**CS2800 Software Engineering**

Good software:

* + business value & timely
* fails early + responsive to customer needs

Software engineering

* techniques(good habits) + process + tools > everyone = good code (team) -> delivered to client

Good code:

* readable
* that works
* reusable
* comprehensive
* deliver business value

Agile:

* Individuals and interactions OVER process and tools
* Working Software OVER comprehensive documentations
* Customer collaboration OVER contract negotiation
* Responding to change OVER following a plan
* responsive process model
* understanding testing
* communication in teams

CASE (Computer Aided Software Engineering)

* support software process activities
* Javadoc, debuggers, Dropbox, VE, Compliers

Workbench (Unified of CASE)

* >2 CASE
* clean, easy invocation
* GUI Builder, Smart Code

IPSE (Integrated Project Support Environment)

* integrates tools and workbenches
* Eclipse

Javadoc

* @author
* @param
* @return
* @throws
* @see
* can use HTML
* 1st sentence = summary
* Follow Oracle style Javadoc

**TDD**

* Code and Test at the same time
* Black box (write tests before code)
* White box (write test to break code)
* Doc (tests document the code)
* Before code, think
* write TESTS that methods would use that was not written
  + write once, run often
* Tests indicate intent
  + drives code that developed
  + what it should do
  + part of documentation
* Programmers X testing, but like TDD
  + Testing = boring, another dep
  + But TDD = programming
* TDD encourages programmers to test
  + Test every single change, no more debugging
* TDD stops fear
  + Fear stops developers testing code, communicating
  + TDD = documentation
* Programmers = testers
* Writes test for complex algo
  + Write for error
  + Write for non-error (hardcode it)
  + Another test case, figure out the algo
  + Gains momentum

**Smell**

* Once and only once (OAOO) = dangerous smell
* TDD requires SETUP & TEARDOWN
* Tests = codes, must be well written
* TDD removes smells

Junit

* Test runner = software that run tests
* Test suite = collection of test cases
* Test case = response to input
* Unit test = test of smallest elements of code to test, e.g. a class
* Test fixture = environment
* Integration test = how well classes work tgt
* Mock objects = fake instances so it works
* @Before = init()
  + Creates text fixtures (creates and init)
* @After = cleanup()
  + Release any resources, files, DB
* @Test = testcase()
  + Methods that contains the tests
* “assert” = challenge the test results
  + Junit catches these errors , record, displays
  + assertTrue, assertFalse
  + assertEquals
  + assertSame, assertNotSame
  + assertNull, assertNotNull
  + fail()
* Notes
  + Size: > code of small sys
  + Complexity: test complex code
  + Effort: Repaid in reduced dev time
  + Behavior: clarifies idea how it behave
  + If reduce error = not too much efforts
  + Omit tests = super straightforward

**Coding Standards**

* Automation
* Avoid Common Errors
* Readable
* Improve the language
* Standard failures are really bad Code Smells

Industry standard = checkstyle (driven by XML)

**Refactoring**

* Same code in more than 1 place
* Unify
  + Create a loop
  + Extract the code into a utility method
* If Sibling
  + Extract code and put into parent
* If unrelated
  + Invoked from the other
  + Requires a new class
* LONGGG methods
  + Extract methods from them
* Feature Envy
  + A method is more interested other class’s data
  + Cure= move the method to its class
  + Use delegation to access the method
* Comments
  + Are a good smell (but used as a deo)
  + Codes are bad (that’s why need comments)
  + After refactoring, comments=superfluous
* Primitive Obsession
  + Reluctant to use small object
    - Money and currency
  + Replace data with objects
* Bad Smells
  + Type Embedded in Name
    - Types/names
  + Uncommunicative Name
  + Inconsistent Names  
    Eg. *Open()* should have *Close()*
  + Dead Code E.g. Commented code

**Releases**

A Version of all the files

* Bugs/Faults
* Wishes (changes we/user comm want to see)
* Features (implemented stores, partially/fully)
* Old/State/Incorrect Assumptions (nd to redsgn)
* Reliance on old Tech (need to update

**Version number**

Higher = more recent

* Keep tracks of who did what, when, which
* Tally with bug fixes
* Identify installations
* Manage releases

Repo

* Stores all versions
* Authorized users can RW these files
* Each version of file knows it’s parent.  
  links to parent/ancestors
* Remembers every change written to it
  + File
  + Directory
  + Addition/deletion/rearrangment

What should be under?

