Definitions

- Verification and Validation
    - They start as soon as we decide to build a software product, or even before. It can last as long as the software is in use
    - Software testing is nothing but validation and verification
    - Verification
        - Evaluate whether or not a product complies with some specification
        - Does it do what it says it does
        - Done internally in the organization
    - Validation
        - Evaluate whether or not a product meets the needs of the user
        - Does it do what it needs to do
        - Contractual acceptance testing between the user and the organization
- Error
    - A mistake in the code (the actual code)
- Fault
    - The result of an error (implementation of the code)
    - One fault may lead to one failure, one defect, or no failures
- Fault of omission
    - software was implemented without a certain specification and therefore a fault occurs
- Fault of commision
    - software was implemented incorrectly. The fault occurs because the developer did something wrong.
- Failure
    - Occurs when the code corresponding to a fault runs (the result of the code)
    - Failures result from faults which result from errors
- Incident
    - The observable symptoms associated with a failure
- Test case
    - An identifiable exercise associated with a program behaviour
    - It has inputs and expected outputs (pass or fail)
    - A set of inputs, execution conditions, and pass fail criteria
    - Initial test cases should be written to reliably test small, indivisible pieces of functionality, and subsequent tests should build upon these
    - A good individual test should reliably indicate whether the system is behaving correctly or not
- Test case specification
    - Part of adequacy
    - A recipe for tests
    - A requirement to be satisfied by one or more actual tests
    - Follows IEEE Standards
- Test case obligations

    - Part of adequacy
    - What needs to happen so the test can run?
    - A partial test case specification, requiring some property deemed important to "thorough testing"
    - Can come from software specifications, code, models or a list of common bugs
- Test
    - The activity of running one or more test cases
    - Our job is to determine the test cases for the item to be tested
    - Testing is performing tasks to answer questions and gather data
    - Software reliability is NOT evaluated solely by software testing
    - The activity of executing test cases and evaluating their results
    - The first place to start testing is in gathering requirements. Make sure you understand the user's needs. Test your specifications by asking questions.
    - When testing a function with a very large range of non uniform input, useful strategies are reduce and conquer, divide and conquer, use data category partitioning
- Test suite
    - A set of test cases
- Test consistency
    - Tests should be implemented consistently over short and long periods of time
    - Will help you to understand errors and compare results
    - Helps to answer the question, Is the outcome a fault? Or just a slight behaviour change?

Techniques

- Assessing requirements
    - An activity that is usually not automatable
    - This requires checklists and interaction with other people to validate the architectural design
- Process improvement
    - The activity of gathering data and conducting analysis to improve techniques and methodologies
- Assessing products
    - Assessing the product you're developing can be partly automatable through different types of testing
    - Test cases, test scaffolds, unit tests, integration tests, structural tests, system tests, regression testing, non functional testing
    - Code reviews can also be done by humans
- Non functional testing
    - Stress testing or load testing for reliability, durability and minimal maintenance
    - Not about how the program functions, but how well it performs (especially under pressure)
- Specification
    - A requirement or condition that must be validated

    - It is a stand in for the user but never a perfect replacement
    - You should still talk with the user to clarify misunderstandings in specifications
- Logical correctness
    - Conforming to an external standard to ensure consistent mathematical correctness
    - Not always feasible because of costs
    - Typically reserved for small systems or life dependent systems
    - Need to make sure all medical systems have been tested for logical correctness
- Undecidability / Halting Problem
    - Can happen when multiple users are trying to edit the same thing
    - Can also happen in users are trying to perform the undo/redo commands resulting in an infinite pattern of edits
    - Figure out how the system would handle these cases
- Pessimistic decidability
    - An inaccurate type of verification
    - The user rejects a program that has the desired property being verified
    - We say that there is a problem when there actually is not
    - Does not pass the test but there is no error
    - False positive - type 1 error
- Optimistic decidability
    - An inaccurate type of verification
    - The user accepts a program that does not possess the desired property being verified
    - We say there is no problem when there actually is
    - The test is passed, but an error exists
    - False negative - type 2 error
- Substitution
    - We may allow a specific type of testing if it is too expensive to test otherwise
    - Optimistic or pessimistic decidability can be used here
    - If P is too expensive to test, then test ^P (not P)
    - ^P => P, but P ⇏ ^P
- Statistic analysis
    - Analyzing a program without running it
    - Tools like Valgrid and CPPCheck
    - Check memory allocation, undeclared variables, memory out of bounds, seg faults, incorrect variable names
- Dynamic analysis
    - Analyzing a program through execution
    - Check inputs and output values

