OBJECT ORIENTED SOFTWARE ENGINEERING

UNIT IV

Learning Objectives

- **Object Oriented Testing Techniques:**
  →Testing Terminology,
  →Types of test,
  → Automatic Tests,
  →Testing Strategies.
- **Agile Process**:
  →Agile Manifesto,
  → Agile Principles,
  →Introduction to Extreme Programming,
  →Scrum,
  → Lean processes.
- Case Studies.

TESTING

Object Oriented Testing Concepts

- Testing
- Verification
- Validation
- Debugging
- Certification
- Clean Room Software Engineering
- Error
- Fault
- Failure
- Testing Level
  - Unit testing
  - Integration Testing
  - System Testing

•Testing Techniques
  •Regression Test
•Testing Focuses
  •Operation test
  •Full-scale test
  •Stress test
  •Overload test
  •Negative test
  •Test based on requirements
  •Ergonomic tests
  •Testing of the user documentation
  •Acceptance testing

Object Oriented Testing Concepts.

- Stubs
- Drivers
- Test bed
- Equivalence set
- Equivalence partitioning
- Automatic Testing
  - Test data
- Test program

Software Testing

- Consumes at least half of the labor
- Process of testing software product
- Contribute to
  - Delivery of higher quality product
  - More satisfied users
  - Lower maintenance cost
  - More accurate and reliable results

Error

- People make errors.
- Typographical error
- Misreading of a specification
- Misunderstanding of functionality of a modı
- A good Synonym is Mistake.
- When people make mistakes while coding these mistakes **"bugs"**

Fault/Defect

Representation of an error

- DFD
- Hierarchy chart
- Source Code
- An error may lead to one or more faults

Fault of Omissions

- If certain specifications have not been programm

Fault of Commission

- If certain program behavior have not been specifi

Fault/Defect

- Defects generally fall into one of the followir categories
  - Wrong
  - Missing
  - Extra

Failure/Incident

Failure

- A particular fault may cause different failures, depending been exercised

Incident

- When a failure occurs, it may or may not readily apparent to t
- Incident is the symptom associated with a failure that alerts occurrence of a failure

Software Testing

- Software testing can be stated as the process of **validating** and **verifying** that a software program/application/product:
- Meets the **requirements** that guided its design and development works as expected;
- Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the **test effort** occurs after the requirements have been defined and the coding process has been completed.

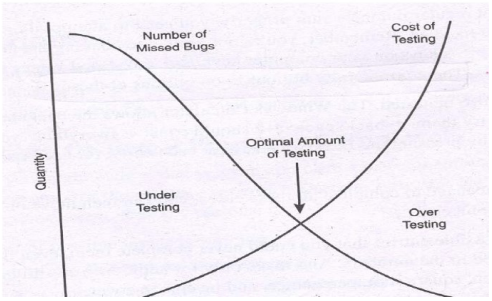
Software Testing

- A primary purpose of testing is to ***detect software failures*** so that defects may be discovered and corrected.
- Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions

Testing takes creativity

- Testing often viewed as dirty work.
- To develop an effective test, one must have:
  - ✓ Detailed understanding of the system
  - ✓ Knowledge of the testing techniques
  - ✓ Skill to apply these techniques in an effective and efficient manner
- Testing is done best by **independent testers**
  - We often develop a certain mental attitude that the program should in a certain way when in fact it does not.

Testing Costs



Testing Objectives

- Executing a program with the goal of finding an *error*.
- To check if the system meets the requirements and be executed successfully in the planned environment.
- To check if the system is " Fit for purpose".
- To check if the system does what it is expected to do.

