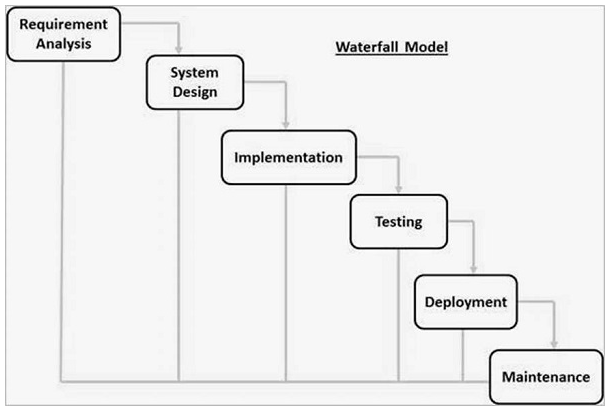
Definition of Software Engineering:

* Software engineering is the systematic application of engineering approaches to the development of software
* "the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software"
* "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of [software](https://en.wikipedia.org/wiki/Software)"

Software Development Lifecycles:

* Waterfall
  + 
  + In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.
  + Sequential phases:
    - Requirement Gathering and analysis − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
    - System Design − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
    - Implementation − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
    - Integration and Testing − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
    - Deployment of system − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
    - Maintenance − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.
  + Advantages:

The advantages of waterfall development are that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.

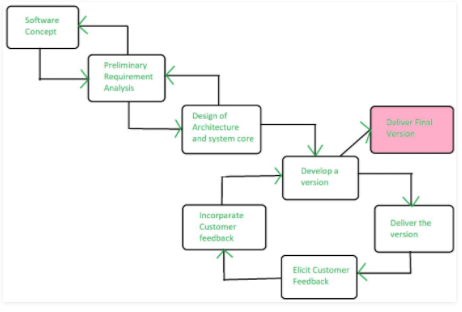
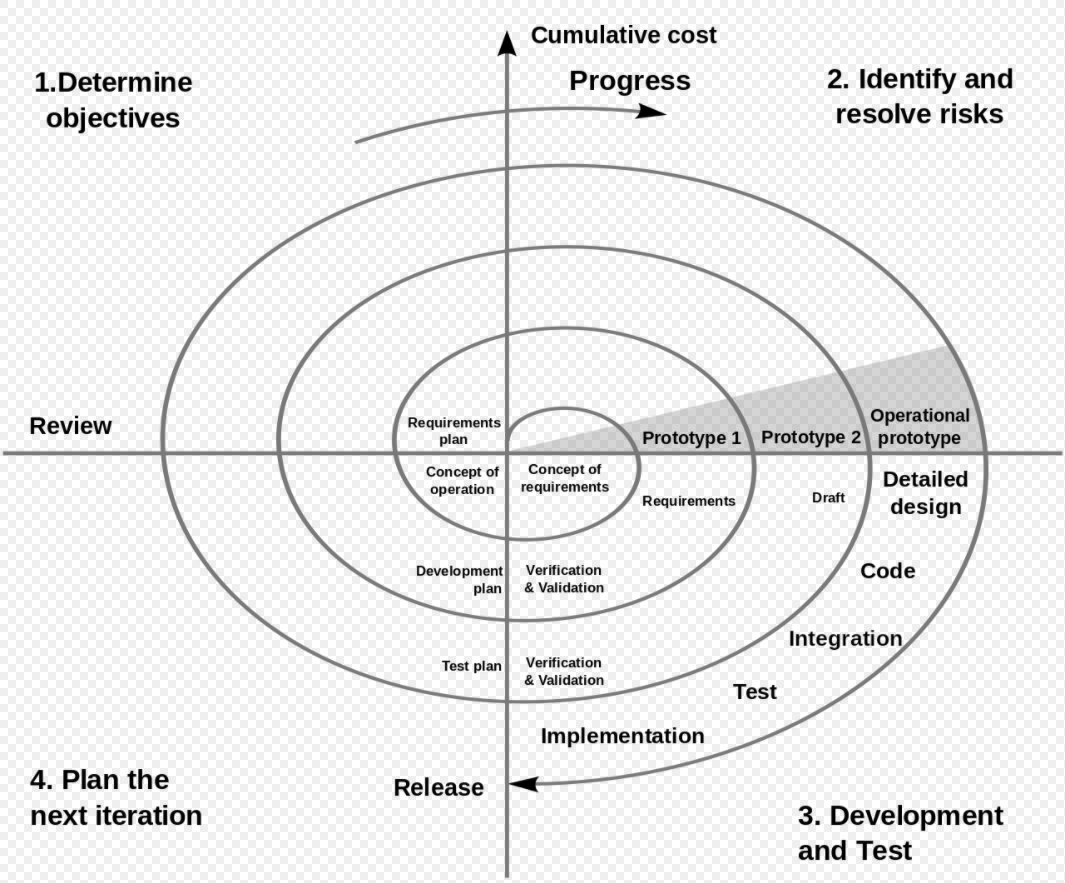
Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.