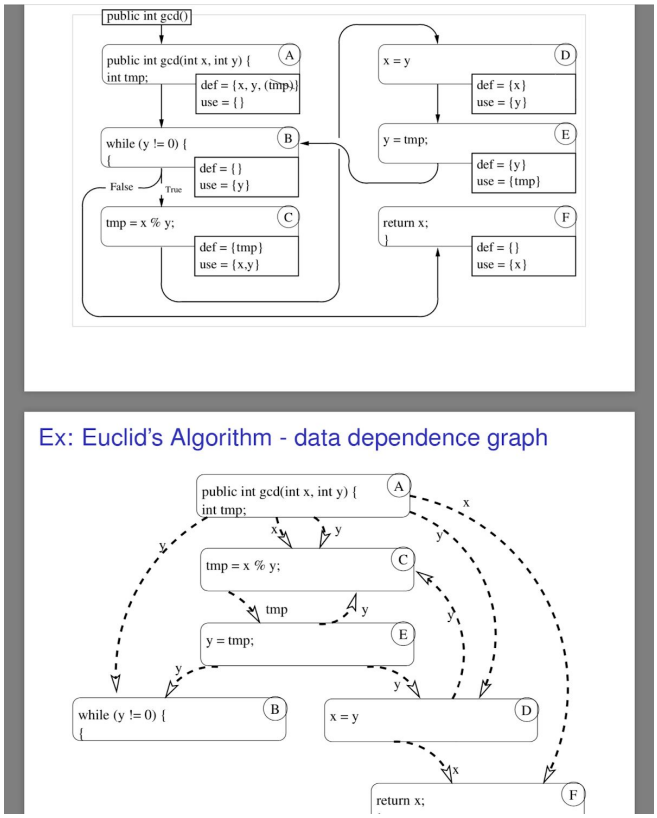
Graphs

- A graph is defined in terms of a set of nodes/vertices and a set of edges/arcs
    - Edges define the connectivity of a graph
    - Degrees of a node is the number of edges attached to it

- Directed graph
    - You can only move in one direction
    - Linked list, Trees
    - E = (n, m)
    - Directed path - Traversal over a sequence of edges that are connected
- Undirected graph
    - You can move in any direction
    - Networks, Leaving and entering a room
    - E = {n, m}
    - Path - A walk from one point to another in either direction
- Cycle
    - A path forming a closed loop where no edge or node repeats
- Circuit
    - Like a cycle but the same node may appear multiple times
- Model
    - Stands in for some artifact
    - Models are simpler but preserve key attributes. They are imperfect.
    - Given an accurate model of some system component, it is usually easier to prove that the model satisfies some system property
    - A good model could be just as expensive as creating the working product, so it is important to follow the specs for creating a good model to get the best bang for your buck
        - Compact - not too specific
        - Predictive - easy to use, good learnability
        - Semantically meaningful - obvious what it represents
        - Sufficiently general - can be applied to different technology (actual airplane or aircraft simulator)
- Control flow graph (flow chart)
    - Describe the movement within a procedure
    - Think of if/else sequences
    - Contains all paths, including some that cannot be executed
    - Basic block (nodes)
        - Maximal region of consecutive statements relevant for the problem
        - Must have a single entry and exit point
        - May have additional constraints due to data flow
        - Has no interesting actions happen within
- Call graphs
    - Nodes are procedures (methods and functions)
    - Edges represent the calls (printf calls strlen)
    - Represents all of the procedures to which a call might be bound
    - Context sensitive includes values passed into the methods
    - Context insensitive only includes method names
- Finite state machines
    - Nodes are states

    - Edges are transitions (how you got into that state)
    - Typically have an edge coming from nowhere to indicate the start state and an extra ring around any end states
    - Makes it easier to prove that the system satisfies all requirements than building the system itself
    - Mealy machine is when the edges are labelled
- Definition use pairs
    - Like a control graph but with boxes indicating what is being used and what is being defined at each node
    - Linking the produced (where the variable gets its value) to the accessed (where the variable used the value)
    - The point in a program where a value is produced (definition)
    - The point in a program where a value is accessed (use)
    - There can be many uses from one definition
    - If a new value is assigned, the old value is killed
    - A definition clear path leads from a definition to a use with no kill on the way
    - If there is a definition clear path for some variable v defined at node n to node k, we say that definition v at n reached k, or v at n is a reaching definition at k
- Data dependence
    - Similar to a definition use except extra arrows are drawn to indicate where the variables defined in node 1 are used in nodes 2 and 3
    - Transition of data
- State space is the complete set of possible states and transitions

Definition Use Pair Graph vs. Data Dependence Graph



Principles

- Sensitivity
  - It is better to fail every time than sometimes
  - A measure of how often a fault manifests
  - Is the cost different if we detect the fault at a different stage
  - Think of dependability
- Redundancy
  - The opposite of independence
  - Describes 2 or more parts of a system that constrain one another (such as a variable and its type)
  - Multiple copies of files or backup harddrives.
  - How long will the back ups take to retrieve? (monitoring)
- Restriction
  - Reduce the set of acceptable programs to those easier to test
  - Restrict certain tests from being executed
  - An undecidable program (often not enough data to make a decision)
  - Variable k was not initialized (java restricts it)
- Partition
  - Divide and conquer