Tester Objectives

- Find bugs as **early** as possible and make sure they get fixed.
- To **understand** the application well.
- Study the functionality in **detail** to find where the bugs are likely to occur.
- Study the code to ensure that each and every line of code is tested.
- Create test cases in such a way that testing is done to uncover the **hidden bugs** and also ensure that the software is usable and reliable

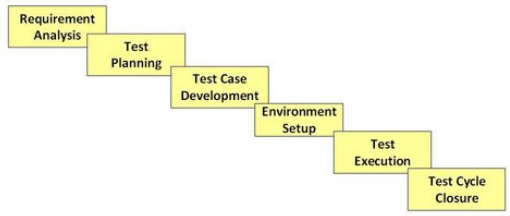
Verification and Validation

Verification - typically involves reviews and meeting to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, and inspection meeting.

Validation - typically involves actual testing and takes place after verifications are completed.

In other words, validation is concerned with checking that the system will meet the customer's actual needs, while verification is concerned with whether the system is well-engineered, error-free, and so on. Verification will help to determine whether the software is of high quality, but it will not ensure that the system is useful.

Software Testing Life Cycle



Requirement Analysis

Activities

- Identify types of tests to be performed.
- Gather details about testing priorities and focus.
- Prepare Requirement Traceability Matrix (RTM).
- Identify test environment details where testing is supposed to be carried out.
- Automation feasibility analysis (if required).

Deliverables

- RTM
- Automation feasibility report. (if applicable)

Test Planning

Activities

- Preparation of test plan/strategy document for various types of testing
- Test tool selection
- Test effort estimation
- Resource planning and determining roles and responsibilities.
- Training requirement

Deliverables

- Test plan /strategy document.
- Effort estimation document.

Test Case Development

Activities

- Create test cases, automation scripts (if applicable)
- Review and baseline test cases and scripts
- Create test data (If Test Environment is available)

Deliverables

- Test cases/scripts
- Test data

Test Environment Setup

Activities

- Understand the required architecture, environment set-up and prepare hardware and software requirement list for the Test Environment.
- Setup test Environment and test data
- Perform smoke test on the build

Deliverables

- Environment ready with test data set up
- Smoke Test Results.

Test Execution

Activities

- Execute tests as per plan
- Document test results, and log defects for failed cases
- Map defects to test cases in RTM
- Retest the defect fixes
- Track the defects to closure

Deliverables

- Completed RTM with execution status
- Test cases updated with results
- Defect reports

Test Cycle Closure

Activities

- Evaluate cycle completion criteria based on Time. Test coverage, Cost. Software, Critical Business Objectives , Quality
- Prepare test metrics based on the above parameters.
- Document the learning out of the project
- Prepare Test closure report
- Qualitative and quantitative reporting of quality of the work product to the customer.
- Test result analysis to find out the defect distribution by type and severity.

Deliverables

- Test Closure report
- Test metrics

Testing Levels

- Unit testing
- Integration testing
- System testing
- Acceptance testing

Types of Testing

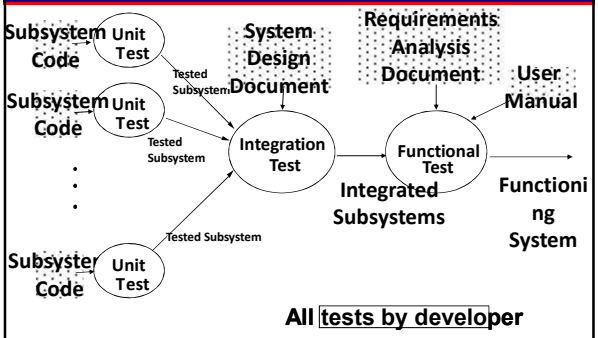
- Unit Testing:
  - Individual subsystem
  - Carried out by developers
  - Goal: Confirm that subsystems is correctly coded and carries out the intended functionality
- Integration Testing:
  - Groups of subsystems (collection of classes) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interface among the subsystem

