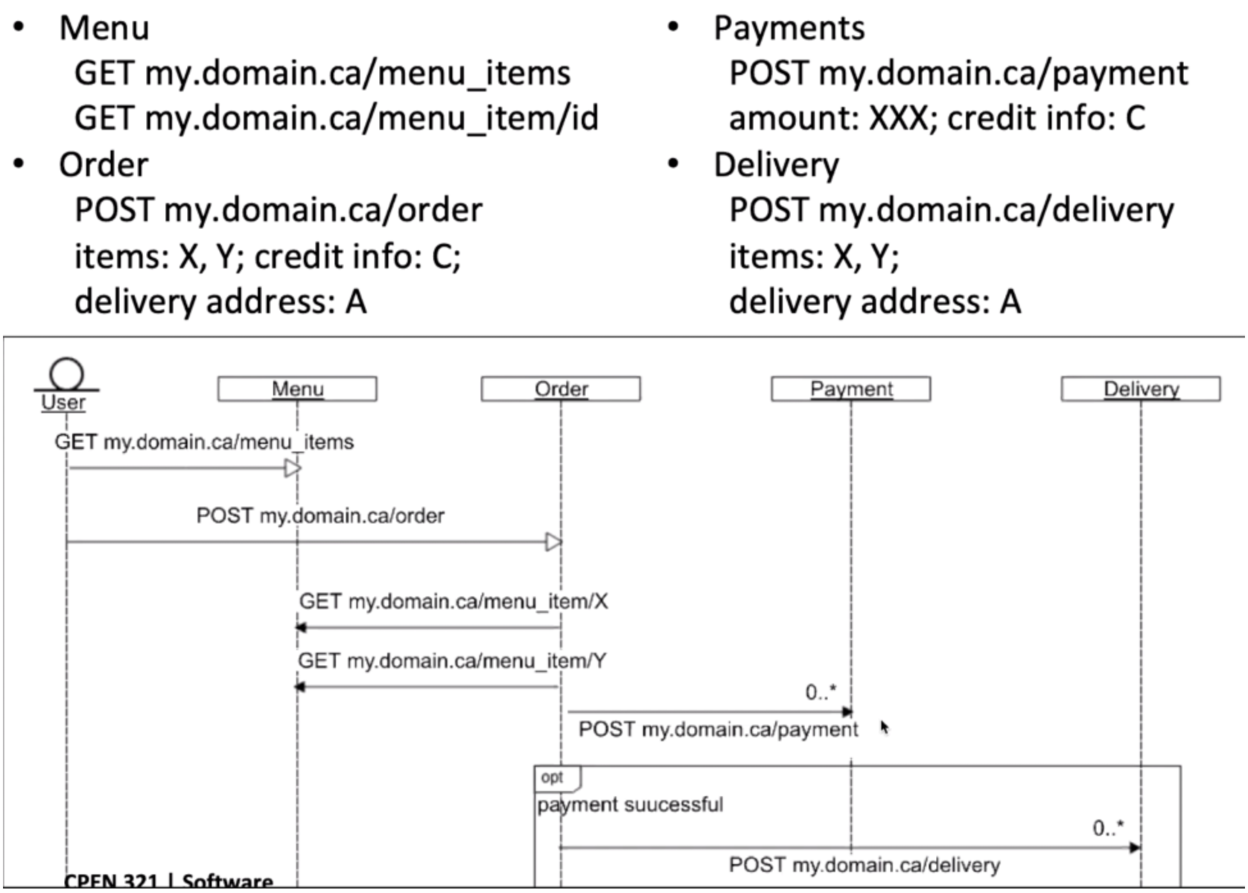


REST (Representational State Transfer)

- It is a design style (guideline) for communication in networked systems
    - Not a protocol or specification
- Main parts
    - Resource identification: URI
        - Most important
        - Every resource has a unique URI
        - Every URI refers to exactly one resource
    - Resource representation: any format, e.g. JSON, XML, web page
        - Can flow to and from the service
    - Unified interface to get, create, delete or update resources
        - REST uniform interface principle uses 4 main HTTP methods
            - GET: retrieve
            - POST: create
            - PUT: update
            - DELETE
        - Don't use GET to delete or post

Stateless server

- Server does not keep track of the client's state
- When a client makes a request, it includes all necessary information for the server to fulfill the request



- Menu
  GET my.domain.ca/menu_items
  GET my.domain.ca/menu_item/id
- Order
  POST my.domain.ca/order
  items: X, Y; credit info: C;
  delivery address: A
- Payments
  POST my.domain.ca/payment
  amount: XXX; credit info: C
- Delivery
  POST my.domain.ca/delivery
  items: X, Y;
  delivery address: A

User
Menu
Order
Payment
Delivery
GET my.domain.ca/menu_items
POST my.domain.ca/order
GET my.domain.ca/menu_item/X
GET my.domain.ca/menu_item/Y
0..*
POST my.domain.ca/payment
opt
payment suucessful
0..*
POST my.domain.ca/delivery
CPEN 321 | Software