    - Assert independence between components
    - For example, construct a model and then ask two **independent** questions
        - Does the model have the desired properties?
        - Is the model an accurate depiction of a program?
- Visibility
    - The ability to measure outcomes
    - Progress against goals
    - Observable states of a system
    - Attributes of the software
- Feedback
    - A process used to change processes to make observable errors less likely to occur in the future
    - Observable changes of a state
    - Monitoring of a process
- Quality process
    - Interacts with other development activities (completeness, timeliness, cost effectiveness, quality assurance)
    - Best predictor of defect repair cost
    - Defect repair cost is the time between introduction and detection
- Planning and monitoring
    - Needs identifiable goals
    - Analysis and testing strategy of the organization
    - pass/fail criteria
    - Schedule, deliverables, hardware and software requirements
    - Risks and contingencies
- Quality goals
    - External properties are groups into dependability and usefulness
    - Dependability, usability, reusability, maintainability
    - Latency, throughput, traceability
- Dependability
    - Correctness, reliability, availability
    - Mean time between failures (MTBF)
    - Robustness, hazards, safety
    - Think of sensitivity
- Analysis (Types of analysis include…)
    - Assessing whether a goal has been reached based on data
    - Statistic vs dynamic analysis
    - Root cause analysis

Adequacy

- An undecidable problem - just because a yes suite passes an adequate set of tests doesn't mean it is reliable and dependable
    - A student assignment won't be very close to adequacy where a medical imaging system will be extremely close to adequacy, as close as possible

    - Consider the broad range of design. A residence building won't need as durable of flooring as a commercial building where there is much more foot traffic
- Design rules do not guarantee good design
    - Good design depends on talented, creative, disciplined designers
    - Design rules help them to avoid or spot flaws
    - Test design is no different
- Inadequate test suite examples
    - Test specification described different test cases but the test suite does not treat them differently
    - A test suite missed a particular program statement
    - A test suite fails to satisfy some criteria (could be useful information for improving the test suite)
    - Even if a test suite satisfied all criteria, we do not know definitively that it is an effective test suite. Maybe it wasn't specified correctly
- Necessity and Sufficiency
    - Necessary: required to fulfil a purpose
    - Sufficient: fulfills a purpose to a desired degree
- Building Codes
    - A set of design rules
    - You wouldn't buy a house just because it is up to code, but you would avoid buying a house that is not up to code
- Adequacy criterion
    - A predicate that is true or false of a "program/test suite" pair
    - Satisfied or unsatisfied (binary)
    - A test suite satisfies an adequacy criterion if all the tests pass, and every test obligation in the criterion is satisfied by at least one of the tests in the test suite.
- Satisfiability
    - Sometimes no test suite can satisfy a criterion for a given program. You have 2 options:
        - Exclude any unsatisfiable obligation from the criterion
        - Measure the extent to which a test suite approaches a criterion (coverage)
- Comparing criteria
    - Empirical approach
        - study the effectiveness of different approaches to testing
    - Analytical approach
        - describe the conditions under which one adequacy criterion is probably stronger (more effective) than another
- Uses of adequacy criteria
    - Test selection
        - Which one gives the best coverage
    - Revealing missing tests
        - What might I have missed with this test suite?
    - Combine with other techniques

        - Design test suites from specifications
        - Then use structural criterion to highlight missing logic

Statefulness

- Is the behaviour of the code influenced by anything other than the current execution?
    - If it can be influenced by external variables, then it is stateful
    - If it can be influenced by system characteristics, then it is stateful
- What role does statefulness have on testing?
    - Is an unexpected state reached?
    - Has it been detected and how was it reached?
    - Is there test coverage for all explicitly defined states?
    - Does the state represent some pathological condition?
        - Low memory
        - Disk full
        - Missing asset
        - No network connection
- Program state refers to the set of values determining program function
- When parsing a string, does the program behave differently depending on what characters it has to read
    - The “ (quote) character might cause the program to change state
    - The characters read before the quote are treated differently than the characters after them
- fread and fopen are examples of functions that cause a program to change state
    - Different states may occur with write mode vs read mode vs append mode
- Size() method in java will differ depending on the size of the vector used as the parameter
- How to know if a program is stateful
    - Use equivalence classes
    - Call the same function multiple times with the same input and check the output
    - Use specific language constructs
    - Inspector methods

Types of Testing

- Functional Testing
    - What is specified and how do we test it?
    - Deriving test cases from program specifications
    - Black box testing (can’t see the code)
    - Specification based testing from software specifications
    - A testing technique where tests are designed in reference to specific points in a program’s documentation
    - Applies to all granularity levels (unit, system, regression, integration)
    - Functional specification

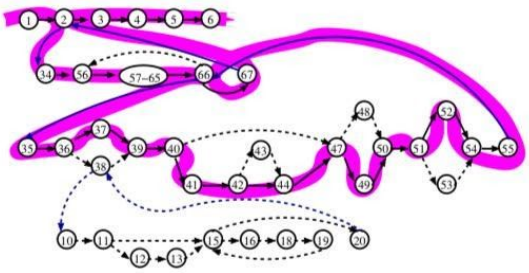
        - The description of intended program behaviour in enough detail for comprehensive tests to be created
        - Document that describes program behaviour
- Functional Testing principles
    - Timely
        - Helps refine and validate the specification
        - Provide talking points with the user
        - Testability and test cases can be constructed before coding
    - Effective
        - Finds some classes of faults that are hard to find otherwise
        - Helpful to find missing logic
        - Code based or structural testing will never focus on code that isn't there
    - Widely applicable
        - Any description and any level
        - Unit, system, integration, regression
    - Economical
        - Less expensive than code based tests
- Steps to turn a specification into a test case
    - Decompose the specification
        - Break it into independently testable features to be considered in testing
    - Select representatives
        - Representative values of each input
        - Representative behaviours of a model
        - Input and output transformations often don't describe a system
        - Use models for program specification design and test design
    - Form test specifications
        - Combination of input values or model behaviours
    - Produce and execute tests
- Random testing
    - Sample based on random choices
    - Every input is assumed to be equally valuable
    - Avoids designer bias
    - Can waste a lot of time testing uninteresting things
- Systematic testing
    - Try to select inputs that are valuable
        - Based on the understanding that some input is more likely associated with failure
    - Systematically allows us to focus on the dense parts of the partitioning haystack
        - Use functional testing to determine the regions of tests
        - Desirable property in partition principles is that each fault leads to failures that are dense in some class of input so that failures are easy to find
    - Better than random tests because less effort is spent testing uninteresting parts of the system
    - Quasi partition