Some of the major advantages of the Waterfall Model are as follows −

* Simple and easy to understand and use
* Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
* Phases are processed and completed one at a time.
* Works well for smaller projects where requirements are very well understood.
* Clearly defined stages.
* Well understood milestones.
* Easy to arrange tasks.
* Process and results are well documented.
  + Disadvantages:

The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

The major disadvantages of the Waterfall Model are as follows −

* No working software is produced until late during the life cycle.
* High amounts of risk and uncertainty.
* Not a good model for complex and object-oriented projects.
* Poor model for long and ongoing projects.
* Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
* It is difficult to measure progress within stages.
* Cannot accommodate changing requirements.
* Adjusting scope during the life cycle can end a project.
* Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.
* Evolutionary
  + 
  + Evolutionary model is a combination of [Iterative](https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/) and [Incremental model](https://www.geeksforgeeks.org/software-engineering-incremental-process-model/) of software development life cycle. Delivering your system in a big bang release, delivering it in incremental process over time is the action done in this model. Some initial requirements and architecture envisioning need to be done.
  + It is better for software products that have their feature sets redefined during development because of user feedback and other factors. The Evolutionary development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle.
  + Applications:
    - It is used in large projects where you can easily find modules for incremental implementation. Evolutionary model is commonly used when the customer wants to start using the core features instead of waiting for the full software.
    - Evolutionary model is also used in object oriented software development because the system can be easily portioned into units in terms of objects.
  + Advantages:
    - In evolutionary model, a user gets a chance to experiment partially developed system.
    - It reduces the error because the core modules get tested thoroughly.
  + Disadvantages:
    - Sometimes it is hard to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented and delivered.
* Spiral
  + 
  + The spiral model is a risk-driven [software development process](https://en.wikipedia.org/wiki/Software_development_process) model. Based on the unique risk patterns of a given project, the spiral model guides a team to adopt elements of one or more process models, such as [incremental](https://en.wikipedia.org/wiki/Iterative_and_incremental_development), [waterfall](https://en.wikipedia.org/wiki/Waterfall_model), or [evolutionary prototyping](https://en.wikipedia.org/wiki/Software_prototyping#Evolutionary_prototyping).
  + Advantages:
    - Changing requirements can be accommodated.
    - Allows extensive use of prototypes.
    - Requirements can be captured more accurately.
    - Users see the system early.
    - Development can be divided into smaller parts and the risky parts can be developed earlier which helps in better risk management.
  + Disadvantages:
    - Management is more complex.
    - End of the project may not be known early.
    - Not suitable for small or low risk projects and could be expensive for small projects.
    - Process is complex
    - Spiral may go on indefinitely.
    - Large number of intermediate stages requires excessive documentation
* Difference between Agile and XP
  + Extreme programming(XP) is a type of Agile programming.
  + Principles of Agile:
    - Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
    - Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
    - Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
    - Business people and developers must work together daily throughout the project.
    - Build projects around motivated individuals.
    - Give them the environment and support they need, and trust them to get the job done.
    - The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
    - Working software is the primary measure of progress.
    - Agile processes promote sustainable development.
    - The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
    - Continuous attention to technical excellence and good design enhances agility.
    - Simplicity--the art of maximizing the amount of work not done--is essential.
    - The best architectures, requirements, and designs emerge from self-organizing teams.
    - At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
  + Extreme programming:
    - Extreme programming (XP) is a [software development methodology](https://en.wikipedia.org/wiki/Software_development_methodology) which is intended to improve [software quality](https://en.wikipedia.org/wiki/Software_quality) and responsiveness to changing customer requirements. As a type of [agile software development](https://en.wikipedia.org/wiki/Agile_software_development),[[1]](https://en.wikipedia.org/wiki/Extreme_programming#cite_note-Informatics85-1)[[2]](https://en.wikipedia.org/wiki/Extreme_programming#cite_note-UPenn49-2)[[3]](https://en.wikipedia.org/wiki/Extreme_programming#cite_note-USFCA601-3) it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.
    - Other elements of extreme programming include: programming [in pairs](https://en.wikipedia.org/wiki/Pair_programming) or doing extensive [code review](https://en.wikipedia.org/wiki/Code_review), [unit testing](https://en.wikipedia.org/wiki/Unit_testing) of all code, [not programming features until they are actually needed](https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it), a flat management structure, code simplicity and clarity, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers.
* When/if one process is “better” to use
  + TODO:

