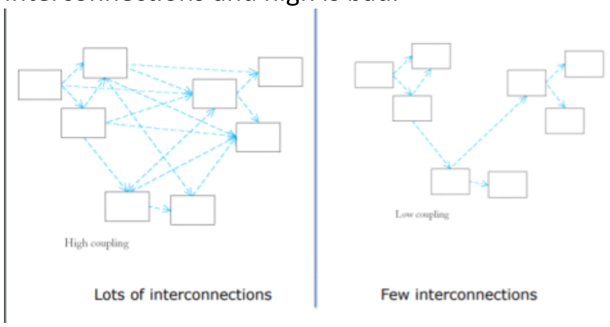
Unit: DESIGN

Monday, February 25, 2019 2:11 PM

- Modular Design
        - A module is an implementation unit that provides a coherent set of responsibilities
        - Coupling is a measure of how modules are interconnected, high coupling means lots of interconnections and high is bad.

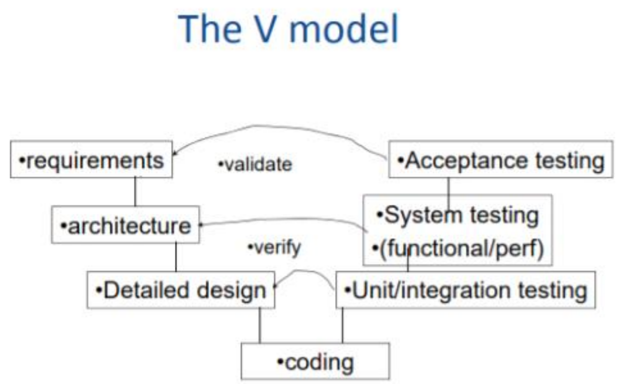


        - High coupling is bad because
            - if you change one module that causes a ripple effect
            - Assembly of modules requires more work
            - Module is harder to reuse and test since too much included
        - Cohesion is a measure of how strongly-related or focused the responsibilities of a single module are
        - Low cohesion is bad:
            - Difficult to understand modules
            - Difficult to maintain a system because change in module mean will have to change other modules
            - Difficult to reuse a module because apps don't need all the random operations provided in the module
    - Modular design
        - Modular design means high cohesion
            - Each module has clear and related responsibilities
        - Modular design means low coupling
            - Small number of interconnections between subsystems
        - Importance of modular design:
            1) Build on a budget!
                - Divide up the development and testing work
                - Don't have to wait for other parts to be done
                - Less time debugging
            2) Build maintainable systems
                - It is easier to pinpoint the cause of bug to a module and then focus and fix the module
                - It is easier to isolate and test before integrating
                - It is easier to understand the system as a whole and identify how to add new features
            3) Build reliable systems
                - You can understand the whole design and identify flaws and fix them
                - You can test each part and make sure they work right

    - Layered design

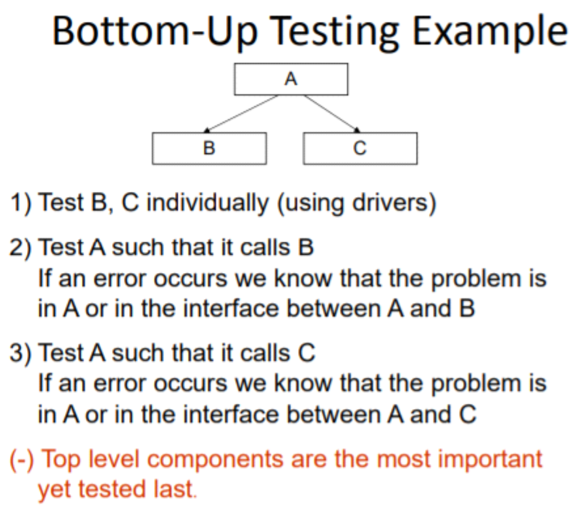
        - The system modules are organized into layers
        - Modules in upper layers are allowed to use the modules in the lower layers
        - Examples : operating systems, networking, ..
        - Benefits of layered:
            - Managing complexity
            - Maintainability - change in a layer can be hidden and/or substituted
            - A blueprint for constructing the system - specialize developers work on different layers e.g. GUI developers

- Integration Testing
        - V-Model

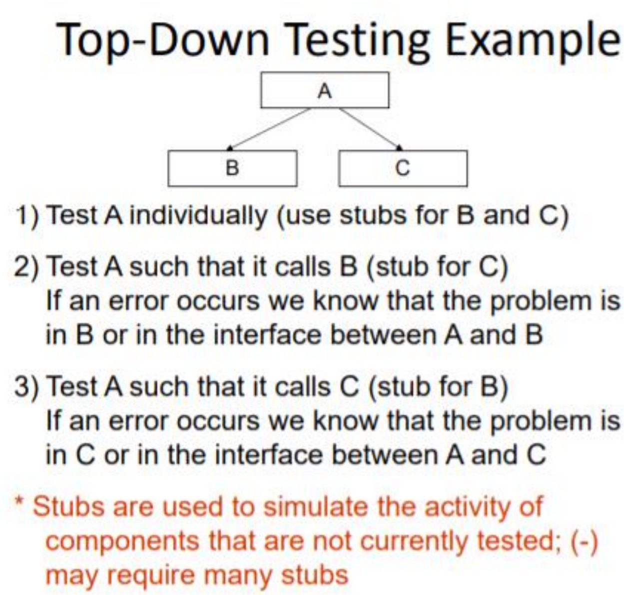


            1) Errors in upstream processes are more expensive to debug and fix
            2) Higher frequency of errors occurring in upstream processes
                - Testers should be involved in requirements and design phase
            - Inspections/reviews to trap errors from flowing downstream
            - What can you do during reqs?
                - Validate
                    - Show prototypes/screen sketches
                    - Design fit-criterion and corresponding acceptance tests
                - Verify
                    - Evaluate each requirement for correctness, ambiguity, testability, etc.
            - What can you do during arch/coding?
                - Design to be testable: controllable/observable
                - Plan out top-down and other integration testing mechs
                - Logging for debugging
                - Checkpointing for debugging
                - Preconditions, postconditions, assertions
        - Big-Bang Integration
            - After all components are unit tested we test the entire system
                - Impossible to figure out where faults occurred!
        - Others:
            - Sandwich Integration
                - TD and BU meet in the middle
            - Sync & Stabilize Approach