System Testing

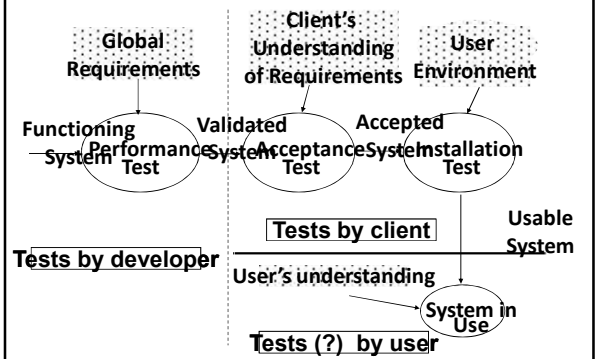
- System Testing:
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and global)
- Acceptance Testing:
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets customer requirements and is ready to use

Implementation (Coding) and testing go hand in hand

Testing Activities



Testing Activities



Unit testing

- The most 'micro' scale of testing.
- Tests done on particular functions or code modules.
- Requires knowledge of the internal program design and code.
- Done by Programmers (not by testers).

Unit testing

Objectives	• To test the function of a program or unit of code such as a program or module • To test internal logic • To verify internal design • To test path & conditions coverage • To test exception conditions & error handling
When	• After modules are coded
Input	• Internal Application Design • Unit Test Plan
Output	• Unit Test Report

Unit testing

Who	•Developer
Methods	•White Box testing techniques
Tools	•Debug •Re-structure •Code Analyzers •Path/statement coverage tools
Education	•Testing Methodology •Effective use of tools

Incremental Integration Testing

- Continuous testing of an application as and when a new functionality is added.
- Application's functionality aspects are required to be independent enough to work separately before completion of development.
- Done by programmers or testers.

Integration Testing

Integration Testing

- Testing of **combined parts** of an application to determine their **functional correctness**.
- 'Parts' can be
  - ✓ code modules
  - ✓ individual applications
  - ✓ client/server applications on a network.

Integration Testing

Types of Integration Testing

- Big Bang testing
- Top Down Integration testing
- Bottom Up Integration testing

Integration Testing

Objectives	• To technically verify proper interfacing between modules, and within sub-systems
When	• After modules are unit tested
Input	• Internal & External Application Design • Integration Test Plan
Output	• Integration Test report

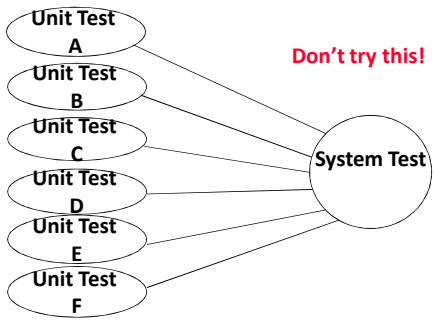
Integration Testing

Who	•Developers
Methods	•White and Black Box techniques •Problem / Configuration Management
Tools	•Debug •Re-structure •Code Analyzers
Education	•Testing Methodology •Effective use of tools

Integration Testing Strategy

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - Big bang integration (Non incremental)
  - Bottom up integration
  - Top down integration
  - Sandwich testing
  - Variations of the above
- For the selection use the system decomposition from the System Design

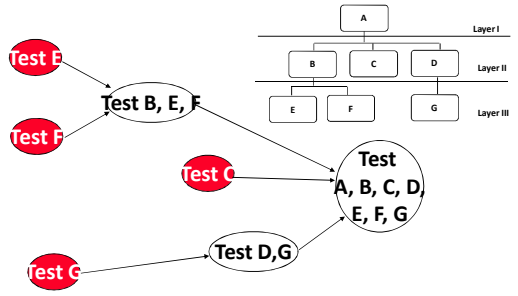
Integration Testing: Big-Bang Approac



Bottom-up Testing Strategy

- The subsystem in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing

Bottom-up Integration



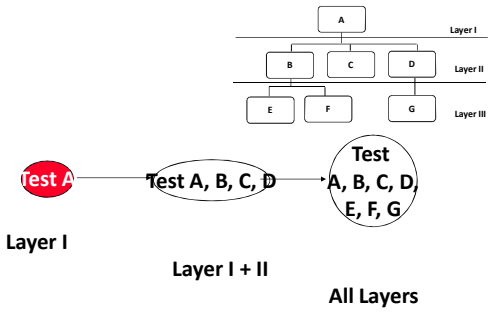
Pros and Cons of bottom up integration testin