* UML Diagrams
* Documentations
* Individual Project work
* Eclipse projects

Never lose anything, Undo Changes

Access from any machine, team working

Systems team will backup Repo

**SVN Update**

* Bring working copy into sync with the “Head” of repo
* MOST FREQ command
* Stays up to date with team
* Leads to conflict
  + Solution: Update BEFORE Commit
  + Cannot commit if “out-of-date”
  + Subversion puts “Conflict markers”  
    3 files: working copy, old rev, new rev
  + SVN RESOLVED

**SVN ADD COPY DELETE MOVE**

* SVN ADD = Create new files and store in repo
* SVN DELETE = delete them
  + Class files need not be in repo
* SVN MOVE = move/rename/refactor
  + Will keep ancestors
* SVN COMMIT = change in repo
  + No one sees changes till u commit
  + Regular: update and status
* SVN DIFF = See the differences in files
* SVN REVERT = Undo changes

**Project Management**

* Manage thousands of releases
* TRUNK = no half-finished features
* Trunk/Branches/Tags
* Trunk = main branch
* Branch = copy of project files
* Tag = RO branch (ready for release)

**Working on a Branch**

* Copy trunk to subdirectory of the branches directory
* Check out branch (edit/commits/deletes)
* Affects branch only
* Trunk changes are important
  + Use **SYNC MERGE** (once a week)
* Before integration, do a SYNC MERGE and commit
* Latest version of branch and trunk will be identical except for branch changes

**Feature/Task Branch**

* Branch born and used until changes work
* Merged into trunk
* Feature branch deleted when not needed
* Not using branch = unstable
* Only using branch = not commit to trunk
* Trunk just need to pass regression tests

**(SYNC) MERGE**

* 2 main reasons
  + bug fixes may help u
  + finished features may interact with your changes
* COMMIT, MERGE (to trunk), COMMIT

**Reintegrate BRANCH** \*EXAM!!!

* Perform a final SYNC MERGE
* Checkout the trunk
* Reintegrate merge the changes
* COMMIT, then DELETE the branch
* COMMIT again

*Svn commit; svn merge (trunk); svn commit   
(Sync merging any trunk changes)*

*Svn checkout (trunk)  
(goes into trunk dir)*

*Svn merge –reintegrate (branch)*

*Svn commit*

*Svn delete (branch)*

**svn merge (trunk);**

sync merge from the last sync merge version.  
Diff = edit to the files  
Do it in the working copy (svn resolved required)

**Agile** = [code, test, release, repeat]

Tags = RO branches

Bug fixes without having to install major new release

**Trunk to candidate release (then tag)**

1. Trunk is copied to **candidate release branch**
2. Rigorous testing done
3. Bug fixes (cherry pick) merged back into trunk
4. When the candidate release branch is tested and stable, released into tag

**Candidate release to trunk (then tag)**

1. Bug fixes in the trunk are (cherry pick) merged into third release branch
2. Candidate release branch is again rigorously tested, and big fixes are cherry pick **merged back to the trunk**
3. After testing, released (tag) as **minor release**
4. Then become /tags/3.7

\*The repo ends up with a number of candidate release branches, and many tags

**Cherry picking merges**

\*One step at a time

1. Manual approach
2. Use the manual (svn merge) indv/range

**Ancestry**

If 2 files share common svn history, common ancestor

Diff between merge and diff

* Svn diff might not spot differences, svn merge would find conflicts

Rename LOCALLY = delete then add

* Loses ancestry

**DO NOT LOCAL COPY**

Copy from friend’s machine

**UML Unified Modelling Language**

* Need to model the application domain to reduce the gap between real world and software
* Structural models = structure of application domain
* Communication tool = analyst & users