    - Bottom-Up Integration Testing
        - Each component at lower hierarchy is tested individually and then the components that rely upon these are tested

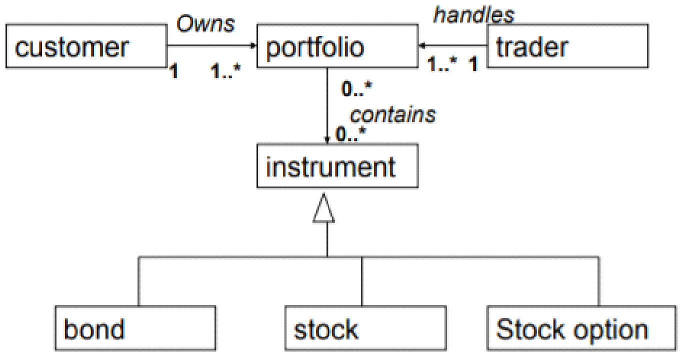


- Top-Down Integration Testing
  - Each component at higher position in hierarchy is tested individually; then the components that they rely upon are tested

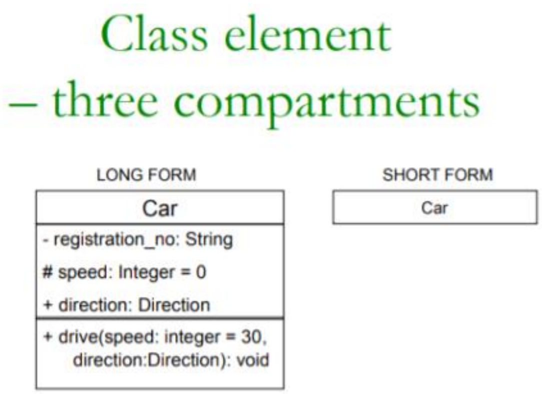


- Drivers
  - a routine that simulates a call from parent component to child component
  - Used in BU integration
- Stubs/Mocks
  - Stubs: a routine that fakes behavior of a child component
  - Used in TD integration
  - Mockito:
    - A testing tool that helps to create stubs and to verify that calls are made
    - When some method is called- do something
    - Verify that some methods were called to test interactions between methods

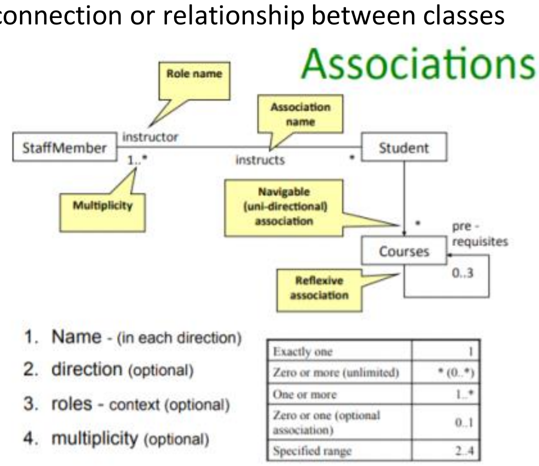
- UML
  - Class Diagram
    - Shows classes and their relationships
    - Example:



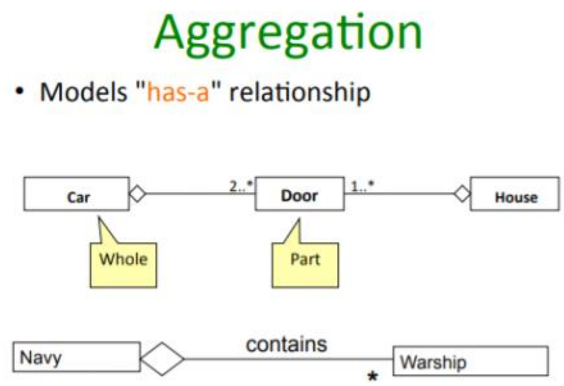
- A class diagram has two types of elements:
  1) Class elements



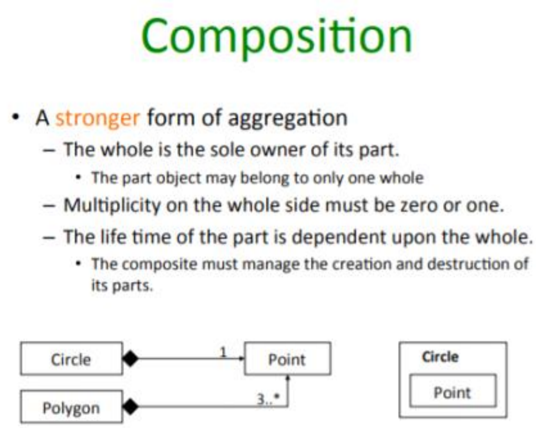
  - An attribute is a named property of a class that describes the object being modeled and appear in the second compartment
  - Attributes can be:
    - +public
    - #protected
    - -private
  - Operations describe the class behavior and appear in the third compartment
  - Specify an operation by stating its signature: listing the name, type, and default value of all parameters, and a return type
  2) Relationship elements
  - Associations- a broad term that encompasses just about any logical connection or relationship between classes



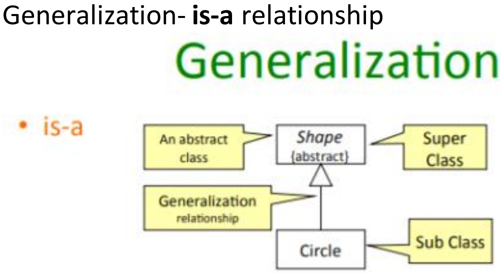
  - Aggregation- **has-a** relationship



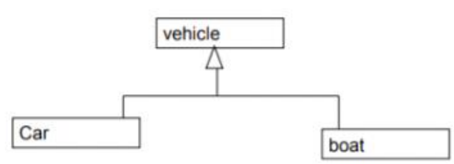
® Composition -parts whole relationship



® Generalization- **is-a** relationship



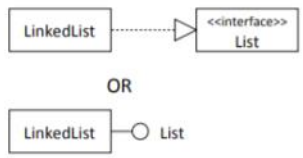
®



® Realization -implements or realizes relationship

Realization

- A realization relationship indicates that one class implements a behavior specified by another class (an interface or protocol).
- An interface can be realized by many classes.
- A class may realize many interfaces.