        - Union of all partitions which must cover the entire space
        - It is possible for partitions or classes to overlap
        - Use the specification to partition the input (functional testing)
- Combinatorial testing
    - Identify interactions between distinct attributes and variables
    - Corner cases are likely where bugs will be found
    - Testing the possible combinations of functionality that are adjacent to one another
    - Tests grow at rates like n! (Exponentially)
    - Testing the individual pathways through the state space
    - Three types: category partition, pairwise, catalog based
- Category partition
    - Separate identification of values that characterize the input space (manual test values) from generation of combination for the test (automatic)
    - Divide input set into regions that may contain one or more failures. This reduces the search space and therefore effort is reduced
    - Testing steps
        - Decompose the specification into individual testable features
            - Identify parameters and environmental elements for each feature
            - Identify elementary characteristics and categories for each parameter and environmental element
        - Identify relevant values
            - For each characteristic or category, identify classes of values
            - Normal, boundary, special, errors
        - Introduce constraints
            - Rule out impossible combinations
            - Reduce the size of the test suite if its too large
            - No needs to test all possible combinations of errors
- Pairwise testing
    - Systematically test interactions among attributes of the program space with a relatively small number of test cases
    - Firefox on vista, chrome on linux
- Catalogue based testing
    - Build a catalog of experiences of test designers to aid in identifying attribute values. Examples include…
        - Empty lists cause trouble
        - What happens if the power goes out
        - What happens if you lose network connection
- Integration testing
    - Any time there is a boundary between two things or change from one thing to another
- Big Bang
    - Everything usually explodes
    - The idea of “winging it” or “trial by fire” and hope that it magically works

    - Things go really wrong really fast. It is difficult to establish where problems are coming from. Complexity may mask the existence of problems.
    - Use partitioning or divide and conquer to try to search for where errors may be coming from
    - Will have to develop a large number of test cases all at once as opposed to unit testing where you can do it gradually
    - Test for the most likely issues first
- Top down integration
    - Take the largest unites, stub them out (emulate the unit), and perform a test on that one piece of functionality
    - You'll end up hard coding a lot of values which indicate the correct responses.
    - Test at least one piece of input.
    - Replace the hard code with actual code and then keep testing until you reach the smallest unit of functionality
    - Pros
        - If you have a large complicated system you still have a shell to be able to test early in the project
        - Fail early fail often
    - Cons
        - Assumptions made about how units fit together
        - Incorrect assumptions about the API
        - May end up stubbing out the wrong thing resulting in wasted effort as a tester
        - Effort to do top down is significant
- Bottom up integration
    - Choose the most atomic or indivisible piece of functionality to test first (leaf) and then create scaffold to test those pieces of code (emulate the parent system)
    - Test as many different inputs as possible
    - Make sure organization of your code is good or you will have a lot of throwaway code
    - If you have a good design, you shouldn't run into many complications because you should be able to foresee the issues
    - Pros
        - Less throwaway code is usually generated
        - Traversal order is not particularly important
    - Cons
        - More complicated in reality
- Sandwich integration
    - Put together two parts that you know already work (top down and bottom up tested)
    - A big bang integration of an entire subtree
    - Stub out unimplemented functionality until you're ready to write that code
    - Pros
        - Easily traceable

        - Semantically clear
    - Cons
        - Artificial with respect to code design and development
        - More about project management than about software development
        - Stubs take a lot of work to develop, and must be maintained if regression testing or future development is to be supported
- Call graph integration
    - Look at related pathways
        - helps with finding out how errors are related to each other
        - Which pathways will be hit very infrequently?
        - Test those last
    - The problems you encounter first are probably the most important ones that you need to fix
        - Efficient way to find high risk defects.
    - Instead of thinking of functional modules, actually trace calls
        - Represent each separate module (or class, or group of cooperating classes) as a node, build a graph based on calls
        - Can do call graph top down
    - Pros
        - Call graph basis mirrors common development trajectory
        - Reduced need for stubs
    - Cons
        - Fault isolation is a major issue, as in any medium or large scale integration
        - Assumption is that call graph level entities give us sufficient information to make system wide quality statements, but this is not the case since the change in scope and scale is not so great
- Pairwise Integration
    - Instead of writing lots of stubs, use very focused testing of a linkage using the actual code
    - Design one integration test session per edge in the call graph (related to path testing)
    - An integration test session must exercise all mechanisms and states present in the linkage
- Neighbourhood Integration
    - Extend pairwise integration to the collection of edges incident upon a node
    - A neighbourhood of radius one is the set of all nodes reachable by a path length of one in either direction
    - Design one integration test session per neighbourhood (path of length one) in the call graph
    - Neighbourhoods will overlap
- Path Based Integration
    - Directly express system testing in terms of the behavioural threads (the thread defining a given set of actions)