Requirements and Specifications:

* Requirements vs Specifications
  + A specification is a document that specifies a system or product, e.g. a prime-item development specification for an F-14. There are lots of sections/content in a spec: requirements, definitions, reference documents, glossary, verification information, etc.
  + A requirement is a single statement of something the product or system must do. A spec may have hundreds of requirements in it. Old school methodology says the requirement statement must use the word "shall", to separate requirements from statements of facts, or definitions. (Not sure if the new-fangled agile kids keep to all this or not; the fastidiousness has it's use but is a little fussy at times.)
  + So a spec is a document full of requirements, plus some other supporting and ancillary information.
* Use cases

In [software](https://en.wikipedia.org/wiki/Software_engineering) and [systems engineering](https://en.wikipedia.org/wiki/Systems_engineering), the phrase use case is a [polyseme](https://en.wikipedia.org/wiki/Polysemy) with two [senses](https://en.wikipedia.org/wiki/Word_sense):

* + A usage scenario for a piece of software; often used in the plural to suggest situations where a piece of software may be useful.
  + A potential scenario in which a system receives an external request (such as user input) and responds to it.
  + A use case is a list of actions or event steps typically defining the interactions between a role (known in the [Unified Modeling Language](https://en.wikipedia.org/wiki/Unified_Modeling_Language) (UML) as an [actor](https://en.wikipedia.org/wiki/Actor_(UML))) and a system to achieve a goal. The actor can be a human or other external system. In systems engineering, use cases are used at a higher level than within [software engineering](https://en.wikipedia.org/wiki/Software_engineering), often representing missions or [stakeholder](https://en.wikipedia.org/wiki/Project_stakeholder) goals. The detailed requirements may then be captured in the [Systems Modeling Language](https://en.wikipedia.org/wiki/Systems_Modeling_Language) (SysML) or as contractual statements.
* Data flow diagrams
  + TODO:
* Class diagrams
  + TODO:

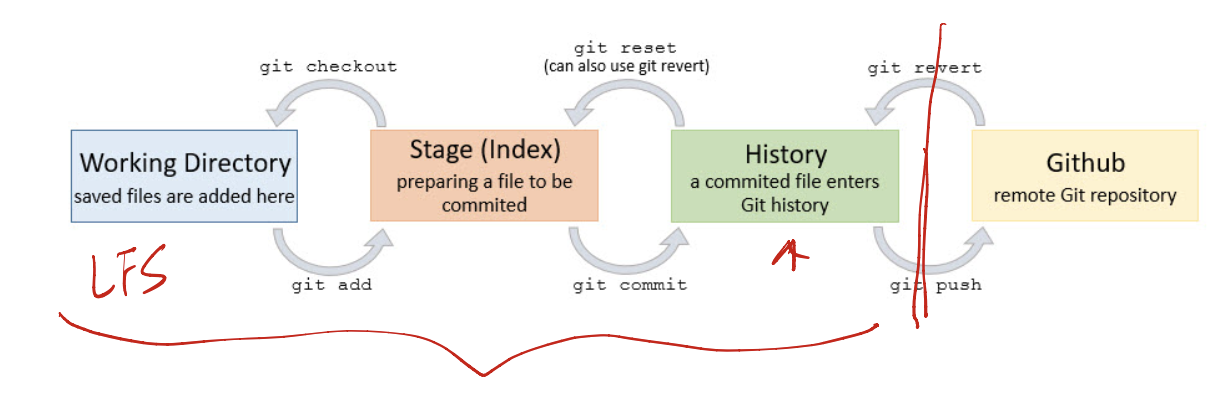
Software Architecture:

* Compare and contrast Model View Controller, Event-based, and Pipe & Filter
  + Model-View-Controller
    - Model
      * The main goal of MCV is to separate the model from the user interface (view + controller)
      * Model is the functional core of the application
      * Encapsulates the state of the application
        + Databases
        + Invariants
        + If model is correct, the data will continue to make sense
      * Independent of the GUI
        + Cannot hold direct references to the view or controller
      * This independence gives the model its flexibility, testability, and robustness
        + Can reuse the model in other MVC tirads
        + Can test the model using an automated controller and machine-checkable views
      * Active Model
        + Model defines a change notification mechanism that view and controller can subscribe to
        + Common in desktop GUI apps
      * Passive Model
        + Model is called by view and controller to get the data
        + Common in web UI
        + This puts somewhat more burden on controller
    - View
      * What data to show and how to present it
      * View informs the controller of the user actions
      * View has access to the model, but should not change the model
      * For web apps, view contains HTML + JavaScript
    - The Controller
      * View deals with “output”
      * Controller deals with “input”
        + Controller decides what model actions are needed
        + Controller also selects the next view
        + Controller allows the user to drive the application
      * In web app, controller will be handler for the incoming HTTP requests
      * In some apps, controller is minimal or generic
    - Advantages
      * Allows to separate and modularize concerns
      * Separation between the data layer and the interface is the key:
        + View is easily replaced or expanded
        + Model data changes are reflected in all views
        + Vetter scalability since UI and application logic are separated
        + Distribution over a network is greatly simplified
    - Disadvantages
      * View and controller depend heavily on model API
      * Close coupling between view and controller
* Terminology
  + Architectural components:
    - Computational elements
    - Where the “work” happens
    - May be course-grained: web server
    - Or fine-grained: a module
  + Architectural connectors:
    - Communica elements
    - Enable communication between components
    - May be explicit: procedure call
    - May be implicit: event
  + Architectural configurations
    - Are arrangements of components and connectors to form an architecture
* Pipe and filter architecture
  + Components: filter transform input into output
  + Connectors: data streams
  + Pipe and filter example
    - From source text to list of identifiers and keywords
    - From list of tokens to parse trees
    - From parse trees to intermediate language
    - From intermediate language to faster intermediate language
    - From intermediate language to machine code
  + Advantages
    - Overall behavior can be understood as a simplie composition of behaviors of individual filters
    - Support reuse
    - Modular testing
    - Easy maintenance and enhancement
      * New filters can replace old ones
    - Support specialized analysis, such as throughput and deadlock analysis
    - Support parallel or distributed execution
  + Disadvantages
    - Not good for handling interactive applications
    - Sometimes for performance we need tighter coupling filters
    - Extra overhead to parse and unparse data to an external format
  + Map-Reduce (Special case of pipe-filter)
    - Used in distributed data processing
    - Can parallelize
* Layered Architectures
  + Example: Operating Systems
    - Each later provides a higher level of abstraction to the layer above
* Layered Systems
  + Interesting properties
    - Support increasing levels of abstraction during design
    - Support enhancement
      * Change in one later affects at most two layers
    - Reuse
      * Can define standard layer interfaces
      * Interchange implementations of same interface
  + Disadvantages
    - May not be able to identify layers
    - For performance reasons one layer may want to communicate with a non-adjacent layer
* Event Based
  + Independent reactive objects
    - Components: objects that register interest in “events” and objects that “signal events”
    - Connectors:
      * Explicit method invocation to register event listeners
      * Implicit listener invocation when event fires
    - Control: decentralized, de-coupling of sender and receiver
    - Examples
      * Database management systems
      * GUI
    - Interesting properties
      * Announcers of events don’t need to know who will handle the event
      * Supports re-use
      * Supports evolution of system
    - Disadvantages
      * Components have no control over ordering of computations
      * Nor do they know when they are finished
      * Hard to understand the control flow
* Apply architectural styles and when to apply design patterns to developing them
  + TODO:

Design Patterns:

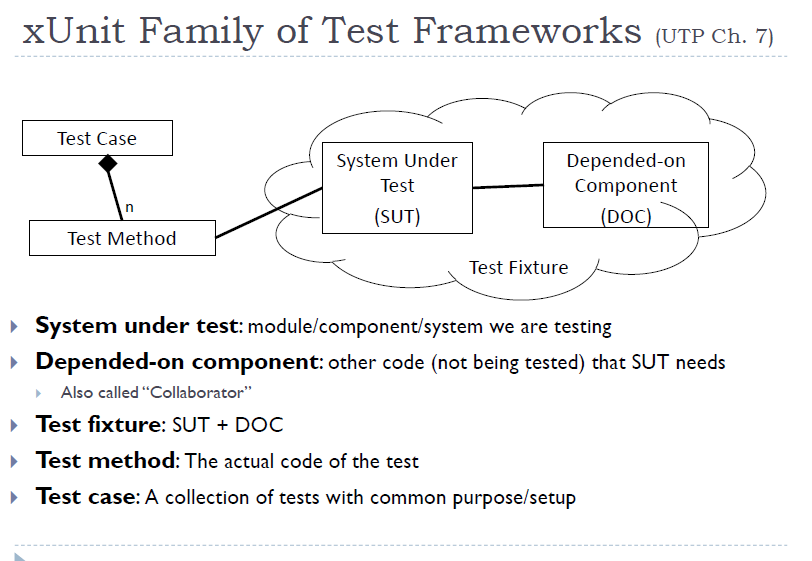
* Three general types of patterns (Only review the patterns discussed in class)
  + Alexander’s Pattern Language
    - A pattern language is an organized way of tackling an architectural problem using patterns
    - Patterns are Designed to Avoid Redesign (caused by)
      * Creating an object by specifying a class explicitly
      * Dependence on specific operations
      * Dependence on hardware and software platform
      * Dependence on object representations or implementations
      * Algorithmic dependencies
      * Tight coupling
      * Extending functionality by subclassing
      * Inability to alter classes conveniently
    - Patterns Apply Three Design Principles
      * Program to an interface, not an implementation
        + Interface should be separately defined, using the constructs in programming language
      * Favor object composition/delegation over inheritance
      * Find what varies and encapsulate it
    - Software Design Patterns
      * Description of Communicating objects and classes that are customized to solve a general design problem in a particular context
      * Not individual classes or libraries
      * Not full designs
      * Patterns let us:
        + Reuse solutions that have worked in the past
        + Have a shared vocab around software design
        + Design patterns provide you not with code reuse but with experience reuse
        + Design patterns can show you how to apply those concepts to achieve those goals
  + Object-Oriented Design Patterns
    - Creational - Patterns deal with the process of object creation
    - Structural - Patterns, deal primarily with the static composition and structure of class and objects
    - Behavioral - Patterns, which deal primarily with dynamic interaction among classes and objects
  + Elements of a design pattern
    - Pattern name
      * Useful part of design vocabulary
    - Problem solved and applicability
      * When to apply the pattern
    - Solution
      * Participants and their relationships
    - Consequences
      * Cost of applying the pattern, space and trade-offs
  + Design Trade-offs
    - With inheritance, we get
      * Code reuse
      * Common behavior in root class
    - With interfaces, we get
      * Specificity
      * No code reuse
    - Encapsulate What Varies
      * We need to pull out behaviors that vary across the subclasses and put them into their own class
      * The result: fewer unintended consequences from code changes
  + Composites
    - Recursive composition, structural induction, tree walk,...
    - Predates design patterns
    - Applies to any hierarchical structure
      * Leaves and internal nodes have same functionality
      * Composite implements the same interface as the contained elements
    - Pros
      * Easy to add, remove new objects
      * Clean interface, elegant code
    - Cons
      * Have to fight to add corner cases
      * Hard to enforce that a composite only has certain elements
  + Visitors
    - The visitor pattern is more general
      * Iterator provide traversal of containers
      * Visitors allow
        + Traversal
        + And type-specific actions on container elements
    - The big idea
      * Separate traversal from the action
    - The visitor patter is extremely powerful
    - Whenever you have a hierarchical
  + Decorators
    - A way of adding responsibilities to an object
      * Decorations can even be choses dynamically
      * Transparent enclosure
    - Commonly extends a composite
  + Factory
    - A class which
      * Abstracts the creation of a family of objects
      * Different instances provide alternative implementations of that family
  + Bridge
    - Logical
      * The view of our application, tuned to our needs
    - Implementation
      * The interface to the outside world
      * Abstract base class with multiple implementations
    - Logical implementations views can evolve independently
      * So long as the “bridge” is maintained
  + Software Adapter
    - Pre-condition: you are maintaining an existing system that makes use of a third-party class library from vendor A
    - Stimulus: Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library
    - Response: Vendor B provides similar class library, but its interface is completely different from the interface provided by vendor A
    - Assumptions: You don’t want to change your code, and you cannot change vendor B’s code
    - Solution: Write new code that adapts vendor B’s interface to the interface expected by your original code
  + Strategy
    - Isolates variation in algorithms we might use
    - Formatter is the strategy, Compositor is next
    - General principle: encapsulate variation
    - In OO languages, this means defining abstract classes for things are likely to change
  + Command
    - User essentially has a small “programming language”
      * The abstraction makes this explicit
      * In this case the language is finite
* Understand their strengths and challenges as well as when they should be applied