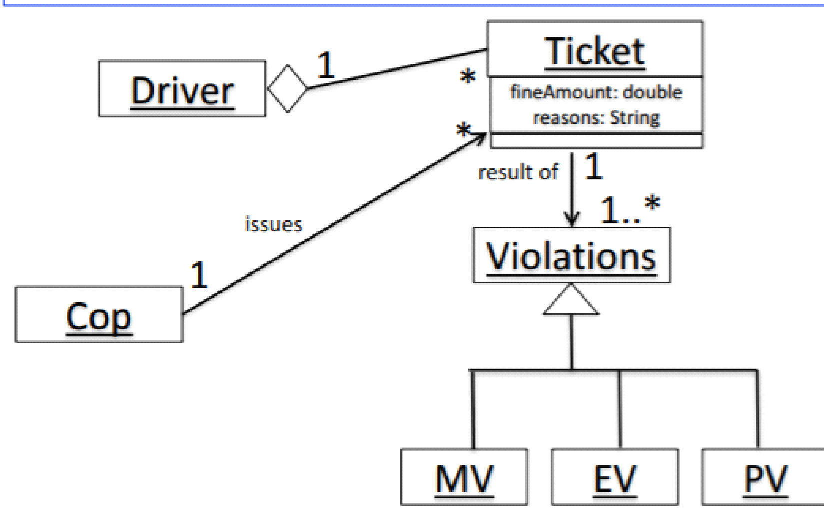


Class Diagram

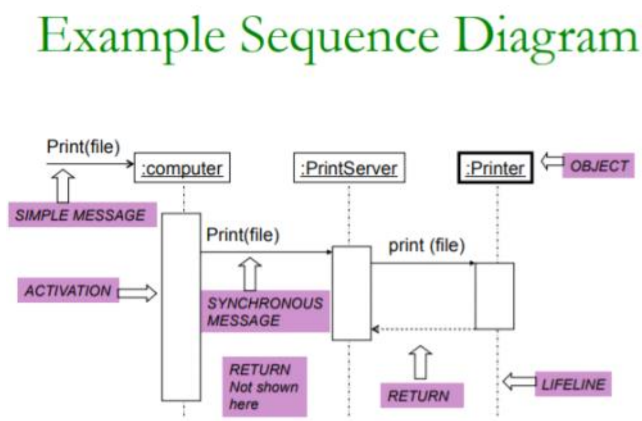
Draw a class diagram that captures ALL of the information in the below description.

1. A Driver has zero or more active Tickets.
2. A Ticket is a result of one or more Violations.
3. There are three types of Violations: Moving Violations, Equipment Violations, and Paperwork Violations.
4. A Ticket has attributes fineAmount and reasons.
5. A Cop issues zero or more Tickets.