Lecture 7

October 14, 2020 5:19 PM

What to track in version control

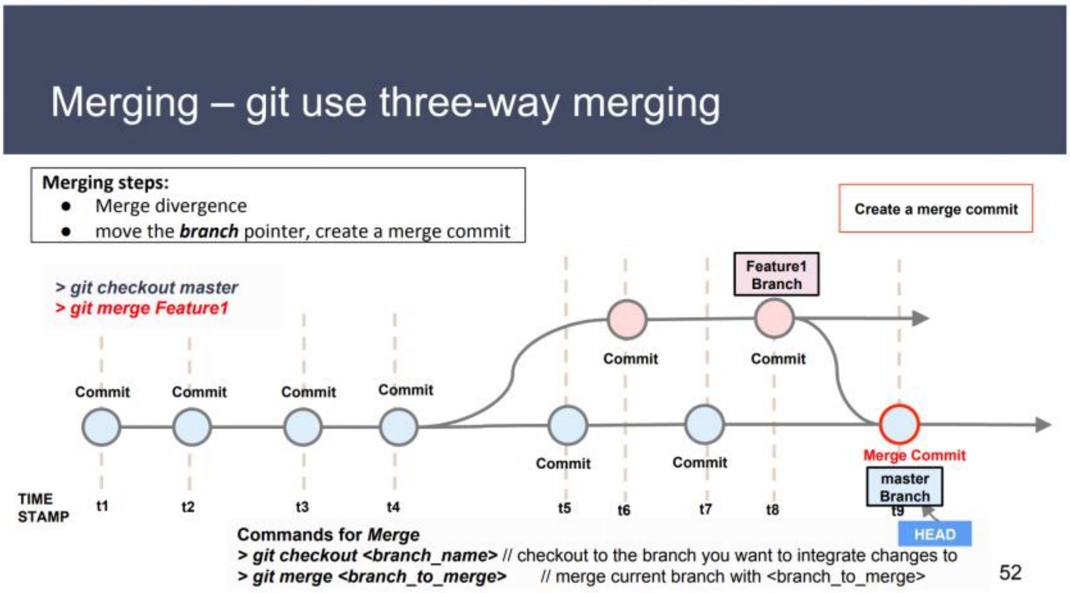
- Source code without generated files
- Tests
- Docs
- Configuration files

Types of version control system

- Centralized
    - A central repository contains all data and histories
    - All commits are made to the central repo
    - Each developer only has a snapshot of the repo
    - Pros:
        - Everyone knows what the others do
    - Cons:
        - If the main server goes down, single point of failure
        - Cannot keep track of their own change without sharing
- Distributed
    - Each copy is a full repo
        - Include data of current version and full history
    - Developers can commit locally to their own repo
        - Push to the remote, if they want their commits to be visible to others
    - No centralized repo, changes can go to any remote
    - Pros:
        - Do local commits, full history is always available
        - Don't need to access a remote server
        - Can commit changes continuously
    - Cons:
        - More complex synch mechanism
        - Require a large amount of space when working with binary files that cannot be compressed

Git

- Branching
    - Can write and test different solutions in parallel
    - Can develop two features at the same time
    - Achieves code isolation
    - Master branch: default branch when creating a repo
    - Head: a special pointer that simply points to the currently checked out branch or commit
        - Git checkout changes the head pointer
        - Git checkout HEAD~1: roll back to the parent of the HEAD
        - Git checkout HEAD~2: roll back 2 generations of HEAD
- Merging
    - Git uses 3-way merging
        - What is the original version
        - What you changed
        - What the other developer changed
    - 2-way merging
        - Cannot tell whether you/I/Both modified something
    - Steps

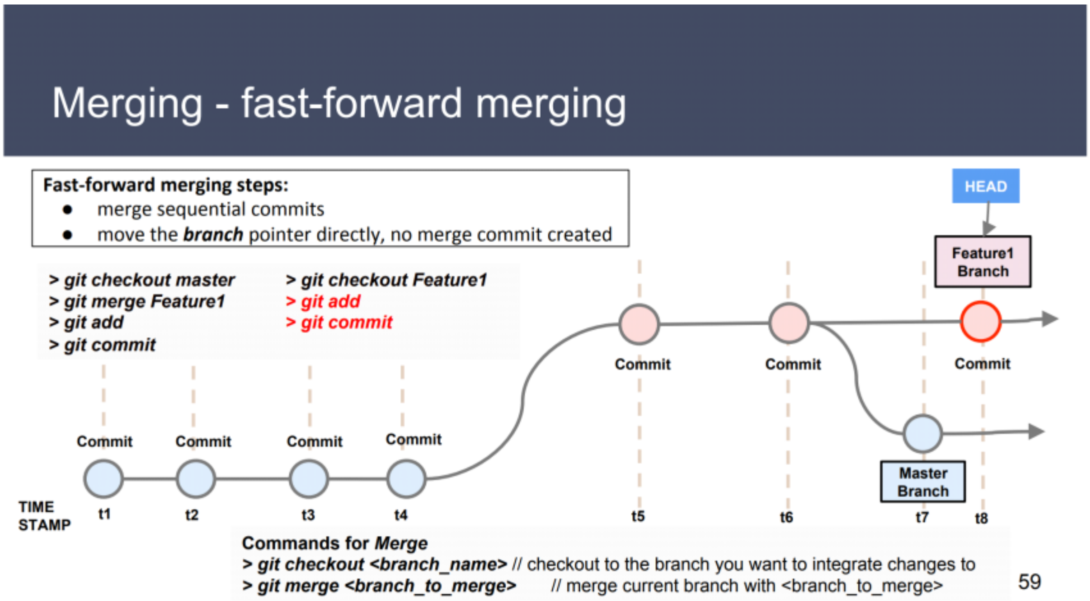
- 

Merging – git use three-way merging

Merging steps:
- Merge divergence
- move the ***branch*** pointer, create a merge commit