    - Focus on testing interactions between units through interfaces
    - Source node / sink node
        - define the entry and termination of paths of a graph. There may be multiple of these per graph
    - Module execution path
        - Sequence of statements leading from a source to a sink



Strategies for Choosing Test Cases

- Input value category
    - If a set of values produce the same category of output, then we don't need to test all of them
    - Ranges help us find interesting boundaries to check
    - Ranges help us find non boundary values
    - Non range based sets are also of interest
- Collections
    - Is there a minimum valid number? Zero?
    - Is there a maximum valid number?
    - Is there a difference between running once and running many times? State change?
- Modules and Stubs
    - We can use these to simplify the system and remove extra complexity so that we can test a focused part of the system
    - Think of integration testing (top down and bottom up)
- Out of Bounds testing
    - Generally we simply treat this as another category of input
    - Each error state should be elicited
    - May involve exceptions
    - May involve return values with special code (null)
    - There may be a single mode of failure or several
- Measuring completeness using coverage
    - Coverage tells us how much of some measurable sort of work has been done
    - Types of coverage include
        - Path (node, edge, variable)
        - Input
        - Output

    - Node and edge coverage graphs
        - Call graphs
        - Control flow graphs
        - Dependency graphs
    - Variable coverage graph
        - Definition use pair graphs

Object Oriented Programming

- OOP Software
    - A unit is a class or small cluster of strongly related classes
    - Unit testing refers to testing inside classes (intra-class testing)
    - Integration testing refers to testing between classes (inter-class testing)
    - Dealing with single methods separately is usually too expensive (complex scaffolding)
    - Therefore, methods are tested in the context of the class they belong to
    - It is common for object oriented code to have more complex control flow that code written in procedural languages
- Characteristics of Object Oriented software
    - State dependent behavior
        - Different values in data members mean different responses from methods
        - For example, in the java vector class, a different number is produced by the size() function depending on how many values are stored in the vector
    - Encapsulation
        - Methods and data are bound together into an instance of the class
        - Methods and data are accessible through the same reference
        - Data is hidden and not accessible except through the methods (encapsulated)
    - Inheritance
        - There is a supersetting relationship between a parent and child
        - All of the behavior of the parent is available within the child
        - Virtual methods may override parent behavior and replace it with child behavior
    - Polymorphism and dynamic binding
        - A child can be made to behave like its parent by casting
        - Some languages also support this through interfaces
        - This implies that there may be a variety of interface definitions through which this object may be seen
        - One variable potentially bound to methods of different subclasses
        - The number of tests required when adding new classes when using dynamic binding grows combinatorially
        - Dynamic binding is just runtime polymorphism
        - Think of overriding the toString() method in different classes. Then you can call toString on any object and it will behave according to its type
    - Abstract classes

        - Abstract classes are only partially defined
    - Generic classes
        - Developed using a template standing in for some type
        - The type template will be filled in later (before or at instantiation)
    - Exception handling
        - Exceptions break the traditional statement flow by leaping to a different location on the data flow graph
        - They may leap several call levels up the stack
        - Functions and methods that are leaped over will never generate a return value
        - Exceptions create implicit control flows and may be handled by different handlers
- Procedural software
    - A unit is a simple program, function or procedure
    - A unit of work may correspond to one or more intertwined functions or programs
- Intra-class testing (inside classes)
    - Functional
        - super/subclass relations
        - State machine testing
    - Structural
        - Augmented state machine
        - Data flow model
        - Exceptions
        - Polymorphic binding
- Inter-class testing (between classes)
    - Functional
        - Hierarchy of clusters
        - Functional cluster testing
    - Structural
        - Data flow model
        - Exceptions
        - Polymorphic binding
- Scaffolding
    - Used when a state is encapsulated and we can't tell whether a method has had the correct effect
    - Used when classes are not complete programs and additional code must be added to execute them
    - Includes stubs, oracles and drivers
- Scaffolding approaches
    - Think about the controllability and observability of your approach
    - Is your approach general enough to be reusable for other projects
    - Is your approach project specific? Class specific? Test case specific?
- Oracles