Abstraction

* Generic Categories (Classes) of things (Objects)
* And relationships

Classes

* Generic type of things

Properties of Objects

* Attributes
* Relevant to application domain

Operations of objects

* Relevant problem specific operations
* Makes them indistinguishable

Class diagram = visual of communicating

* Classes and relationships
* Mainstream in all OOD

Relations

* Association with each other
* Should be named

5 kinds of relationships

* Dependency
  + Weakest
  + Class knows about another class
  + Calls to method of other class
  + Dependent classes will need to updated when the class on which depends changes
  + E.g. interface class
* Association
  + Designed to work together
* Aggregation
  + Involved 1 or more instances of other class
  + can be built and destroyed separately
  + no lifetime implication
* Composition
  + One class made up of >1 other class
  + Closely coupled
  + Destroy the container will destroy contents
  + One object contains the others
* Inheritance
  + One is a specialized version of other
  + stereotypes

Multiplicity (1, 0..1, \*)

Good Design: Coupling (EXAM!!!)

**Low Coupling**

* level of coupling = reliance of classes of each other
* high coupling = class cannot be modified independently (which is BAD)
* HARD to modify, test, reuse

**High Cohesion**

* Measures Design of individual classes
* Methods serves given class = high cohesion
* HARD to modify, test, reuse, understand

Refractor to achieve low coupling, high cohesion

Break up modules that are low cohesion

Related to code smells.

Design patterns originally, when we refactor

**Tasks**

Manageable part of dev process that can be done by one person in reasonable time

Faults: fix

Wishes: new functionality

Changes: improve functionality

Extensions: implement new user story

**Slicing**

Vertical Slicing

* Great for client, allows client to involved
* Have GUI, and limited (or fake) functionality
* Can run and executable

Horizontal Slicing

* Single classes are perfected to gradually build up project from core components
* Effort spent on perfecting indiv components
* Necessary for key central service components
* Delays client feedback

**Task Summary**

* Task assigned to individual
* Task will be completed in a short time
* Working on several tasks
* Whole project = tasks using horizontal/vertical slicing

**Use Case (vertical slice)**

* Derives from conversation
* Functional aspects, processes of the system
* Business/client processes
* Significant characteristics
* Simple (can talk to client)
* Requirements = help with acceptance testing

**Actors/Processes**

* Identify the primary element and processes
* Primary elements = actors
* Processes = use cases

**Not design doc**

* WHAT, not HOW
* Necessary functionality/requirements
* Actor = external to systems

**User Story**

* Features from user perspectives
* Not technical, understandable

“as a [user role], I want to [goal], so I can [reason]”

* Ongoing discussion
* Part of agile process (incremental functionality)
* Use Case diagrams = more upfront + stable

GUI Designer

* MOST IMPT, interaction btwn users and system
* Flow of interaction/ storyboard
* Layout managers = hard to uds in code without directly seeing the layout
* Legislative limitation: easier to see
* Time saving
* Modifiability

GUI using framework

* Framework = software components easily build software systems
* Can use designer to generate code
* Easy to uds Layout Managers and makes screens

**Design Patterns**

* Meeting same problems
* Rmb and communicate same pattern that solves
* Slot in whenever there’s the need
* Simplify structure of app
* Apply a design pattern in class diagram
* Design patterns (use stereotypes)
* Characterized by use: Architectural, Behavioral, Creational, Structural, etc.