> git checkout master
> git merge Feature1

Commands for *Merge*
> git checkout <branch_name> // checkout to the branch you want to integrate changes to
> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

- 

Merging - fast-forward merging

Fast-forward merging steps:
- merge sequential commits
- move the ***branch*** pointer directly, no merge commit created

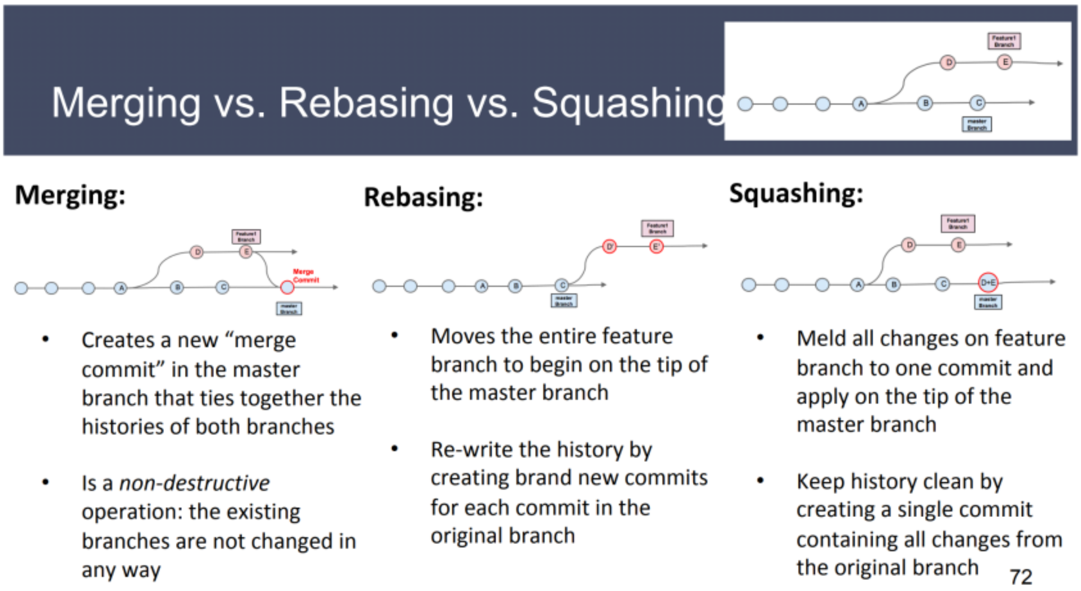
> git checkout master
> git merge Feature1
> git add
> git commit

> git checkout Feature1
> git add
> git commit

Commands for *Merge*
> git checkout <branch_name> // checkout to the branch you want to integrate changes to
> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

- If used properly
  - Non-destructive
  - Keeps info in merge commit
- If used improperly
  - Creates large amount of extraneous merge commits
  - Might cause the project histories to be messy and less readable

- Rebasing
  - To avoid messy history
  - Shift the branch from one base master branch timestamp to another
  - Pros
    - Keep a clean linear project history
    - No merge commits
  - Cons
    - Rewrite project histories
    - Lose information such as conflict resolutions
- Squashing
  - Meld a series of commits down into a single commit

- 

Merging vs. Rebasing vs. Squashing

Merging:

- Creates a new "merge commit" in the master branch that ties together the histories of both branches
- Is a *non-destructive* operation: the existing branches are not changed in any way

Rebasing:

- Moves the entire feature branch to begin on the tip of the master branch
- Re-write the history by creating brand new commits for each commit in the original branch

Squashing:

- Meld all changes on feature branch to one commit and apply on the tip of the master branch
- Keep history clean by creating a single commit containing all changes from the original branch

72

- Cherry-pick
  - Choose a commit from one branch and apply it to another by creating a new commit
  - Useful when developers need a specific commit applied to some branches, but not commits prior to this one
  - Creates a duplicate commit with the same changes and developers lose the ability to track the history of the original commit
- Conflicts in integration
  - Conflicts occur when
    - Two commits modified the same line in the same file
    - A file is deleted that another person is attempting to edit
  - Must resolve merge conflicts before merging
  - Integrate frequently to avoid merge conflicts

GitHub

- Git is the version control system, a tool to manage source code history
- GitHub is a hosting service for Git repos

Clone and fork

- Clone uses the same copy
- Fork makes a new copy of the repo
  - You will not affect the original copy when modifying the forked copy
  - Used to propose changes or use other people's repo as starting point

Pull request

- If have write access, can push directly
- Otherwise, need a pull request

Workflow

- Master only good for small simple projects (master is always deployable)
  - Everyone works on the master branch
  - Always pull before push
- Master/develop workflow (develop is center of development work)
  - Two branches: master and develop
  - Master HEAD always reflects a production-ready state
  - Develop HEAD always reflects a state with the latest delivered changes for next release
- Feature branch (used for individual features)
  - Exists when the feature is in development
  - Eventually merged back into develop or discarded
- Release branch (keep track of all releases)

- Create a branch for each upcoming release
- Enables concurrent release management, multiple and parallel releases

Lecture 8

October 19, 2020 3:02 PM

Push notifications
Three components

- Front-end client
- Back-end server
- Push notification server

Workflow:

- Front-end client creates a persistent connection with the push notification server and receives a token that reflects their connection
- The token is sent to the back-end
- Back-end, sends the message to the push notification server with the token
- Push notification server notifies the front-end through the persistent connection

Use Firebase cloud messaging for push notifications

Lecture 9

October 21, 2020 5:06 PM

Code review

- When
  - When developer's code is integrated with any of the main branches
- Who
  - Everyone.
- Types
  - Manual
    - Improve the code
      - Direct feedback leads to better algorithms, tests, design patterns
      - Prospect of someone reviewing your code raises the quality threshold
      - Forces code authors to articulate their decisions
      - Reduces redundancy
    - Improve the programmer
    - What to look for?
      - Bugs
      - Security vulnerabilities
      - Performance issues
      - Common code problems related to
        - Understandability, readability
          - Inconsistent names
          - Disagreement between code and specification
          - Not following style standards
        - Adherence to coding standards and best practices
        - Design and architecture
        - Documentation/comments
      - Magic numbers
      - Fail fast
      - Duplicated code
      - Long lines of code, methods, classes
      - Conditional complexity
  - Automated
    - Manual code review is expensive
    - Code can be analyzed statically and dynamically
    - If automated analysis fails, the code is rejected and developer needs to fix

Lecture 10

October 26, 2020 3:04 PM

Verification: does the implementation meet the spec
Validation: does it address the customer needs
Testing involves both verification and validation

Test plan: A document describing the scope, approach, resources, and schedule of intended test activities
Test case: a single unique unit of testing code
Test suit: collection of test cases
Test oracle: expected behavior
Test harness: collection of all the above

Process:

- Choose input data
- Define expected outcome
- Run on the input to get the actual outcome
- Compare the actual and expected outcomes

Software testing is a dynamic verification of the behavior of a program:

- On a finite set of test cases
- Suitably selected from the usually infinite executions domain
- Against the specified expected behavior (oracle)

White-box/Black-box testing

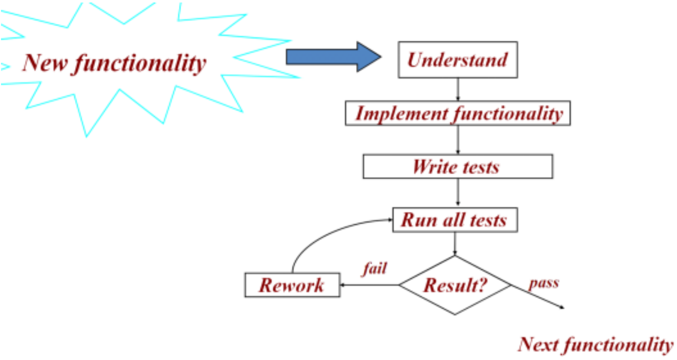
- White-box (code internal)
  - Unit testing
  - Component testing
  - Every line of the code is covered
    - Statements, branches, paths
  - Find bugs in the implementation that are not covered by the specification
  - Test may have same bugs as implementation
- Black-box (input-output)
  - Integration testing
  - User acceptance testing
  - Based on requirement or design specification of the software
  - Robust with respect to changes in the implementation
    - No need to change test when code changed
  - Allows for independent testers
  - Process is not influenced by component being tested

Level of automation:

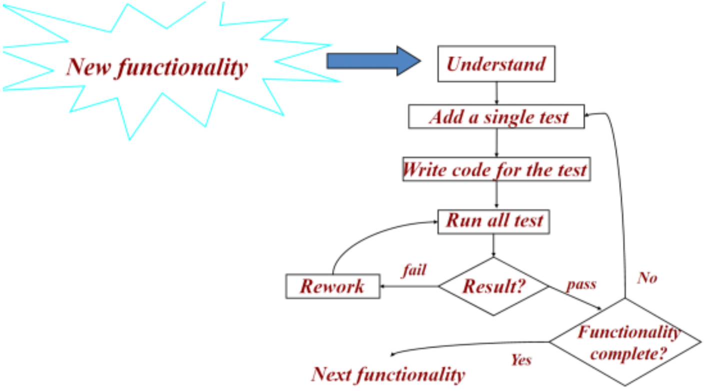
- Manual testing
  - Manually creating test cases
  - No automation
  - Pros:
    - Clever test case design
    - Interaction with system inspiration for new tests
    - Human oracle
  - Cons:
    - Single test case execution
    - Limited data
    - Might not be repeatable

- Test scripting
    - Manually creating test cases
    - Automated test execution
    - Repeatable
- Test generation
    - Automatically generate test cases
    - Based on some criteria (e.g. path coverage)
    - Oracle problem
    - Pros:
        - Clever test case design
        - Repeatable, facilitates continuous testing
        - More test cases and input data possible
        - Human oracle (documented)
    - Cons:
        - Cost of setting up test infrastructure
        - Maintenance cost of test suites