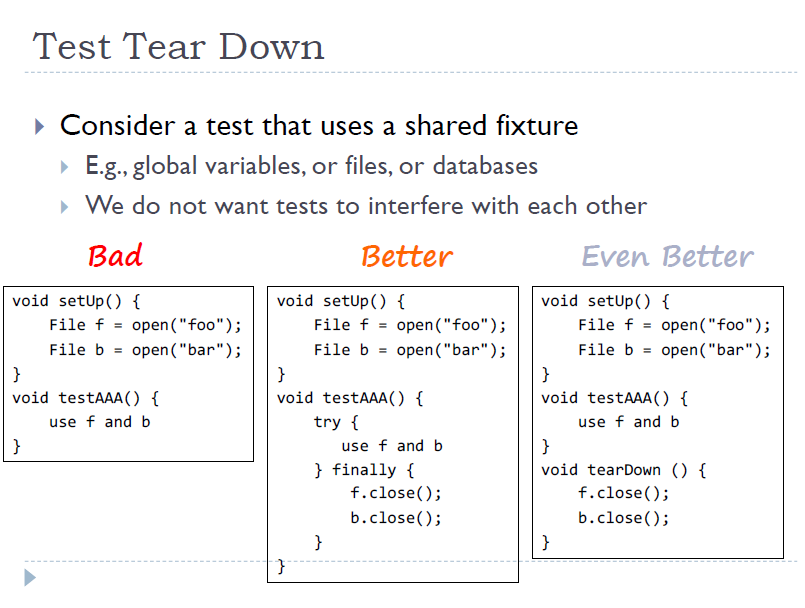
Version Control:

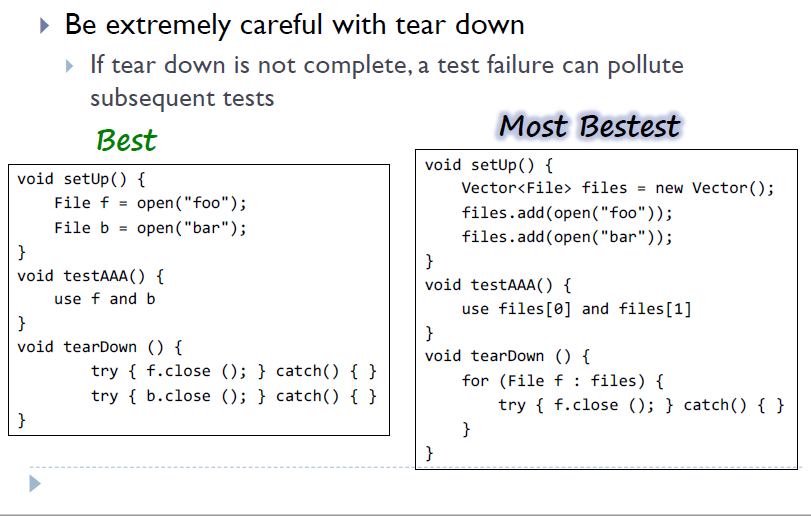
* Usage of git commands covered in class and homework assignment
* Be able to apply it to a specific coding example(programming language agnostic)
* Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later
* 
* Creating Repo:
  + Git init
* Basics
  + Git add .
  + Git commit -m “message”
* Inspection
  + Git status
  + Git log
  + Git diff
  + Git diff head
* History to Stage:
  + Git revert commit\_number
  + Git reset
* History to working directory
  + Git checkout -- x
* Create and merge branches
  + Git branch branch\_name
  + Git checkout branch\_name
    - Git checkout -b branch name
  + Git branch to inspect current branch
  + Git merge branch\_name
  + Get branch -d branch\_name
* Resolving Merge Conflicts
  + Either keep one of the different versions
  + Or Make modifications as needed
* Cherry Picking
  + Enables specific git commits to be picked by reference and appended to the current working HEAD
* Rebase
  + Rebasing is the whole process of moving a whole branch to a new base commit. Git takes your changes, and “replays” them onto the new base commit. This creates a brand new commit for each commit in the original. As such, your history is rewritten when you rebase.
* Remote Repos
  + Git clone
  + Git pull
  + Git push