    - Test oracles must be able to check the correctness of the behavior of the object when executed with a given input
    - Behavior produces output and brings an object into a new state
        - We can use traditional approaches to check for the correctness of the output
        - To check the correctness of the final state we need to access the state
    - The purpose of an oracle is NOT to see outside of class instances and evaluate the external state of the class
- Accessing the state
    - Intrusive approach
        - Use language constructs (c++ friend state)
        - Add inspector methods
        - In both cases we break encapsulation and we may produce undesired results
    - Equivalent scenarios approach
        - Generate equivalent and non equivalent sequences of method invocations
        - Compare the final state of the object after equivalent and non equivalent sequences
- The combinatorial problem
    - Polymorphism and Dynamic Binding
    - There may be many possible combinations of dynamic bindings just for one method
- Combinatorial Data flow approach (sd 20-22)
    - Identify a set of combinations that cover all pairwise combinations of dynamic bindings (definition and use pairs)
    - Derive a test case for each individual possible polymorphic def/use pair
        - Pairwise combinatorial selection may help in reducing the set of test cases
        - We need tests that bind the different calls to different methods in the same run
- Inheritance
    - When testing a subclass we would like to re-test only what has not been thoroughly tested in the parent class
    - We should still test any method whose behavior may have changed
    - It is usually necessary to retest an ancestor class hierarchy when introducing new subclasses even if the hierarchy has already passed testing
- Reusing tests
    - Track test suites and test executions
        - Determine which new tests are needed
        - Determine which old tests must be re-executed
    - New and changed behavior
        - New methods must be tested but other inherited methods do not

        - Redefined methods must be tested but we can partially reuse test suites defined for the ancestor
- Testing history
    - Abstract methods and classes
        - Design test cases when abstract methods are introduced even if it can't be executed yet
    - Behavior changes
        - If another method changes its behavior, we do not need to consider our current method as redefined
- Testing exception handling
    - Impractical to treat exceptions like normal flow
        - Too many flows - every array subscript reference, memory allocation or cast
        - Multiplied by matching them to every handler that could appear immediately above them on the call stack
        - Many are actually impossible to test
    - So we separate testing exceptions
        - Ignore program error exceptions
        - Test to prevent them not to handle them
    - Test each exception handler and each explicit throw or rethrow of an exception
- Testing program exception handlers
    - Local exception handlers
        - Test the exception handler
        - Consider a subset of points bound to the handler
    - Non local exception handlers
        - Difficult to determine all pairings of points and handlers
        - Enforce and test for a design rule: if a method propagates an exception, the method call should have no other effect

Planning and Monitoring

- Planning
    - Scheduling activities
        - May have to revise as the project goes on
        - What steps? In what order?
    - Allocating resources
        - Typically start with a smaller team
        - As the project progresses and becomes larger, slowly integrate more people
        - Who will do it?
    - Devising unambiguous milestones for monitoring
- Monitoring
    - Judging progress against the plan
    - How are we doing?
    - Are we at where we said we would be?

    - If you need to change the direction of the project, do so proactively
- A good plan must be transparent/have visibility
    - Ability to monitor each step and to make objective judgements of progress
    - Wishful thinking and denial are an enemy of quality
    - Different teams and people in the organization should understand what's happening so they can better do their work
    - People will leave the project and take key expertise with them, so you need to be able to adapt for that. Multiple people should know whats going on, not just one.
- Quality Process
    - A set of activities and responsibilities
    - Focuses primarily on ensuring adequate dependability
    - Concerned with project schedule or with product usability
    - A framework for selecting/arranging activities and considering interactions and trade offs
    - Follows the overall software process in which it is embedded
        - Waterfall software process would focus on the V model which means unit testing starts with implementation and is finished before integration
        - XP and agile software processes would put emphasis on unit testing and rapid iteration for acceptance testing by consumers
- Cleanroom Process
    - Thorough design activities and more formal verification will detect almost all issues
    - Statistical testing activities and responsibilities are focused on quality
    - Incremental development planning is integrated into an overall development process
    - Focus on defect prevention over defect removal
- SRET (Software Reliability Engineered Testing) Process
    - Spiral model with rigorous testing at each step
    - Development testing to find bugs early
    - Certification testing to confirm final product (validation and verification)
    - Activities and responsibilities of development of operational profiles is focused on quality
    - Design and implementation is integrated into an overall development process
- XP (Extreme Programming) Process
    - Choreography vs orchestration
    - Generality, project visibility and quality communication over formal structure
    - Fast iterations
    - Continuous testing and analysis
    - Creating unit test activities and responsibilities are focused on quality
    - Incremental released is integrated into an overall development process
- Quality planning
    - The cost of detecting and repairing a fault increases as a function of time between committing an error and detecting the resultant faults

    - An efficient quality plan includes matched sets of intermediate validation and verification activities that detect most faults within a short time of their introduction
    - Validation and verification steps depend on the intermediate work products and on their anticipated defects
- Verification steps for intermediate artifacts
    - Internal consistency checks
        - Compliance with structuring rules that define well formed artifacts of that type
        - A point of leverage - define syntactic and semantic rules thoroughly and precisely enough that many common errors result in detectable violations
    - External consistency checks
        - Consistency with related artifacts
        - Often conformance to a prior or higher level specifications
    - Generations of correctness conjectures
        - Correctness conjectures lay the groundwork for external consistency checks of other work products
        - Often motivate refinement of the current product

Strategy	**Plan**
Scope: Organization	Scope: Project
Structure and content based on organization structure, experience and policy over several projects	Structure and content based on Standard structure prescribed in strategy
Evolves Slowly, with organization and policy changes	Evolves quickly, adapting to project needs