- Bad for functionally decomposed systems:
  - Tests the most important subsystem (UI) last
- Useful for integrating the following systems
  - Object-oriented systems
  - real-time systems
  - systems with strict performance requirements

Top-down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the **tested subsystems** and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test

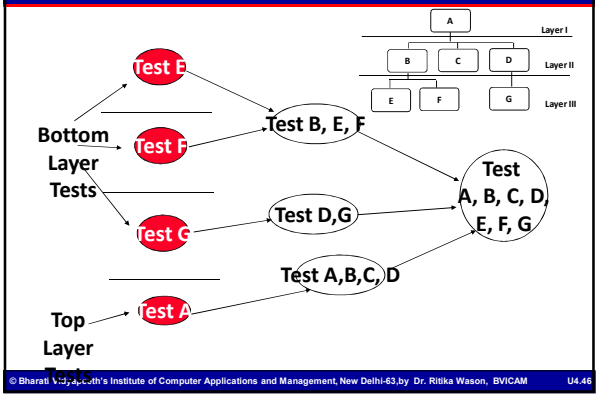
Top-down Integration Testing



Sandwich Testing Strategy

- *Combines top-down strategy with bottom-up strategy*
- *The system is view as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer
- How do you select the target layer if there are more than 3 layers?
  - Heuristic: Try to minimize the number of stubs and drivers

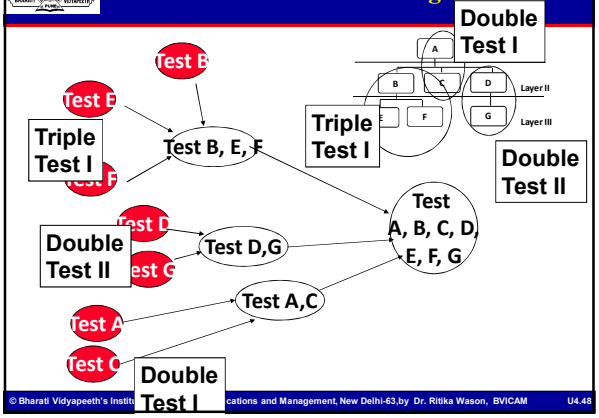
Sandwich Testing Strategy



Pros and Cons of Sandwich Testing

- Top and Bottom Layer Tests can be done in parallel
- Does not test the individual subsystems thoroughly before integration
- Solution: Modified sandwich testing strategy
- **Test in parallel:**
  - Middle layer with drivers and stubs
  - Top layer with stubs
  - Bottom layer with drivers
- **Test in parallel:**
  - Top layer accessing middle layer (top layer replaces drivers)
  - Bottom accessed by middle layer (bottom layer replaces

Modified Sandwich Testing Strategy



System Testing

- Functional Testing
- Structure Testing
- Performance Testing
- Acceptance Testing
- Installation Testing

Impact of requirements on system testing:

- The more explicit the requirements, the easier they are to test.
- Quality of use cases determines the ease of functional testing
- Quality of subsystem decomposition determines the ease of structure testing
- Quality of nonfunctional requirements and constraints determines the ease of performance tests:

Structure Testing

- *Essentially the same as white box testing.*
- Goal: Cover all paths in the system design
  - Exercise all input and output parameters of each component.
  - Exercise all components and all calls (each component is called at least once and every component is called by all possible callers.)
  - Use conditional and iteration testing as in unit testing.