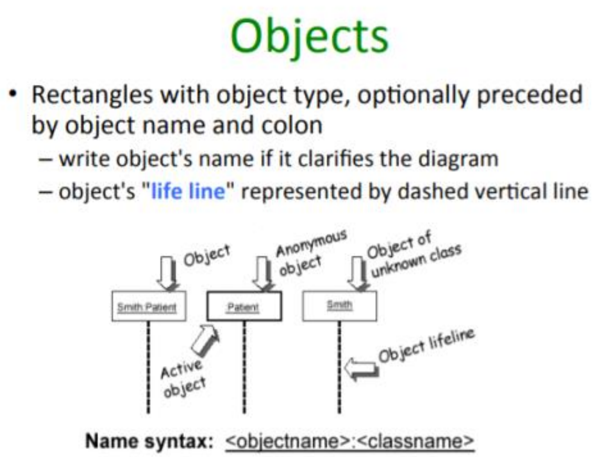
FINAL DIAGRAM

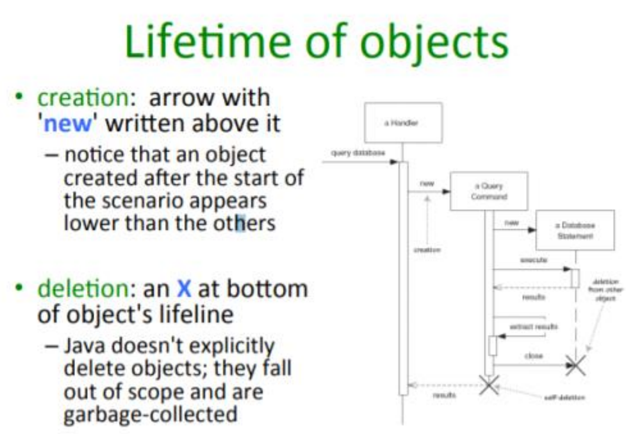


- Sequence Diagram
  - Shows interactions between objects over time

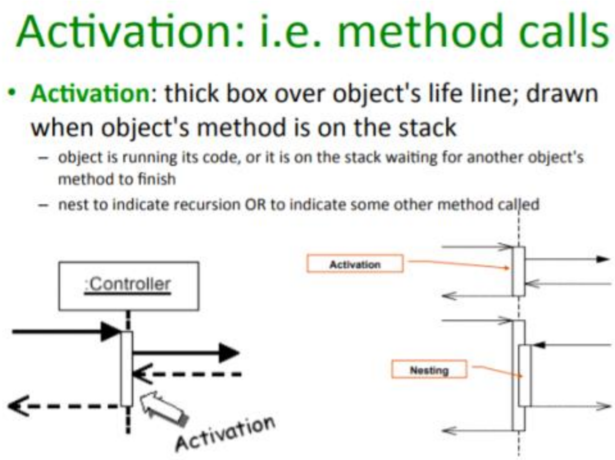


1) Objects- and not FUNCTIONS

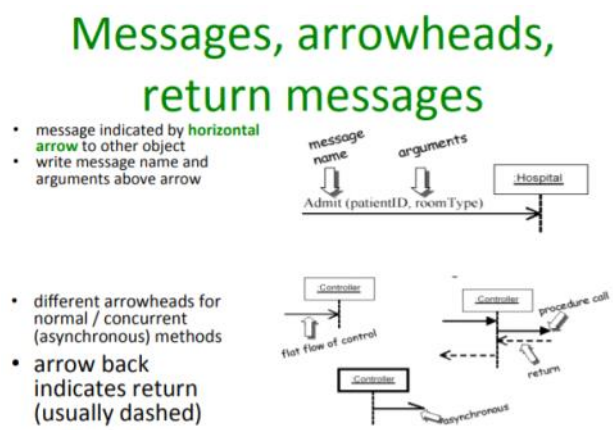




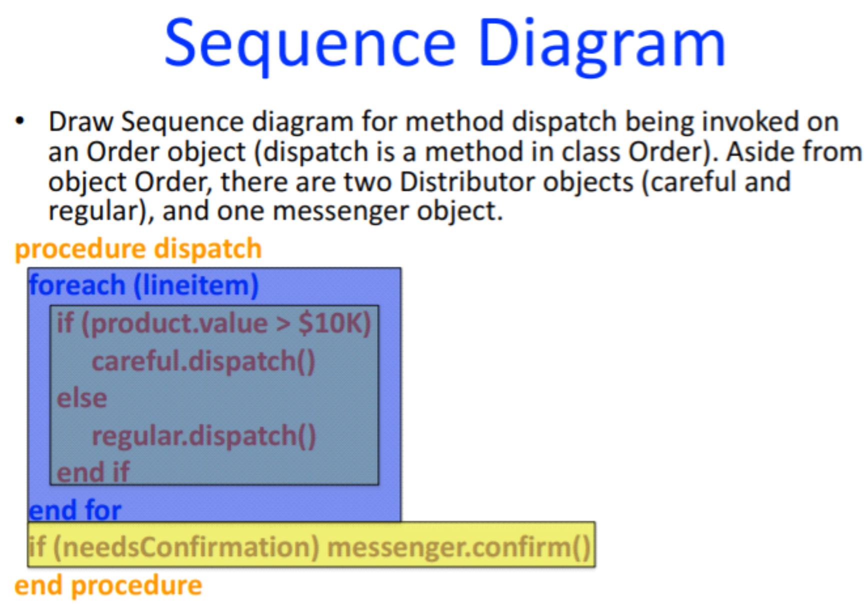
2) Activation - only when object's method is on stack (i.e. activated)

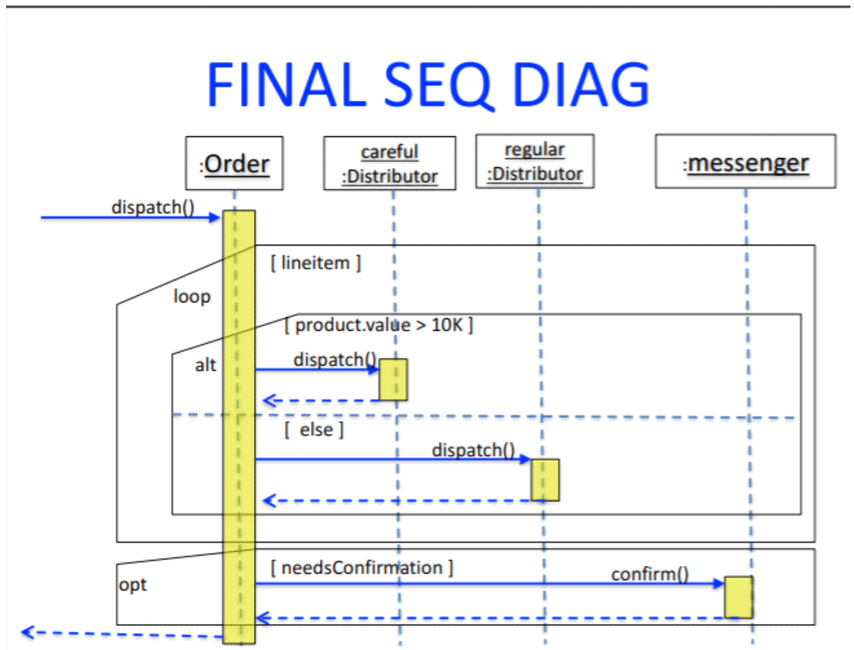


3) Messages are named
4) Correct arrowhead (synchronous vs async)
5) Return labeled with value (if needed)

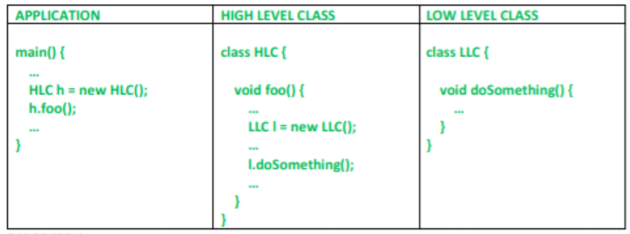


- Use a frame or box around part of a sequence diagram to show
  - If-then use OPT frame
  - If-then-else use ALT frame
  - Loop use LOOP frame
  - Method use REF frame
- Linking sequence diagrams - if one sequence diagram is too large or refers to another diagram, indicate it with either an unfinished arrow and comment or a ref frame that names the other diagram



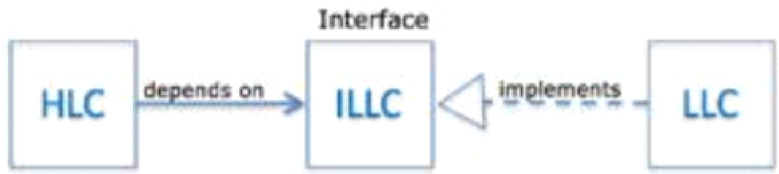


- Design Patterns
    - Enables reuse of software design ideas
    - Makes expert knowledge and design trade-offs widely available
    - Helps developer-developer communication by forming common vocab and helps in documentation and enhanced understanding
    - Eases transition to object oriented tech
    - To make code better think of how maintainers, utility developers, application developers work will be effected
  - Dependency Inject Pattern
    1) Problem
       - Diagram:



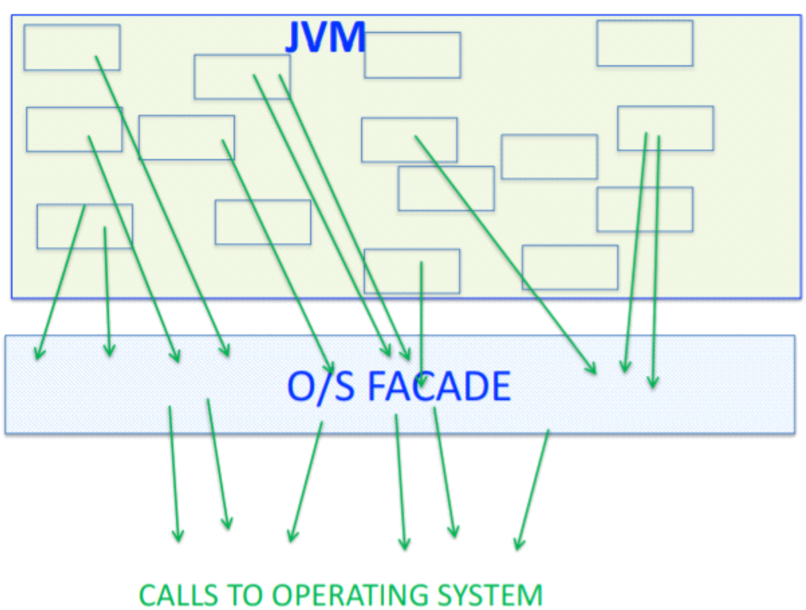
- For maintainers
  - HLC has LLC hardcoded
  - Changing databases means code needs to be changed
- For testers
  - Testing of HLC can't be done without changing source code of HLC
  - It is advisable to mock dependent classes to reduce the compile/run/debug cycle

2) Solution
- Diagram:



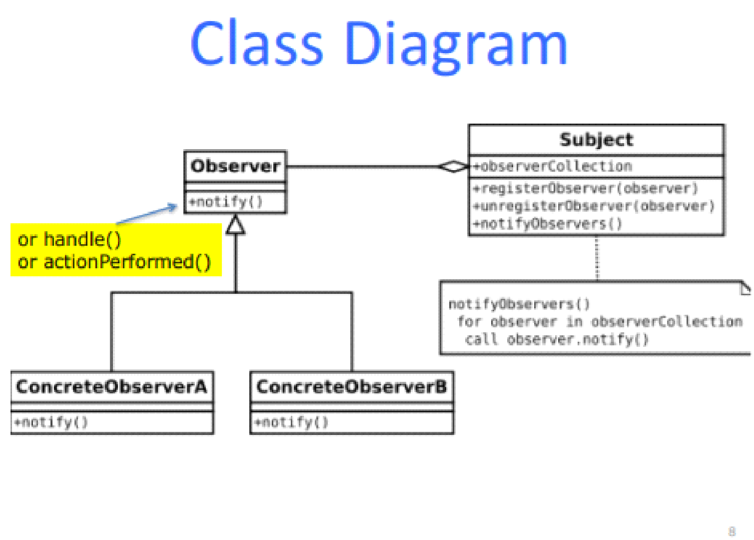
- Remove the direct dependency between HLC and LLC by introducing an interface
- Inject the LLC object into the HLC code using the HLC constructor

- Façade Pattern
  - Problem is we use #ifdefs in code to change behavior based on OS
    - Time consuming, recompile, retest, more bugs, halt development, not maintainable, messy code
  - Add a new layer the implements the OS façade for each different OS
  - Existing code makes calls to our OS façade instead of #ifdefs
  - Gets rid of #ifdefs, new ports don't need to change code, less errors introduced, less tedious



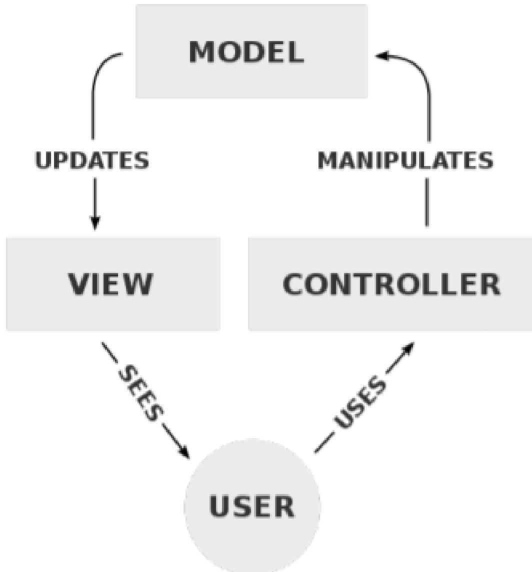
- Observer Pattern

1) Problem
   - Given two objects one of them wants to know when something happens to the other
   - Ex: line graph wants to know when the table entries are changed
   - Subject code should be unaware of specific observers, many observers should be able to observe the same subject, an observer should be able to observe multiple subjects
   - It is a bad idea to have subject aware of specific observer because it is hardcoding in class names (not modular), not able to have multiple observers
2) Solution Idea
   - Subject has a list of observers (implement observer interface)
   - When some change happens a notifyObservers() method is called
   - Observers can register/unregister themselves from a subject
3) Class Diagram



4) Benefits
   - All observers are notified automatically
   - Loosely coupled since subject doesn't know about specific observes
   - Observers code has no reference to specific subjects
   - Many observers on subject, an observer and observe many subjects
   - New observers can be added and removed without changing subject code

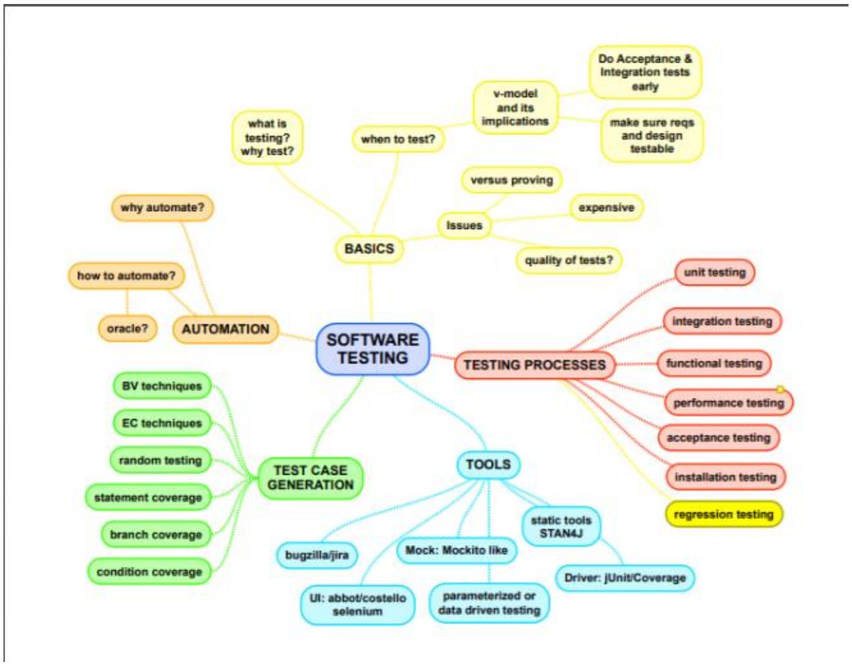
- MVC Pattern
  - MVC pattern has a model, one or more views, and one or more controllers