- Test and Analysis Strategy
    - Lessons of past experience
        - An organizational asset built and refined over time
    - Body of explicit knowledge
        - More valuable than islands of individual competence
        - Amenable (open/responsive) to improvement
        - Reduces vulnerability to organizational change (loss of key employees)
    - Essential for
        - Avoiding recurring errors
        - Maintaining consistency of the process
        - Increasing development efficiency
- Considerations in fitting a strategy to an organization
    - Structure and size
        - Distinct quality groups in large organizations, overlapping roles in smaller organizations

        - Greater reliance on documents in large organizations rather than small ones
    - Overall process
        - Cleanroom requires statistical testing and forbids unit testing
            - Fits with tight, formal specs and emphasis on reliability
        - XP prescribes “test first” and pair programming
            - Fits with fluid specifications and rapid evolution
    - Application domain
        - Safety critical domains may impose particular quality objectives and require documentation for certification
- Elements of a strategy
    - Common quality requirements that apply to all or most products. Unambiguous definitions and measures.
    - Set of documents about contents and relationships normally produced during the quality process.
    - Activities prescribed by the overall process. Standard tools and practices.
    - Guidelines for project staffing and assignment of roles and responsibilities.
- Test and Analysis Plan
    - What quality activities will be carried out?
    - What are the dependencies among the quality activities and between quality and other development activities?
    - What resources are needed and how will they be allocated?
    - How will both the process and the product be monitored?
- Main elements of a plan
    - Items and features to be verified (scope and target)
    - Activities and resources. Constraints imposed by resources on activities
    - Approaches to be followed. Methods and tools
    - Criteria for evaluation results
- Quality Goals
    - Expressed as properties satisfied by the product
        - Include metrics to be monitored during the project
        - Not all details are available in the early stages of development
    - Initial plan
        - Based on incomplete information
        - Incrementally refined
- Task Schedule
    - Initially based on quality strategy and past experience
    - Breaks large tasks into subtasks, refining as the process advances
    - Includes dependencies among quality activities, and between quality and development activities
    - Guidelines and objectives
        - Schedule activities for steady effort and continuous progress and evaluation without delaying development activities
        - Schedule activities as early as possible

        - Increase process visibility (how do we know we're on track)
- Schedule risk
    - Critical path - a chain of activities that must be completed in sequence and that have maximum overall duration
        - Schedule critical tasks and tasks that depend on critical tasks as early as possible to provide schedule slack and prevent delay in starting critical tasks
    - Critical dependence - a task on a critical path schedule immediately after some other task on the critical path
        - May occur with tasks outside the quality plan
        - Reduce critical dependencies by decomposing tasks on critical path, factoring out subtasks that can be performed earlier
- Risk Planning
    - Risks cannot be eliminated by they can be assessed, controlled, and monitored
    - Generic risk
        - Personnel
            - Risks
                - Loss of staff members
                - Under qualified staff members
            - Control strategies
                - Cross training
                - Continuous education
                - Identification of skill gaps early in the project
                - Promotions and friendly competition
                - Include training time in the project schedule
        - Technology
            - Risks
                - High fault rate due to unfamiliar components
                - Automation tools do not meet expectations
            - Strategies
                - Schedule extra time for potential issues
                - Train with new tools
                - Monitor and document common errors
                - Introduce new tools in lower risk projects first
        - Schedule
            - Risks
                - Inadequate unit testing leads to delays later on
                - Difficult to schedule meetings and inspections
            - Strategies
                - Track and reward quality unit testing
                - Schedule time where inspections take precedence
                - Try distributed, less face to face inspection meetings
    - Quality risk
        - Development

        - Risks
            - Poor quality software delivered to the testing group
            - Inadequate unit testing and analysis before committing to the code base
        - Strategies
            - Provide early warning and feedback
            - Schedule inspection of design and code
            - Reward system for development and inspection
            - Increase training through inspection
            - Require coverage or other criteria at unit test level
    - Test Execution
        - Risks
            - Execution costs higher than planned
            - Scarce resources available for testing
        - Strategies
            - Minimize parts of that require full system execution
            - Inspect architecture to improve testability
            - Increase intermediate feedback
            - Invest scaffolding
    - Requirements
        - Risks
            - High assurance critical requirements increase expense and uncertainty
        - Strategies
            - Compare planned testing effort with former projects
            - Balance test and analysis
            - Isolate critical parts
- Contingency Plan
    - Part of the initial plan
    - What could go wrong? How will we know? How will we recover?
    - Evolves with the plan
    - Derives from risk analysis
    - Essential to consider risks explicitly and in detail
    - Defines actions in response to bad news (Plan B)
- Orthogonal Defect Classification (ODC)
    - Categorize defects into classes that collectively point to the part of the system which need attention
    - Accurate classification schema
    - Used for very large projects
    - To distill an unmanageable amount of detailed information
    - Two main steps: fault classification and analysis
- ODC Fault classification
    - When faults are detected
        - Activity executed when the fault is revealed