Functional Testing

Essentially the same as black box testing

- Goal: Test functionality of system
- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box.
- Unit test cases can be reused, but in end user oriented new test cases have to be developed as well.

Performance Testing

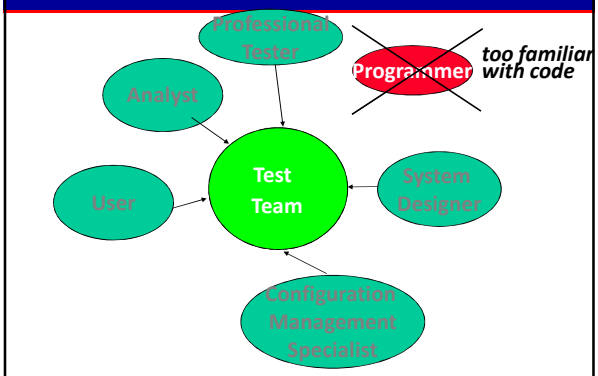
- Stress Testing
  - Stress limits of system (maximum # of users, peak demands, extended operation)
- Volume testing
  - Test what happens if large amounts of data are handled
- Configuration testing
  - Test the various software and hardware configurations
- Compatibility test
  - Test backward compatibility with existing systems
- Security testing
  - Try to violate security requirements
- Timing testing
  - Evaluate response times and time to perform a function
- Environmental test
  - Test tolerances for heat, humidity, motion, portability
- Quality testing
  - Test reliability, maintain- ability & availability of the system
- Recovery testing
  - Tests system's response to presence of errors or loss of data.
- Human factors testing
  - Tests user interface with user

Acceptance Testing

- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client/sponsor
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer.
- Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:

- *Alpha test:*
  - Sponsor uses the software at the *developer's site.*
  - Software used in a controlled setting, with the developer always ready to fix bugs.
- *Beta test:*
  - Conducted at *sponsor's site* (developer is not present)
  - Software gets a realistic workout in target environ-ment
  - Potential customer might get discouraged

Test Team



System Testing

Objectives	• To verify that the system components perform control functions • To perform inter-system test • To demonstrate that the system performs both functionally and operationally as specified • To perform appropriate types of tests relating to Transaction Flow, Installation, Reliability etc.
When	• After Integration Testing
Input	• Detailed Requirements & External Application Design • Master Test Plan • System Test Plan
Output	• System Test Report

System Testing

Who	•Development Team and Users
Methods	•Problem / Configuration Management
Tools	•Recommended set of tools
Education	•Testing Methodology •Effective use of tools

TESTING METHODOLOGIES AND TYPES

Testing Methodologies

Black box testing

- No knowledge of internal design or code required.
- Tests are based on requirements and functionality

White box testing

- Knowledge of the internal program design and code required.
- Tests are based on coverage of code statements, branches, paths, conditions.

Testing Methodologies

Black box / Functional testing

Based on requirements and functionality

Not based on any knowledge of internal design or code

Covers all combined parts of a system

Tests are data driven

Black-box Testing

- Focus: I/O behavior. If for any given input, we can predict the output, then the module passes the test.
  - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class. (Example: If an object is supposed to accept a negative number, testing one negative number is enough)

White box testing / Structural testing

Based on knowledge of internal logic of an application's code

Based on coverage of code statements, branches, paths, conditions

Tests are logic driven

White-box Testing

- Statement Testing (Algebraic Testing): Test single statements (Choice of operators in polynomials, etc)
- Loop Testing:
  - Cause execution of the loop to be skipped completely. (Exception: Repeat loops)
  - Loop to be executed exactly once
  - Loop to be executed more than once
- Path testing:
  - Make sure all paths in the program are executed
  - Branch Testing (Conditional Testing): Make sure that each possible outcome from a condition is tested at least once

White Box - Testing Technique

- All independent paths within a module have been exercised at least once
- Exercise all logical decisions on their *true* and sides
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to ensure their validity

Loop Testing

This white box technique focuses on the validity of loop constructs.