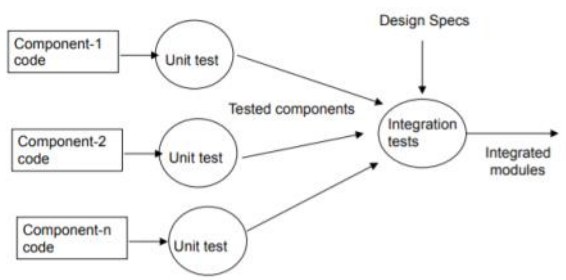
- Model - is in charge of storing/managing data
  - Store application data
  - Does not know about the view or controller
  - Provide interface for making changes to the data (set)
  - Notifies views of changes made to data
  - Provides interface for providing access to data (get)
- View - is in charge of displaying data
  - Draws or represents the model
  - Responds to changes to model events by registering handlers
  - Loosely coupled with Model
- Controller - is in charge of handling user events and controlling model and view operations
  - Registers and acts on user events
  - Updates model
  - Decides application behavior on user action
  - Controller knows about model and view
- Benefits:
  - Decouple view and logic from data
  - Separate into modules for separate dev/testing, views can be modified easily, additional controllers easy to add
  - Model can be used with multiple views
  - View can be used for other models too if separated by an interface

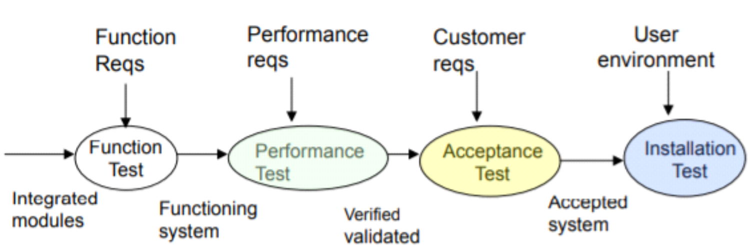
Unit: TESTING

Thursday, May 9, 2019 12:02 AM



- Basics of Testing



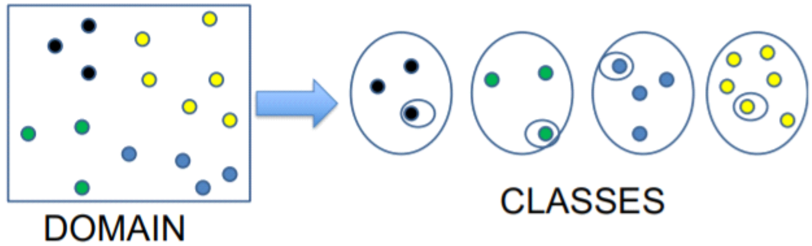


- Testing Processes
  - Testing Processes
    - Integration Testing
      - Assemble tested components to form the subsystem
      - Easier to integrate small pieces and test them than to integrate

the entire system and then test the whole system
      - Top-down or bottom-up
    - Functional Testing
      - Test all functionality per requirements
    - Performance Testing
      - Load tests, stress tests, recovery tests, volume tests, etc.
      - Reliability and usage
    - Acceptance Testing
      - Benchmark tests
      - Alpha test - pilot test run-in-house
      - Beta test - pilot test run at customer-site
      - Parallel testing - both existing and new system run in parallel
    - Installation Testing
      - Running tests at customer site to verify working of installed system
  - Issues in Testing
    - Regression testing
      - After changes have been made in software to check that it doesn't break you have to re-test after changes
    - Testing is not same proving
      - Goal is to find bugs in a smart way
      - Exhaustive testing is impossible
      - Black box- number of test cases/scenarios too large
      - White box - number of paths too large
      - There are infinite possible bugs and testing can't show bugs don't exist
      - More tests don't mean better testing
    - Testing is expensive, effort must be managed
      - How much percent of overall software development is devoted to testing? A good amount
      - Testing is an umbrella activity - can start once specifications are defined
      - Testing involves a lot of work
        - Specifying test cases, designing tests, creating tests, …
        - Risk based exercise - balance between cost of testing vs bugs missed
  - How do you select testcases?
  - How do you judge how effective your tests are?
  - How do you automate testing?
    - Automate Testing
      - Manual testing is expensive and error prone
      - Automate generate testcase, generate expected, oracle (is answer correct?)
      - Automated test case generation?
      - Drivers (Junit)
      - Oracles - checks if results are correct

- Test Case Generation
  - Blackbox- means you are testing from a functionality point of view, given a function how is it expected to behave
    1) You don’t have access to the source code
    2) You know what the code is supposed to do
    - Check whether it is working ok

- Boundary Value Testing
  - Vary one variable, hold value of rest of variables fixed at a value
  - Check min-1, min, min+1, nominal, max-1, max, max+1
  - Easy to generate
- Equivalence Class Testing
  - Domain partitioned into disjoint classes