Test last: the conventional way. Testing follows the implementation



Test first: agile view in which testing is used as a development tool

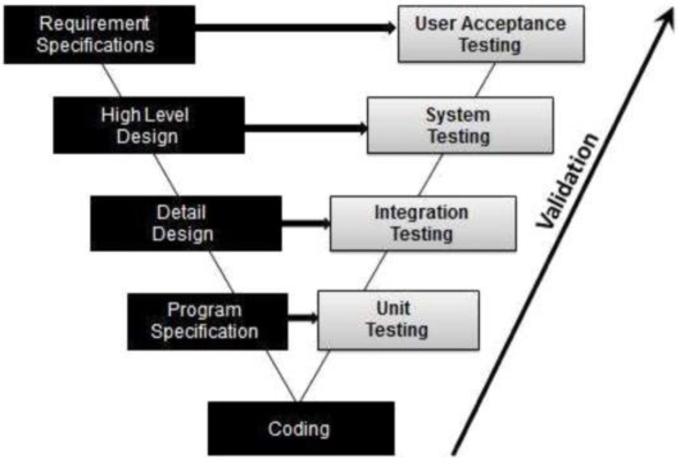


Regression testing

- Verifies that software which was previously developed and tested still performs the same way after it was changed or interfaced with other software.
- Process: when find a bug
    - Store the input that elicited that bug, plus the correct output
    - Add these to the test suite
    - Check that the test suite fails
    - Fix the bug and verify the fix
- Why
    - Ensures that the fix solves the problem
    - Helps to populate test suite with good tests
    - Protects against versions that reintroduce the bug
    - It happened at least once, and it might happen again

Summary:

- Write tests first, then implement
- Regression
- Automation
- Statement-level coverage



Unit tests

- Tests the behavior of an individual unit in isolation
- Typically written by developers
- Typically automated

Assertions:

- If the condition is true:
  - Execution continues normally
- If the condition is false:
  - Test fails
  - Execution skips the rest of the test method
  - Message is printed

Mocking:

- A controllable replacement for an existing software unit to which your code under test has a dependency
- A mock is a type of test double object
  - A test double object replaces a production object for testing purposes
    - To test partially implemented systems
    - To eliminate dependencies of your system so your tests are more focused on your functionality
    - To abstract away difficult-to-control elements
  - Other types of test double object
    - Dummy: passed around but never used. (to fill parameter list)
    - Fake: take shortcuts which makes them not suitable for production
    - Stubs: canned answers to calls made during the test
    - Spies: stubs that also record information based on how they were called
- Core idea:
  - Identify the external dependency
    - Suppose A depends on B
  - Extract the core functionality of the object into an interface
    - Create an interface B based on B
    - Change all of A's code to work with interface B
  - Write a tub class that also implements the interface, but returns predetermined fake data
- Mocking with Jest
  - Reassign a function to the mock function (jest.fn())

Lecture 11

October 28, 2020 5:08 PM

Java script promise

- All async functions return a Promise object
- Represents the eventual completion or failure of an asynchronous operation and its resulting value
- Can be resolved or pending

Integration testing

- Individual software modules are combined and tested as a group
- Approaches
    - Big-bang
        - Most of the developed modules are coupled together to form a complete software system
        - Effective for saving time in the integration testing process
        - Failures are hard to pinpoint
    - Bottom-up
        - Lowest level components are tested first
        - Repeat until the component at the top of the hierarchy is tested
        - Helpful only when all or most of the modules of the same development level are ready
    - Top-down
        - Reverse of bottom-up
        - Simulate the behavior of the lower-level modules that are not yet integrated
    - Mixed (sandwich)
        - Combines top-down with bottom-up
    - Risky-hardest
        - Starting with the risky and hardest software module first

System testing

- Test the behavior of the system as a whole
    - Functional testing (all requirements are met)
        - From the backend and front-end side
    - Installation
    - Performance, load, stress testing
        - Performance is a major aspect of program acceptance by users
        - Measure before optimizing
            - Runtime CPU/memory usage
            - Web page load times, requests/minute, latency
        - Focus on high-level optimizations
        - Lazy evaluation, caching, combining queries saves time
    - Usability
    - Graphical user interface testing
    - Other non-functional requirements

Profiling:

- Log and monitor
    - Especially for cloud-based systems
- Profiling is expensive and slows down the code
    - Make sure it is short
- If the app meet's the project's stated performance requirements, don't optimize it

User acceptance testing

- System is shown to the user/client/customer to make sure that it meets their needs
    - A form of black-box system testing
- Beta testing
    - Advantages
        - Customers test for free
        - Gives test cases representative of customer use
        - Helps to determine what is most important to the customers
        - Test in real settings other than in lab
    - Disadvantages
        - Do not exhaust your beta-testers
        - Beta testers may have a particular perspective to the system, may not able to catch system bugs
- GUI testing
    - GUI responds to user events (clicks)
        - Event-driven systems
    - GUI interacts with the underlying code by method calls or messages
    - Testing GUI correctness is critical for system usability, robustness and safety
    - Difference between GUI and non-GUI
        - Non-GUI: test cases invoke methods of the system and catch the return values
        - GUI:
            - Identify the components of a GUI
            - Exercise GUI events
            - Provide inputs to the GUI components
            - Test the functionality underlying a GUI set of components
            - Assert the GUI properties to see if they are consistent with the expectations
    - Types:
        - During acceptance testing: accept the system
        - Regression testing test the system with respect to changes
    - Challenges
        - Maintenance is hard and costly
            - Non-deterministic behavior
            - GUIs are dynamic and change
            - Small structural changes can break the test cases
        - Adequacy hard to measure
        - Technology-dependent
    - Approaches
        - Manual
            - Based on the domain and application knowledge of the tester
        - Capture and replay
            - Based on capture and replay of user sessions
            - Difficult to detect faults looking at the GUI
            - Indeterministic state transitions
            - Relies on screen diffing
            - Some tools produce scripts that can be updated by the tester to include conditions and acceptance criteria
        - Manual test generation
            - E.g. Espresso for Android
                - Instrumentation-based framework
                - Use Android Instrumentation to inspect and interact with Activities under test
        - Automated test generation
            - Random event generator
                - E.g. Monkey tester
                    - Fires random events
                    - Report crashes or errors
                    - Struggles to provide text inputs