Different classes of loops can be defined

- simple loops
- nested loops

Other White Box Techniques

Statement Coverage – execute all statements at least once

Decision Coverage – execute each decision direction at least once

Condition Coverage – execute each decision with all possible outcomes at least once

Decision / Condition coverage – execute all possible combinations of condition outcomes in each decision.

Multiple condition Coverage – Invokes each point of entry at least once.

Comparison White & Black-box Testing

- White-box Testing:
  - Potentially infinite number of paths have to be tested
  - White-box testing often tests what is done, instead of what should be done
  - Cannot detect missing use cases
- Black-box Testing:
  - Potential combinatorial explosion of test cases (valid & invalid data)
  - Often not clear whether the selected test cases uncover a particular error
  - Does not discover extraneous use cases ("features")
- Both types of testing are needed
- White-box testing and black box testing are the extreme ends of a testing continuum.
- Any choice of test case lies in between and depends on the following:
  - Number of possible logical paths
  - Nature of input data
  - Amount of computation
  - Complexity of algorithms and data structures

Testing Techniques

Functional testing

- Black box type testing geared to functional requirements of an application.
- Done by testers.

System testing

- Black box type testing that is based on overall requirements specifications; covering all combined parts of the system.

End-to-end testing

- Similar to system testing; involves testing of a complete application environment in a situation that copies real-world use.

Testing Techniques

Regression testing

- Re-testing after fixes or modifications of the software or its environment.

Acceptance testing

- Final testing based on specifications of the end-user or customer

Load testing

- Testing an application under heavy loads.
- Eg. Testing of a web site under a range of loads to determine, when the system response time degraded or fails.

Testing Techniques

Stress Testing

- Testing under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database etc.
- Term often used interchangeably with 'load' and 'performance' testing.

Performance testing

- Testing how well an application complies to performance requirements.

Testing Techniques

Install/uninstall testing

- Testing of full, partial or upgrade install/uninstall process.

Recovery testing

- Testing how well a system recovers from crashes, HW failures or other problems.

Compatibility testing

- Testing how well software performs in a particular HW/SW/OS/NW environment.

Testing Techniques

Exploratory testing / ad-hoc testing

- Informal SW test **that is not based on formal test plans** or test cases; testers will be learning the SW in totality as they test it.

Comparison testing

- Comparing SW **strengths and weakness to competing products**.

Test Plan

Objectives

- To create a set of testing tasks.
- Assign resources to each testing task.
- Estimate completion time for each testing task.
- Document testing standards.

Test Cases

Test case is defined as

- A set of **test inputs**, **execution conditions** and expected results, developed for a particular objective.
- Documentation specifying inputs, predicted results and a set of execution conditions for a test item.

Test Cases

Contents

- Test plan reference id
- Test case
- Test condition
- Expected behavior

Good Test Cases

Find Defects

Have high probability of finding a new defect.

Unambiguous tangible result that can be inspected.

visible to requirements or design documents

Execution and tracking can be automated

Do not mislead

Feasible

Defect Log

- Defect ID number
- Descriptive defect name and type
- Source of defect – test case or other source
- Defect strictness
- Defect Priority
- Defect status (e.g. New, open, fixed, closed, reopen, reject)

Defect Log

7. Date and time tracking for either the most recent status change, or for each change in the status.
8. Detailed description, including the steps necessary to reproduce the defect.
9. Component or program where defect was found
10. Screen prints, logs, etc. that will aid the developer in resolution process.
11. Stage of origination.
12. Person assigned to research and/or corrects the defect.

Test Metrics

User Participation = User Participation test time **Vs.** Total test time.

Path Tested = Number of path tested **Vs.** Total number of paths.

Acceptance criteria tested = Acceptance criteria verified **Vs.** Total acceptance criteria.