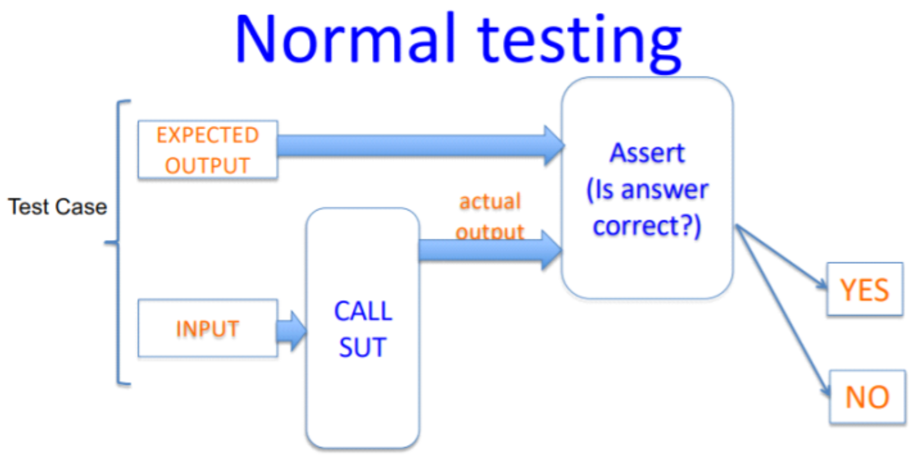


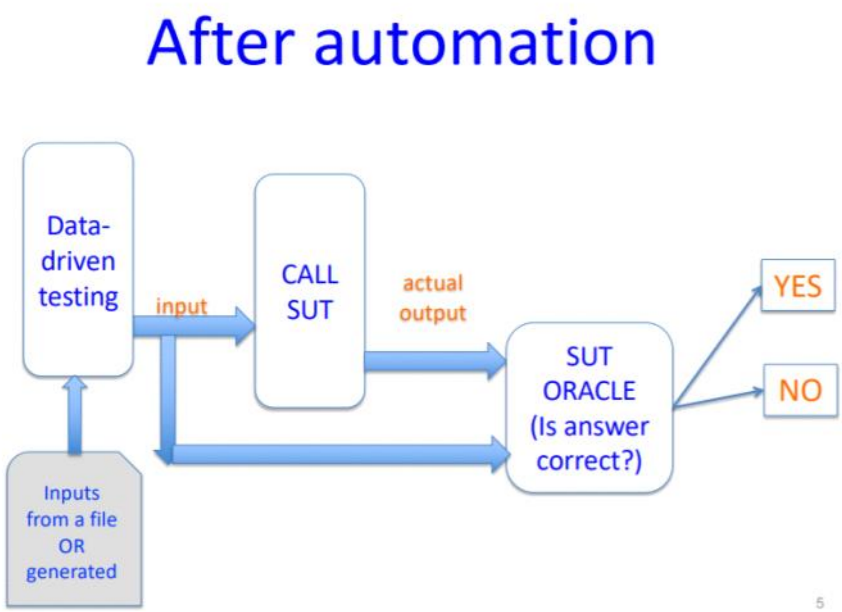
  - Each element of domain must belong to one class
  - Testing one element implies testing the whole class since all elements of a class are expected to behave similarly
- Random Testing
  - Generate random tests using operation profile (based on frequency of use during actual operation) as a guide to generate tests
- White Box - look at the source code and test the internal working for defects
  - Take black box tests and test code for coverage then think of more black box tests you may have missed and keep checking code coverage
  - Backtrack and determine inputs that would force tests to execute chosen paths
- Code Coverage
  - Code coverage metrics measure which parts of the code were executed by running all our tests
- Statement Coverage
  - A measure of the percentage of program statements that are run when tests are executed
  - We want to achieve 100 percent statement coverage
- Decision/Branch Coverage
  - Measure of how many decisions have been evaluated as both true and false in the testing
  - Objective is to achieve 100 percent branch coverage
- Condition Coverage
  - Reports the true or false outcome of each Boolean sub-expression of a compound predicate
  - Measures the outcome of each of these subexpressions independently of each other
  - Ensure that each of these subexpressions has independently been tested as both true and false
  - Helps generate tests cases because you find more example test cases that meet the conditions by matching the subexpressions that got missed

Automation

- Creating tests
  1) Make a list of test cases (input data and expected result values)
  2) Write n test codes for n test cases
  3) Run test code
- Automating
  1) Generate test data automatically

2) Write one test code per function, create an oracle
3) Test suites driver

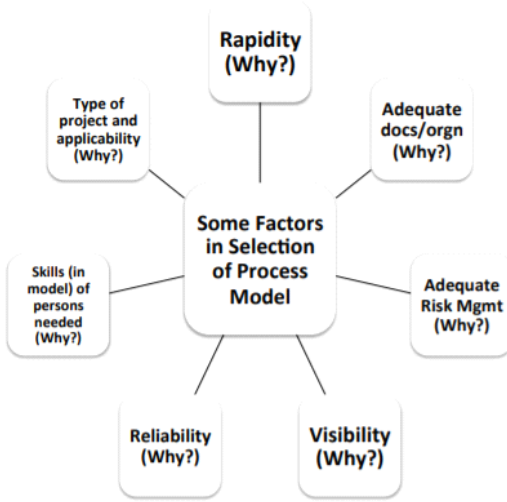




Unit: PROCESS MODELS

Thursday, May 9, 2019 12:44 AM

- What is a process model?
    - Failure rate for big software projects is 50-75%
        - management know project is likely to fail
        - Millions of dollars spent on abandoned projects
        - Examples: Alaska criminal information database abandoned , Australian census website collapses, Nest thermostat fails to heat homes, etc.
    - Software process models are general approaches for organizing these processes in a project. They prescribe:
        - When to start each process like reqs, design, etc.
        - What is criteria for transitioning between processes
        - They help project manager and their team to decide:
            - What work should be done
            - Task, milestones, deliverables
    - Different projects are handled differently
        - P1: you have to create a binary search program that the customer can use to search info in a data structure
            - Easy to plan accurately and few risks in project - use waterfall approach
            - Define interface, develop code, test code, finalize documentation
        - P2: create a topological sort for the customer
            - Harder to plan, known ways to solve, not risky - use prototype approach
            - Research, play around with algorithm, define interface, develop code, test
        - P3: create a library of different sort programs
            - Known sort algos, obvious increments - use iterative fashion
            - Plan of work divided into phases to develop and release each sorting algorithm
        - P4: sort program that returns results within 1 millisecond for big data
            - Very stringent reqs, many unknowns, very risky - use the spiral/iterative approach
            - Cycle1 - develop an algo and collect data, Cycle2- analyze data and propose solutions, Cycle3- develop a solution?
    - Selection of an approach
        - Choose based on the nature or type of project
        - Ask:
            - Are the reqs clearly defined?
            - Is quality/reliability important?
            - Does it have performance reqs?
            - Is there existing architecture?
            - What is the time-frame?
            - What is skill-level of team?
    - Management point of view
        - Concerns-
            - Visibility, how much work is left
            - Rapidity, time-to-market
            - Reliability, trapping errors
            - risks-handling