        - ◇ Low code coverage
        - ◇ No test oracle
    - □ Model-based
    - □ Search-based

Testing is one of the most important SE activities
Be systematic

Lecture 12

November 2, 2020 3:00 PM

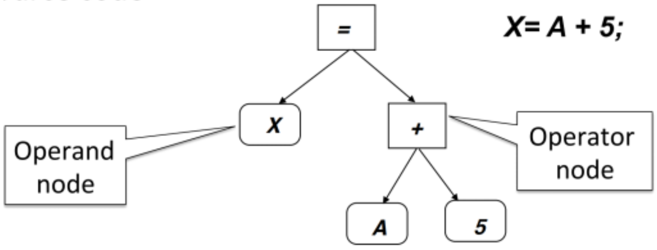
Static program analysis: reasoning about code

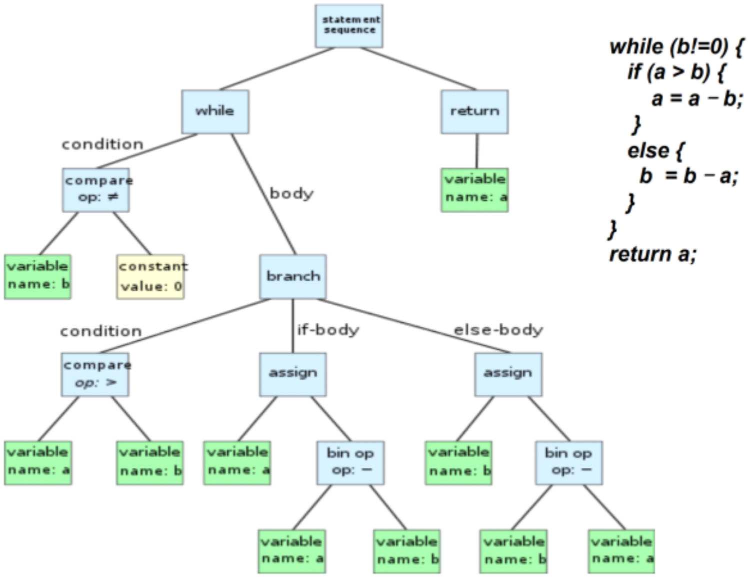
- Process of automatically analyzing the behavior of programs
    - Input: the code of the program
    - Output: code or interesting facts about the code
- E.g. compilers, intellisense
- Major application
    - Program correctness
    - Program optimization
    - Program understanding, validation, and repair

Why program analysis:

- Reduce development costs
    - Validation and verification is usually 50%
- Maintenance costs
    - 2-3 times as much as development costs

Models: abstract syntax tree (AST)

- Common form of representing expressions and program statements
- Two kinds of nodes: operator and operands
    - Operator applied to N operands
- Each node denotes a construct occurring in the source code
- 

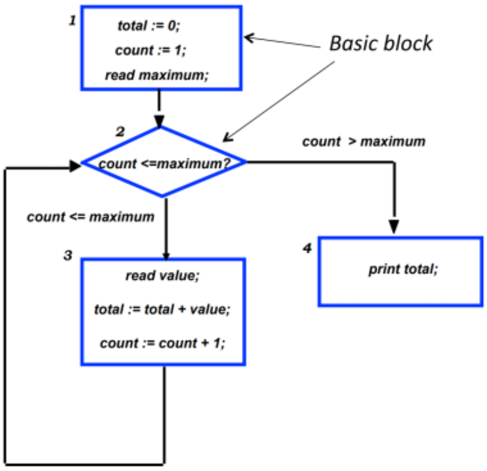
- 

```
while (b!=0) {
  if (a > b) {
     a = a − b;
  }
  else {
    b  = b − a;
  }
}
return a;
```

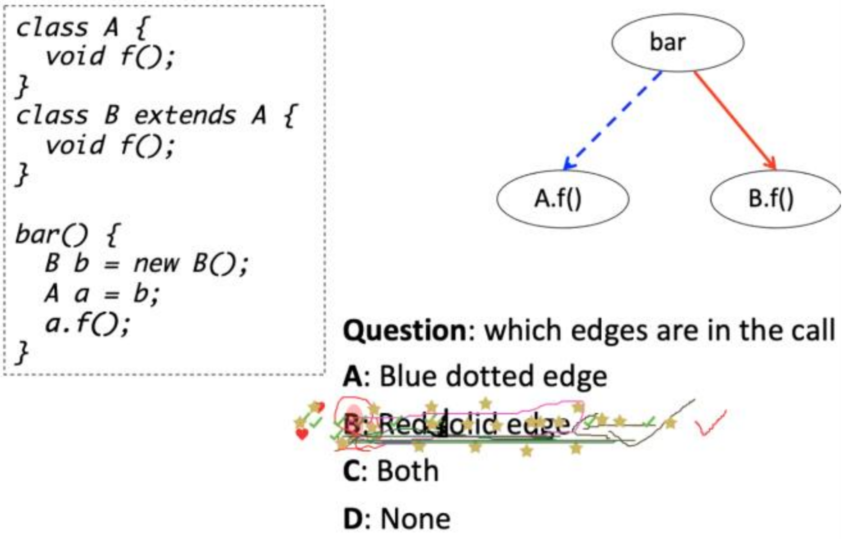
Control flow graph (CFG):