**Creational Patterns**

* Control Create objects suitable to situation
* Uncontrolled = inefficiencies, complexity

**Singleton (simplest design pattern)**

* Restrict instantiation of a class to 1 obj
* Usage: one registry, one frame, one display, centralized management of resources, global point of access
* Hard to serialize and subclass
* USE private constructor

**Factory**

* Create objects without the client knowing how
* Usage: Creating graphics objects, Threads for networking, Web Image Galleries, Hotel rooms
* Class hierarchy decoupled
* Involves only one extra class where methods are used for building and returning objects
* Variety of objects implementing same interface
* Separates object creations from constructors = create complex objects which share initialization logic.
* No need to change classes
* Creation is costly (object pool)

**Structural: Façade (API)**

* Create a simplified interface to ease usageUsage: simple interface to complex systemsShield/Hide clients from complexityDecouples the client from underlying system1 method call = invoke sequence of methodsPGSQL very complicated, can hide behind, hide details behind simple object (JDBC)

**Structural: Adapter**

* Interface = wrong, adapt itUsage: interface to third party systems. Component reuse, specialize a general classCombines and modifies existing interface to make them usefulDecouples the client from underlying systemMotivation and Description same as façade

Façade hides messiness, Adapter connects.

**Structural: Bridge**

* Decouple abstraction from implementationUsage: Windowing, logging, db, abstraction for portabilityChanges to an implementation have no effect on the clientVariety of implementation on some sys without changing codeUses abstraction decoupled from differences in implementationCan be: whole abstract class hierarchy, delegate to actual implementation, works well with Factory, choose concrete implementation at run timeBehavioral PatternsCommon communication patterns (flexibility + decoupling)

**Behavioral Patterns**

* Common communication patterns (flexibility + decoupling)

**Observer (Listener)**

* Register to observe event raised by another object
* Decouples classes. (dont know details of observer)
* Used when freq changes in objects (reflected in another object without tightly coupled)
* New observers with minimal changes
* Good desgn=decouple AMAP=reduce dependencies

Observer INTERFACE (observable) defines:

* Operations for attaching/detaching
* Methods that observer must have to notifyChange in state = notifies any attached observers

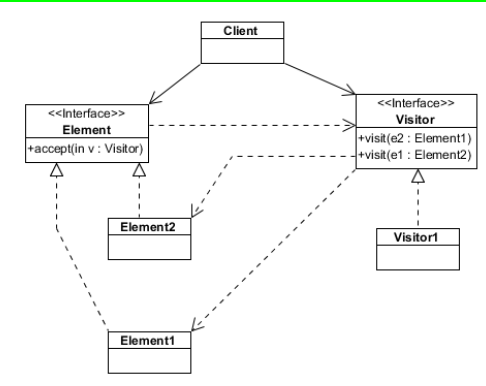
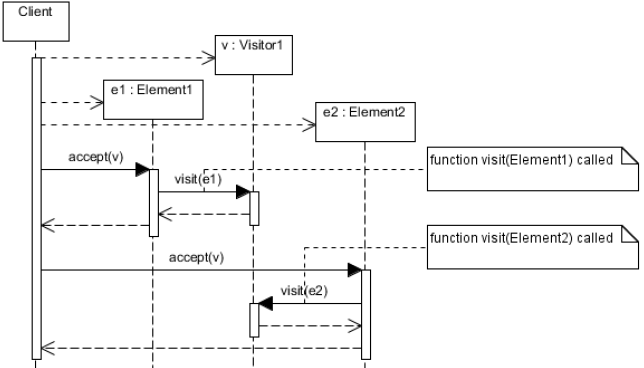
Observer implementation

1. Instantiating a concrete observable object (subject)
2. Attach concrete observer object (register interest)
3. State of observer subject changes (notify) =   
   (update) attached observers
4. New observer added = existing code remain unchanged/

**State**

* Allows an object to alter its behavior when it’s internal state changes. Object will appear to change class
* Simplifies programming
* Makes code simpler. No complex IF or SWITCH
* Create a state INTERFACE captures all behaviors that depend on state
* Create a class for each possible states
* Code behavior relevant to state

**Visitor**

* Separate an algo from an object structure
* An algo depends on type of arg
* Allows new algo on obj structure without changing obj
* Separate responsibilities, object hierarchy gets cluttered if every object must how to do everything
* Remove processing code from each object
* ****

**Architectural: MVC**

* Reduce complexity
* Increase reusability, flexibility and maintainability of code
* Decouple models and views
* New views are easy, model coding independent of interface