Test Metrics

Test cost = Test cost **Vs.** Total system cost.

Cost to locate defect = Test cost / No. of defects located in the testing.

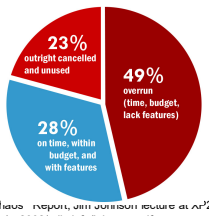
Detected production defect = No. of defects detected in production / Application system size.

Test Automation = Cost of manual test effort / Total test cost.

Extreme Programming and Scrum - Getting Started with Agile Software Development

The Problem: The Chaos Report

- Started in 1994, studied over 35,000 application development projects
- In 2000:



Source: Standish "Chaos" Report, Jim Johnson lecture at XP2002 conference, http://www.xp2003.org/xp2002/talksinfo/johnson.pdf

What is Needed: DuPont Study

- 25% of features are needed
- 75% of features are "nice to have"

Source: Jim Johnson lecture at XP2003 conference, http://www.xp2003.org/xp2002/talkinfo/johnson.pdf

Lifecycle Costs

- Up to 80% of software lifecycle cost, the total cost of ownership (TCO), is in *maintenance,* not first development
- Focusing on "abilities" is critical to ROI:
  - maintainability, extensibility, adaptability, scalability, and most importantly *understandability* (usability, readability, testability)

Agile Methods

- **Extreme Programming** (XP) (Kent Beck, Ward Cunningham, Ron Jeffries)
- **Scrum** (Jeff Sutherland, Mike Beedle, Ken Schwaber)
- **DSDM** – Dynamic Systems Development Method (Community owned)
- **Crystal** (Alistair Cockburn)
- **ASD** – Adaptive Software Development (Jim Highsmith)
- **XBreed** (Mike Beedle)

All Agile Methods

- Maximize value by minimizing anything that does not directly contribute to product development and delivery of customer value
- Respond to change by inspecting and adapting
- Stress evolutionary, incremental development
- Build on success, not hope

We've Seen It Before

- Lean Manufacturing (1990, Toyota)
- Agile Manufacturing
- Just-in-time JIT
- Common goals include:
  - Reduce Cycle Time
  - Maximize Quality
  - Reduce Costs

Lean

- Lean means ***prioritize and optimize everything to deliver value to the customer***
- One common technique: ***Postpone decisions until the last responsible moment***. Live with uncertainty but define, communicate, and manage it
  Lean Manufacturing and the Toyota Production System, SAE International
- Lean Software Development: An Agile Toolkit, Mary & Tom Poppendieck, 2003

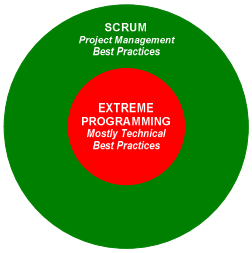
Scrum

- Term in rugby to get an out-of-play ball back into play
- Term used in Japan in 1987 to describe hyper-productive development
- Used by Ken Schwaber and Mike Beedle to describe their Agile methodology

Extreme Programming

- A collection of best practices – each done to the "extreme"
- Sounds extreme, but very disciplined
- Created by Kent Beck, Ward Cunningham, Ron Jeffries

Scrum with Extreme Programming



Scrum works well as a *wrapper* around Extreme Programming

Agile Independence

- Not created by any single company, but by a group of software industry experts to find "better ways of developing software by doing it and helping others do it."*
- Agile Principles:
  - highest priority is customer satisfaction
  - welcomes changing requirements
  - frequently deliver working software
  - advocates close collaboration and rapid feedback
  - reinforces "inspect and adapt"

* www.agilealliance.org

Key PM Difference

Defined/ Predictive Project Management
vs.
Empirical Project Management

Defined PM

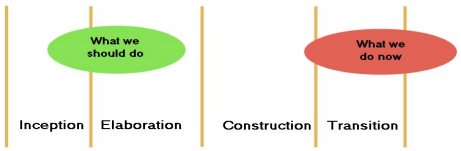
- ***Assumes we can predict*** how the project will unfold – ***assumes very little uncertainty***
- Time to complete and costs predictable
- Uses work breakdown structure
- Manages to a static plan
- Primary participants: development team
- Success; ***On Time & On Budget***