- Basic block: maximal program region with a single entry and single exit point
- Nodes N: statements or basic blocks
- Directed edges E: potential transfer of control from the end of one region directly to the

beginning of another

- Intra-procedural (within a method)
- A sub path through a control flow graph:
  - A sequence of nodes such that for each n_i, (n_i, n_{i+1}) is an edge in the graph
- A complete path starts at the start node and ends at the final node
- Infeasible path: path that will never been reached
  - CFG overestimates the executable behavior
- Benefits
  - The most commonly used representation
  - Basis for many types of automated analysis
    - Graphical representations of interesting programs are too complex for direct human understanding
  - Basis for various transformations
    - Compiler optimizations
    - Software analysis
- 

Call graphs (Inter-procedural CFG)

- Between functions
- Node represent procedures
- Edges represent potential calls relation
- 

```
class A {
   void f();
}
class B extends A {
   void f();
}

bar() {
   B b = new B();
   A a = b;
   a.f();
}
```

Question: which edges are in the call

A: Blue dotted edge

[✓ **B**: Red solid edge]

C: Both

D: None

  - F is overridden in B
- Creating the exact (static) call graph is an undecidable problem
  - All non-trivial semantic properties of programs are undecidable
    - A semantic property is about the program's behavior (i.e. does the program terminate for all inputs)
    - A property is non-trivial if it is neither true nor false for every computable function

- Computing call graphs requires
  - Point-to analysis
  - Exceptions
- Multiple existing heuristic algorithms
  - Various degree of precision/scalability

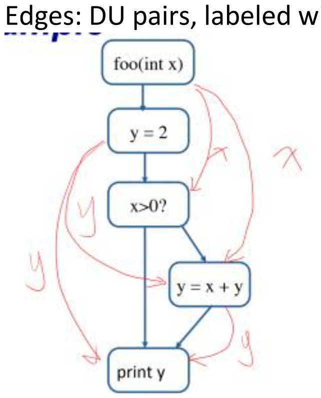
Data flow analysis

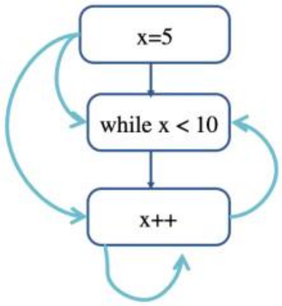
- A technique for gathering information about the propagation of data values in the program

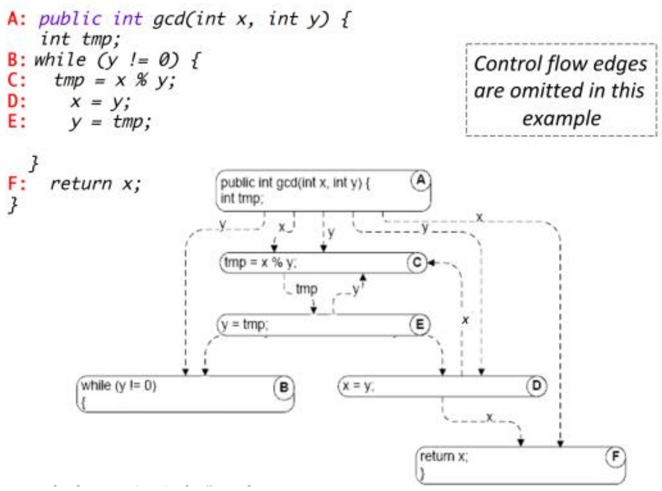
Variable Definition and uses(DU)

- Variable definition: the variable is assigned a value
  - Variable declaration (often the special value uninitialized)
  - Variable initialization
  - Assignment
  - Values received by a parameter
  - Value increments
- Variable use: the variable's value is actually used
  - Expressions
  - Conditional statements
  - Parameter passing
  - returns

Data dependence graph:

- Nodes: program statements
- Edges: DU pairs, labeled with the variable name
- 

  - Keep all the arrows
- 

- 

```
A: public int gcd(int x, int y) {
     int tmp;
B: while (y != 0) {
C:   tmp = x % y;
D:     x = y;
E:     y = tmp;

   }
F:   return x;
}
```

Control flow edges are omitted in this example

- Used in
  - Compilers and optimization
  - Security analysis

Lecture 13

November 4, 2020 5:19 PM

Testing is a dynamic verification of the behavior of a program

- On a finite set of test cases
- Suitably selected from the usually infinite executions domain
- Against the specified expected behavior

Systematic testing:

- Black-box: test cases come from requirements/user stories
- White-box: inspect the code/coverage criteria to see if you missed cases

Measuring test suite quality with coverage

- Various kinds of coverage
  - Statement: is every statement run by some test case?
    - Each statement (or node in the CFG) must be executed at least once
    - Coverage = $\frac{\#\ executed\ statements}{\#\ statements}$
  - Branch: is every direction of an if or while statement taken by some test case
    - Every path going out of a node executed at least once
    - Coverage: percentage of edges hit
    - Each predicate must be both true and false to achieve 100%
  - Path: is every path through the program taken by some test case
    - Coverage: $\frac{\#\ executed\ paths}{\#\ paths}$
    - Each CFG path must be executed at least once