**View**

* View can be assembly of classes that react to user inputs and displays data
* User requires length processing, UI becomes unresponsive
* View observer of model

**Controller** uses Observer pattern to get req from view

* Registers listeners for user interaction events
* Calls methods in View and model to accomplish task

**Model** (independent of View and controller)

**Bugs**

**Mistake =** made by person becomes a **Fault**

**Fault =** becomes defect in exec code

**Anomaly =** fault becomes anomaly, when defective piece of code is executed = **Failure**

Fault remains **Latent**

**Testing** = discover failures, not **fault**

**Error =** measure of how incorrect results are

**Testing**

Cause failures = make faults visible = faults can be fixd

* Another purpose: Quality of code

Validation = building right software?

Verification = building the software right?

**Defects**

33% defects = failures less than 1 in 5000 years

<2% defects = failures >1 in 5 years

\*more about “process of looking” than code quality

**Testing**

**Black Box Testing** (functional) ignores internal mechanism, focuses solely on output generated

White Box Testing (structural) takes into account the internal mechanism of a system/component

**Test Scaffolding**

Scaffolding = built to support software development and testing

Simulates functions of component that don’t exist yet

Stub = simulate component

Drivers = test a module and provide test inputs/control

Mock Objects = temporary substitutes for real objects

Opacity and Scale

Opacity = tester’s view of code

Scale = tester examining bit or whole code/system

**Unit Testing**

Tests individual hardware/software units. Testers verify code that it does what it is intended to do a very low structural level. Calls methods with certain parameters.

* Whitebox
* Low level design, code structure
* Programmer that wrote the code
* Generally done within a class

**Integration Testing**

Components combined and tested to evaluate interaction. Verify units work together after integration

* Black/White Box
* Low and High Level
* Software developer
* Groups of related components, up to and including the whole system

**Functional and System Testing**

Plan test cases against high-level and customer req. Ensures functionality, putting new program in diff environment to ensure program works.

* BlackBox
* High level
* Independent testing team
* Whole system in real environments.

**Acceptance Testing**customer runs acceptance tests, based on acceptance criteria. Pre-specified by customer, and given to test team. Not a sub for functional testing

* Blackbox
* Requirements based
* Customer/independent testing team
* Whole System in real life

**Regression Testing**

Subset of original set of test. For the changes, run that modification causes unintended effects.

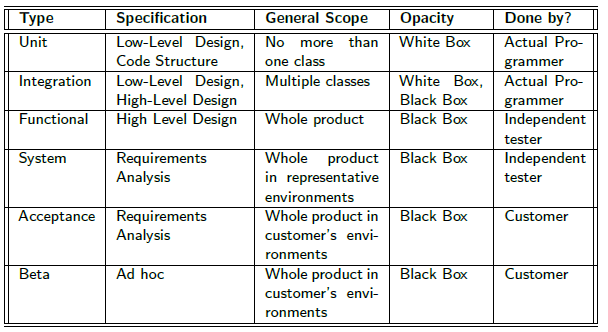
Smoke tests (subset of regression test): stable and all major functionality is present and works under normal condition. (for stability)

* Black and white box testing
* Any changed doc, high-level design

**Beta Testing**

Offered to potential users (beta testers) and they use as they wise. Reports any errors.

* BlackBox
* No specification
* Unpaid independent Beta Testers
* Use software in unexpected ways, avoid prejudice
* Many eyes/systems/environments = expose faults
* Unsystematic, hard to process and produce reports

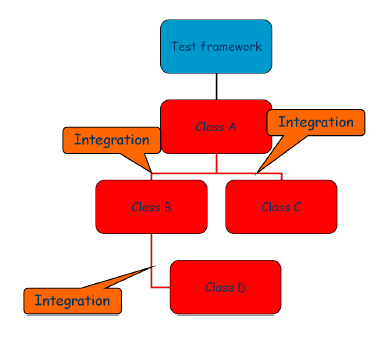


Unit Test VS Integration Testing

Unit Test = isolate single code unit, then test against behavioral specification.