Test-Driven Development:

* Writing effective asserts
  + Rather than writing “assertTrue(fFull.size() == 100 + size);” write “assertEqual(100 + size, fFull.size());” because the former will only say “Assertion failed: myTest.java:150 (expected true but was false)” on failure while the latter will say “Assertion failed: myTest.java:150 (expected 103 but was 102)” on failure. This is much more telling of what the problem may be.
    - Expected value is first argument in assertEqual
  + Fuzzy-equality assertions may be useful
    - assertAlmostEqual(int expected, int found, int tolerance)
      * Checks: expected - tolerance <= found <= expected + tolerance
  + Can create our own assertions as well
    - assertStringContains(string needle, string haystack)
      * More informative than assertTrue(haystack.contains(needle))
  + Test fails if an assertion fails or an uncaught exception is thrown
* Average of 1-5 errors/KLOC (1,000 lines of code)
  + >10 for prototype
* Testing: exercising software to try to generate failures
  + Purpose: reveal failures by running program
  + Limits: small subset of domain => risk of inadequate test set
* Static verification: identify (specific) problems by looking at source code
* Inspection/review/walkthrough: manual review of program text to detect faults
  + Limits: informal, uneven (presumably also error prone and time consuming)
* Formal proof: proving, starting from program source, that the program text implements the program specification
  + Limits: complexity, high cost
* Testing is running a program with a sample of input data
  + Dynamic technique
  + “Optimistic approximation”
    - Test exercised with small subset of all possible data
    - Assume behavior with any other input is consistent with behavior of selected subset of input data
* Types of testing:
  + Unit testing: verification of single modules
    - Focus on smallest unit of design
      * Procedure, class, etc
    - Test the following
      * Local data structures
      * Algorithms
      * Boundary conditions
      * Error handling
  + Integration testing: verification of the interaction among different modules
  + System testing: testing of the system as a whole
  + Acceptance testing: validation of the software against the user requirements
  + Regression testing: testing of new versions of the software against old tests