- Goals
  1) **High quality** software at
  2) **Low cost** and a
  3) Small cycle **time**
  4) **consistently**

- SCRUM process model
  1) Scrum Basics

- Scrum projects make progress in series of 'sprints'
- Typically 2-4 weeks
- Constant duration = better rhythm
- Product is designed, coded, and tested during each spring

  2) Scrum Roles
- Product Owner
  - Define the features of the product
  - Decide on release date/content
  - Responsible for profitability
  - Prioritize features as needed and according to the market
  - Accept or reject results
- The ScrumMaster

            - Represents management
            - Enacts the scrum values and practices
            - Ensures team is functional and productive
            - Enable cooperation between roles
            - Shield team from distractions
        - The team
            - 5-9 programmers, testers, user experience designers, etc.
            - Self-organizing, full-time

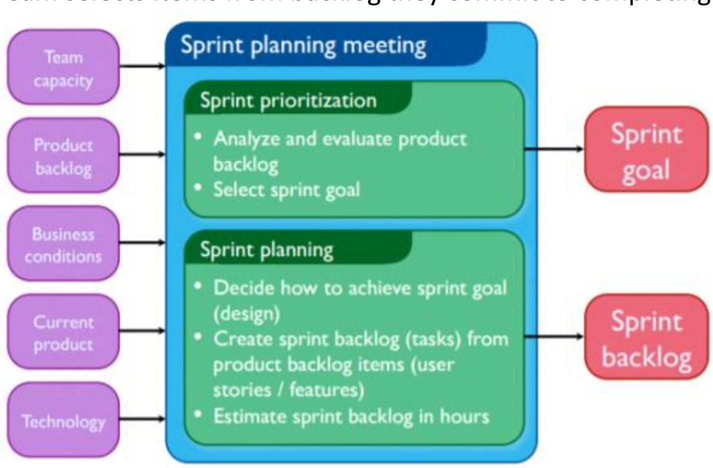
3) Scrum Artifacts
    - Product backlog
        - The requirements (user stories)
        - Each item is valuable, prioritized by product owner, reprioritized at start of each sprint
        - Example:

A sample product backlog

Backlog item	Estimate
Allow a guest to make a reservation	3
As a guest, I want to cancel a reservation.	5
As a guest, I want to change the dates of a reservation.	3
As a hotel employee, I can run RevPAR reports (revenue-per-available-room)	8
Improve exception handling	8
...	30
...	50

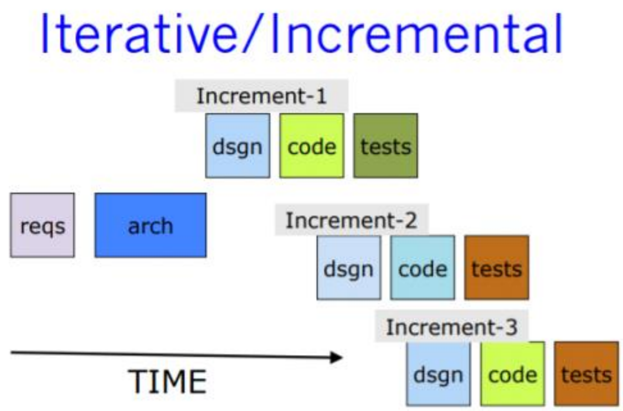
    - Sprint backlog
        - A TODO list selected from the product backlog
        - Estimate hours each task will take
    - Scrum Board (Trello)
        - User stories are tasks sorted into to do, in progress, done

4) Scrum Ceremonies
    - Sprint planning
        - Team selects items from backlog they commit to completing

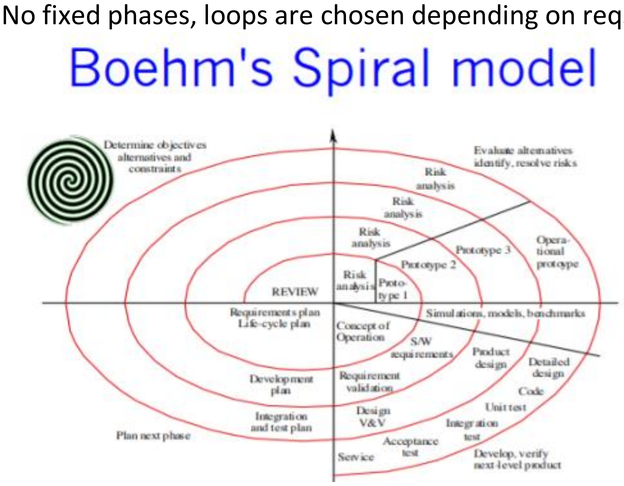


    - The daily scrum
        - What did you do yesterday? Will do today? Where are you stuck?
        - Not for problem solving, and not status checks
    - Sprint Retrospective
        - After every spring gather and discuss what continue doing, stop doing, start doing

        - Sprint review
            - Demo of new features
    - Cons:
        - Stressful, anxiety about productivity
        - Herded through small use-cases
- Other process models
    - Code and Fix Model
        - Only for small throwaway assignments
        - Poor reliability, visibility, can't distribute work, messy code, not maintainable
    - Waterfall Model
        - Reqs, architecture, detailed design, coding, testing
        - Linear order of stages
        - Simple to understand and manage
    - Prototyping -waterfall variant
        - Verify reqs with users
        - Try out design alternatives
    - Iterative/Incremental



        - Pros:
            - Quick time-to-market
            - Validate each step with user/feedback
            - Focus on area of expertise at a time
        - Cons:
            - Need maintenance and dev teams
            - Messy code
    - Boehm's Spiral model
        - Process is a spiral not a sequence
        - No fixed phases, loops are chosen depending on reqs



- Pros:
  - Works well with internal software development
  - Clear focus on planning and determining risks/alternatives
  - Flexible
- Cons:
  - Milestones not clear