* Lots of ‘scaffolding’ code
* Needs Driver & Stub Code
* Depends on specification
* Often testing functionalities not exercised in sys.

Integration/Incremental testing = reduce need of ‘scaffolding’ code. Use actual code modules that are developed.

* Carrying out unit tests + integration of sys across modules
* Often hierarchical (but not necessarily)

2 types

* Incremental
  + Reveals more errors/misunderstandings
  + Earlier identification of problems/easier debugging
  + More thorough (tests behavioral specification)
* Non-incremental

More Scaffolding

* + Top-down?
  + Bottom-up?
  + More effective use of test effort (system testing)
  + More concurrency

**Top-Down VS Bottom-Up** Incremental Test

* Dependent on structure of the system
* Layered architecture (bottom up)
* Generic (Top down) components in product provide service

**Top—Down**

* Commences with top modules
* Descends in layers through dependency
* Successive layers of “Stub” (Can be complicated)
  + Stub =should be simple
  + Seq of results for seq of calls. (needs to check)
  + Stub =needs to be replaced by real modules
  + Stub, if deep, = difficult

Top-Down Testing

* Major defects = likely at the top level
* Getting I/O functions in early, ease test writing
* Early demo of main functionality = helpful in req issues, boost morale
* Too much effort on stubs
* Stub complexity = more errors
* Defining stub = difficult
* May be impossible accurately to reproduce test conditions
* Some observations may be impossible to make
* Encourage idea that test and dev can overlap
* Encourage deferring full testing of modules.

Bottom-Up Testing

* Initiate testing with unit tests for bottom modules
* Next batch = depends on dependency structure
* Prioritization of modules for inclusion in test sequence based on “criticality”
* Helpful if errors are likely deep down in dependency
* Test conditions are easier to create
* Observation of test results is easy
* Reduced effort in creating stub
* Need to create driver modules (easier than stub)
* Entire system is subjected to smallest amount of test

Hybrid

Middle-out approach

* Non-iterative approach = testing them all at once (bigger granularity for integration steps)
* Based on:
  + Criticality: decide on groups of modules that most critical functions, which one to integrate
  + Cost: look for collections of modules with few dependencies on code. Reduce cost of creating stub

Summary

* Integration = little studied, few support tools
* Providing guidance on adequancy of integration testing to use coupling based coverage measures
* View sequence and scaffolding as part of sys release
* Data collected on faults detected in field = modify the integration sequence to provide more adequate testing of error-prone parts of system

**System Testing**

* Heterogeneous, what included = depend on app
* Can be expensive, time consuming, involve construction of physical components and software. (may exceed cost of primary software development)

**Capacity Testing**

* When: cope with high volumes of data (how they fail when capacity exceeded)
* What/How: construct a harness capable to simulating very large volume
* Why: ensure system fit for purpose
* Strength: confidence of capable of handling high capacity
* Weakness: unrepresentative, difficult to representative tests, long time to run

**Stress Testing**

* When: intend to react in real-time
* What/How: bursty traffic and environmental extremes. Test rig to test.
* Why: depend on conditions
* Strengths: essential kind of test avoidable
* Weakness: environment may be unrepresentative or may omit component

**Usability Testing**

* When: significant UI involved
* What/How: simulator OR have have group of users to test with clear objectives
* Why: safety issues, or produce smt more usable than competitor
* Strengths: Well-defined contexts = good feedback
* Weakness: hard to express usability req. Can test extensively and then not know what to do.