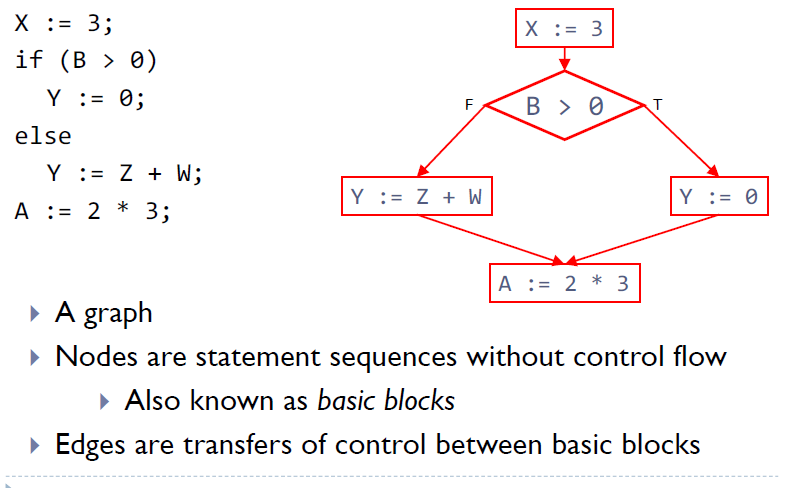




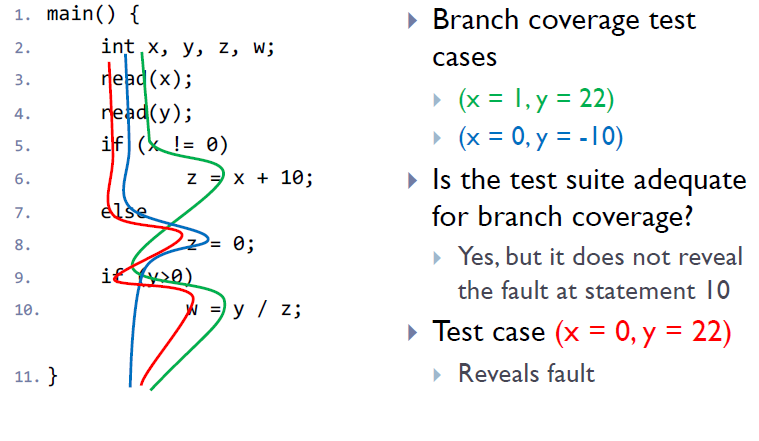
* Tests should be runnable by script
  + Increases frequency of testing
  + Otherwise one may eventually forget how to set them up and run them
    - Leads to tests being forgotten completely.
* Tests should verify their own results without human intervention
* Automated tests should be repeatable
* Automated tests should be robust
* Tests should be fast
* Basic TDD principles:
  + Test First
  + Automate Tests
  + Use “Test Lists”
    - Write tests you know are needed
    - Add more as you find new ones
  + 5-Stepp TDD “Rhythm”
    - Add a test
    - Run all tests, see new one fail
    - Make a little change to pass test
    - Run test, see them pass
    - Refactor to remove duplication
* Regression Testing
  + Idea
    - When you find a bug
    - Write a test that exhibits that bug
    - And always run that test when the code changes
    - So that bug doesn’t reappear
  + Ensures a test suite that grows with the code
  + Should be automatic
  + Can be expensive
* Nightly Build
  + Build and test the system regularly
  + Easier to fix the problem earlier

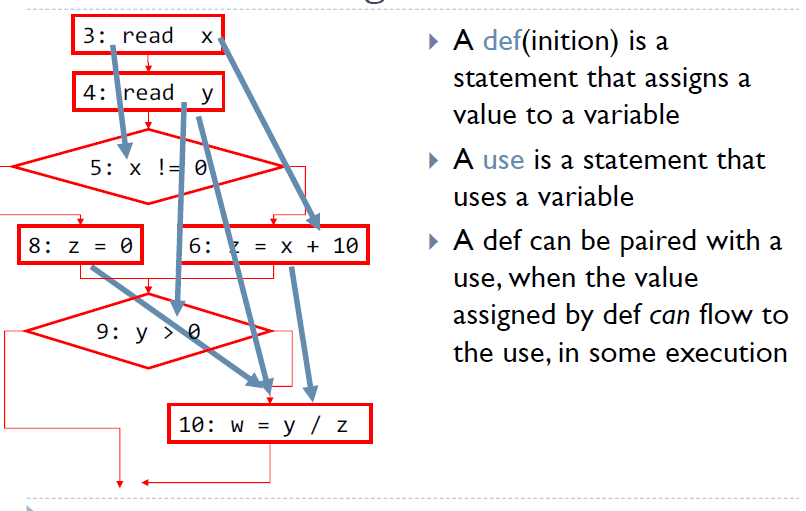
Code Coverage Analysis:

* Code that has never been executed is liable to have bugs
* Code coverage is
  + (# of elements executed by test suite)/(# of elements in program)
* Control flow graph drawing and interpretation



* Perform various forms of coverage analysis:
  + Branch
    - Edge in graph
    - Branch is “executed” when both outcomes have been tested
      * “Multiple condition coverage”
  + Statement
    - Node, basic block in graph
  + Path
  + Data Flow
    - Syntactic dependency





* Test requirements: def–use pairs in a program
  + (# of executed def– use pairs)/(# of def– use pairs in program)