        - Trigger that exposed the fault
        - Impact of the fault on the customer
    - When faults are fixed
        - Target: entity fixed to remove the fault
        - Type: type of the fault
        - Source: origin of the faulty modules (in house, library, imported, outsourced)
        - Age of the fault element (new, old, rewritten, refactored, refixed code)
- ODC activities and triggers
    - Review and code inspections
        - Design conformance
        - Logic and flow
        - Backward compatibility
        - Internal document
        - Lateral compatibility
        - Concurrency
        - Language dependency
        - Side effects
        - Rare situation
    - Structural test (white box testing)
        - Simple path
        - Complex path
    - Functional test (black box testing)
        - Converge
        - Variation
        - Sequencing
        - Interaction
    - System test
        - Workload and stress
        - Recovery and exception
        - Startup and restart
        - Hardware configuration
        - Software configuration
        - Blocked test
- ODC impact - What did the defect impact? How did it affect the behavior of the system? Use this as a checklist for trying to devise a testing strategy.
    - Installability / Migration
    - Integrity / Security
    - Performance / Reliability / Usability (have you done load testing?)
    - Maintenance / Serviceability
    - Documentation / Accessibility
    - Standards / Capability / Requirements
- Improve the process

    - Resource allocation faults resulting from shallow architectural design can be reduced by introducing specific inspection tasks
    - Faults attributable to inexperience with the development environment can be reduced with focused training
    - Analysis of the fault history can help software engineers build a feedback mechanism to track relevant faults to their root causes
- Root cause analysis (RCA)
    - Technique for identifying and eliminating process faults (troubleshooting)
    - First developed in the nuclear power industry; used in many fields
    - Four main steps
        - What are the faults? A clear, concise description. What is the affect and who is affected?
        - When did the faults occur? When, and when were they found? What is the system state before, during and after it occurred?
        - Why did the faults occur? Distinguish other causal factors from root cause.
        - How could the faults have been prevented?
- What are the faults?
    - Identify a class of import faults
    - Faults are categorized by severity (impact of the fault on the project) and kind
    - Severity : critical, severe, moderate, cosmetic
- Pareto Distribution (80/20)
    - Rule: in many populations, 20% are vital and 80% are trivial
    - Fault analysis
        - 20% of the code is responsible for 80% of the faults
- Stack overflow problem
    - You have to sift through the 80% of people that all have the same issue and are posting about it to get to the kernel of actually important information
    - However, there might be issues that occur infrequently that are still really important (20%)
    - Problem frequency is not the same as problem severity
- Summary of planning and monitoring
    - Planning is necessary to
        - Order, provision, and coordinate quality activities
        - Coordinate quality process with overall development
        - Includes allocation of roles and responsibilities
        - Provide unambiguous milestones for judging progress
    - Process visibility is key
        - Ability to monitor quality and schedule at each step
        - Intermediate verification steps because cost grows with time between error and repair
        - Monitor risks explicitly with contingency plan ready
    - Monitoring feeds process improvement of a single project and across projects

Analysis and Test

- Why produce quality documentation
    - Monitor and assess the process
        - For internal use (process visibility)
        - For external authorities (certification, auditing)
    - Improve the process
        - Maintain a body of knowledge reused across projects
        - Summarize and present data for process improvement
    - Increase reusability of test suites and other artifacts within and across projects
- Major categories of documents
    - Planning documents
        - Describe the organization of the quality process
        - Include organization strategies and project plans
    - Specification documents
        - Describe test suites and test cases
        - Test design specifications, test case specification, checklists, analysis procedure specifications
    - Reporting documents
        - Details and summary of analysis and test results
- Metadata - approval, history, table of contents, summary, goals, references, glossary
- Naming conventions
    - Help people identify documents quickly
    - Standard document names include keywords indicating
        - General scope of the document (project and part)
        - Kind of document (test plan, contingency plan etc)
        - Specific document identify
        - Version
- Strategy document
    - Describes quality guidelines for sets of projects
    - Usually for an entire company or organization
    - Varies among organizations
    - May depend on business conditions
        - Safety critical software producer may need to satisfy minimum dependability requirements defined by a certification authority
        - Embedded software department may need to ensure portability across product lines
    - Sets out requirements on other quality documents
- Plan document
    - Standardized structure
    - Overall quality plan comprises several individual plans
    - May refer to the whole system or part of it
    - May not address all aspects of quality activities
        - Should indicate features to be verified and excluded
        - Indication of excluded features is important

- Test Design specification document
    - Describe complete test suites
    - May be divided into
        - Unit, integration, system, acceptance (granularity)
        - functional, structural, performance (objective)
    - Include all information needed for initial selection of test cases, and maintenance of the test suite over time.
    - Identify features to be verified
    - Include description of testing procedure and pass/fail criteria
    - Include a logical list of test cases
- Test case specification document
    - Complete test design for individual test cases
    - Define inputs, required environmental conditions, procedures for test execution, expected outputs
    - Indicate the item to be tested
    - Describe dependence on execution of other test cases
    - Label with a unique identifier
- Test and analysis report document
    - Report on results
    - For developers to identify open faults and schedule fixes and revisions
    - For test designers to assess and refine their approach
    - Prioritize a list of open faults
    - Must be consolidates and categorized to manage repair effort systematically
    - Prioritized to properly allocate effort and handle all faults
    - Include summary tables with
        - Executed test suites
        - Number of failures
        - Breakdown of failures (repeated, new, improved)