**Security Testing**

* When: open to public, vulnerable
* What/How: review the code based on bugs/errors. Form a team that attempts to break system.
* Why: essential systems, keep running
* Strengths: effort should go into design and use of secure components
* Weakness: only cover known ways in the checklist

**Performance Testing**

* When: meet performance targets stated in SLA
* What: modeling/simulation or in real
* Why: company charges for level of service (disputed when company fails to deliver)
* Strength: Good evidence of performance, modeling can identify bottlenecks/problems
* Weakness: issues with representatives tests

**Reliability Testing (MTTF)**  
**Mean Time to Failure**

* When: guarantee some system will only fail very infreq (important in telco)
* What/How: representative test set, gather info support statistical claim
* Why: guarantees about reliability satisfy regulator, market leader expectations
* Strengths: if test representative = accurate predictions
* Weakness: a lot of data for high-reliability, easy to be optimistic

**Compliance testing**

* When: selling in regulated market, to sell, needs compliance
* What/How: standardized test sets = good coverage of file system behavior
* Why: we can identify problem area, create tests to check condition sets
* Strengths: regulation shares cost of tests (develop capable test set)
* Weakness: tendency for software producers to orient towards compliance test (outside test= worse)

**Availability/Reparability Testing**

* When: avoiding long down times, another SLA/KPI
* What/How: similar to reliability testing – can see errors that cause components failures, see how long they fix
* Why: critical service (cannot afford interruptions)
* Strengths: similar to reliability
* Weakness: Similar to reliability – in the field faster to fix

**Documentation Testing**

* When: have documentation on how it should tested
* What/How: test set verifies doc set matches sys behavior. Get someone to do tutorial and point errors
* Why: users = confused if documentation fails
* Strengths: ensure consistency
* Weakness: Not particularly good on checking consistency or narrative rather than example.

**Configuration Testing**

* When: life of sys, config is altered
* What/How: usually record set of relationships and constraints between everything. Change components = check config maintained
* Why: config faults = cause failure
* Strengths: address increasingly common source of error (can automate)
* Weakness: only good as record of dependencies recorded in system. No universal approach

**Corridor Testing**

Informal test to get quick user feedback

Fault injection

Improve test coverage: introduce faults to test code paths, error handling code paths which otherwise might not get tested, as they are infreq executed. Robust safe code

Test Case:

* Purpose of test
  + Specification, requirements, behavior, structure
  + Expected outcome, test outcome

4 type tests:

* unit, integration, system, acceptance

Good Codes:

* Testing (it works)
* Code smells
* Standards
* Coherent, low coupling

“Cannot control what you cannot measure”

* Temperature
* No. of faults
* Code quality
* Code usability
* Code reliability
* Code length

Line of Codes (LOC)

Walston and Felix Estimation Technique

Documentation = 49 \* L1.01

LOC estimates:

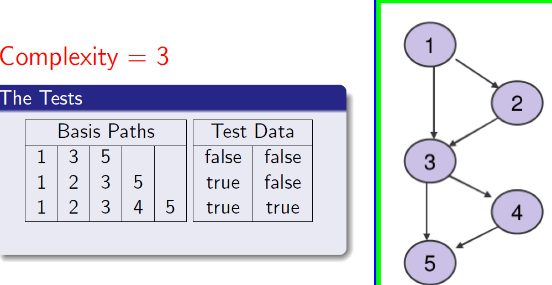
* Modifiability
* X Complexity
* Cohesion
* Progress
* Readability
* Faults
* X NOT productivity

Measure structure:

* Complexity = Number of different paths
* Long sequences = still ok
* If and while = more complex
* Control flow graph = directed graph in between 2 nodes.
  + Stop node = 0 outdegree
  + Every node lies on a walk from the start node to the stop node
  + Limit structure = structured programming

Test Coverage using Flow Graphs

* Statement: All codes executed
* Condition: Evaluation paths tested
* Path: Every route taken?
* Cyclomatic complexity = E - N + P
* OR number of Regions
* where,
* E = number of edges in the flow graph.
* N = number of nodes in the flow graph.
* P = number of nodes that have exit points



Basic testing = good to determine no. of tests required.

Checkstyle measures

* **Cyclmoatic Complexity** (1-4 = good)
* NPathComplexity (200 limit)
* JavaNCSS (method: 50, class: 1500)

Correlation

* Cyclomatic complexity affects no.of defects

Code quality:

* Low complexity

Exam is hard

Q1 is always the same 40% (**Define and Connect**)

Questions are NOT one thing.

Not all questions count the same.