Limitations of Symbolic execution

- Expensive
  - Executing all feasible program paths is exponential in the number of branches
  - Does not scale to large programs
- Problems with function calls
- Problems with handling loops
  - Often unroll them up to a certain depth rather than dealing with termination or loop invariants

To write a test

- Identify the fault
- Write a test case that does not execute statements related to the fault
- Write a test case that executed the statements related to the fault, but does not result in a detectable error state
- Write a test case that detects the fault

Limitations of coverage

- Coverage is just a heuristic
- 100% coverage may not be achievable
- 100% is not sufficient
- Common practice: statement-level coverage + clever test selection + test case for all found bugs + regression
- More advanced techniques: input space partitioning, combinational testing

Lecture 14

November 9, 2020 3:36 PM

DevOps:

- A software engineering practice that aims at unifying software development (Dev) and software operations (Ops)
- Why
  - Limited capacity of operations staff
  - Limited dev insights into operations
  - Developers and operators don't always pursue the same goals
    - Developers want to push new features
    - Operators want to keep the system available
  - Poor communication between developers and operators
- Encourages communication and collaboration between development and operations staff, get them talking
- Tool Chain
  - Plan: requirements, architecture, design
  - Create: code development and review, source code management tools
  - Code merging
  - Build: continuous integration tools, build status
  - Test: continuous testing tools that provide feedback on business risks
  - Package: artifact repository, application pre-deployment staging
  - Release: change management, release approvals, release automation
  - Configure: infrastructure configuration and management, infrastructure as code tools
  - Monitor: applications performance monitoring, end-user experience

Continuous integration:

- The practice of routinely integrating code changes into a main branch of repository, and testing the changes, as early and often as possible
- Developers work on a feature branch
- At regular intervals they submit pull requests
- Branch tested and integrated with development branch
- Tools:
  - Travis
  - Jenkins
  - Pipelines
  - Integrate with Git-based version control system

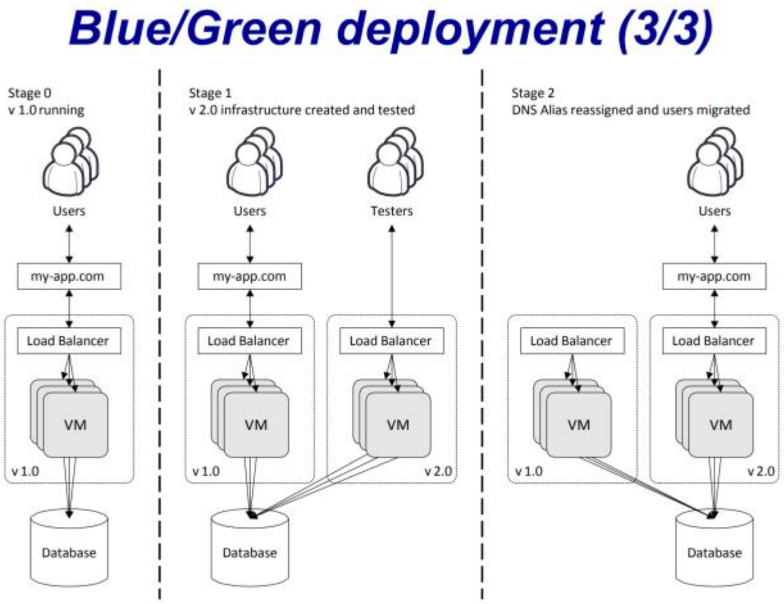
Deployment is not trivial:

- Challenges:
  - Any development team can deploy their code at any time - no synchronization among development teams
  - It takes time to replace one instance of version A with an instance of version B
  - Needs to be always available to customers
- Solution: API Gateway/Proxy
  - Single entry point for all clients for a number of different underlying APIs
  - Limit clients' visibility of your internal structure
  - Performs authentication/authorization/logging
  - Can be configured to route the request to the appropriate version/service

Load balancer

- Facilitates load distribution
- Directs traffic efficiently to all the servers present in the application configuration

Usage and tools

- Multiple concepts can be implemented in one tool
- Support continuous integration, blue-green deployments, API management
- 

  Blue/Green deployment (3/3)

  - Only one version is available at any time
  - Requires 2N VMs
    - Additional cost
  - Rollback is easy
- Rolling upgrade: upgrade VM, APIs one by one
  - Multiple versions are available at the same time
  - Requires N+1 VMs
    - Can be done at nearly no extra cost

Canary testing

- Canaries are small number of instances of a new version placed in production in order to perform live testing in a production environment
- Canaries are observed closely to determine whether the new version introduces any logical or performance problems. If not, roll out new version globally. If so, roll back canaries
- Implementation
  - Create set of new VMs as canaries
  - Designate a collection of customers as testing the canaries.
    - Organization-based
    - Geographically based
    - At random
  - Then
    - Route messages from canary customers to canaries
      - Can be done through making registry/load balancer canary aware
    - Observe the canaries closely
    - Decide on rolling out/back

Dev and Ops are related activities

- Developers' responsibility: unlikely to be able to "throw your final version over the fence" and let operations worry about running it!
- Result: Shorter development cycles, increased deployment frequency, closer alignment with business objectives

Automation is important

- Makes the processes faster, more manageable, more repeatable

Tools can help but cannot replace good practices and processes