Empirical PM

- Software and systems construction ***is a discovery process*** – ***manages uncertainty***
- Focuses on value/cost tradeoffs
- Plan is volatile; use discoveries to reprioritize and adjust
- Primary participants: project steering team
- Success; ***Delivering good value in reasonable time***

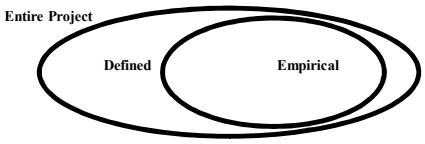
PM End-Game

- Almost all projects eventually revert to empirical PM



Both Are Needed

- Defined PM works because many things in a project *are* deterministic.
- Defined model provides constraints:
  - "deliver not the best solution, but the best we can afford"



Empirical PM Strategy

- Early estimates of cost and value, tied to business processes
- Deliver subsets of functionality prioritized by business value
- Reassess and re-plan to fit resources, schedule, and discoveries

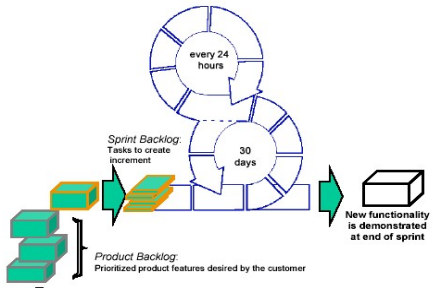
Agile PM Concepts

- Software construction is a discovery process
- Not the best solution; the affordable solution
- Invent successful outcomes

Scrum Overview

- Empirical management and control process for projects and products
- Widely used since 1990's
- Wraps existing engineering practices
- Manages noise, allows overhead to wither
- Simple, common sense
- Delivers business functionality in 30 days
- Scalable

Scrum Scheduling and Tracking



Scrum Roles

- Product Owner
- Team
- Scrum Master

Scrum Roles – Product Owner

- Single person who owns, maintains, prioritizes Product Backlog
- Empowered to make decisions for customers and users
- Responsible for vision, ROI, and releases of product
- Attends Sprint planning and Sprint review meetings

Scrum Roles - Team

- Self-organizing, cross-functional, no formal roles
- Seven plus or minus two people
- Best experts available
- Cost and commit to work, and responsible for delivering
- Full autonomy and authority to deliver during Sprint

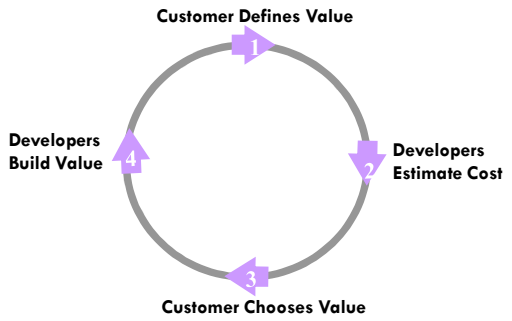
Scrum Roles – Scrum Master

- Project manager, Coach, and/or Player-Coach
- Responsible for process and maximizing team productivity
- Sets up and conducts meetings
  - Sprint Planning
  - Daily "Scrum"
  - Sprint Release

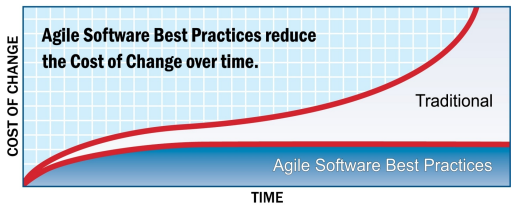
XP Definitions

- Kent Beck's idea of turning the knobs on all the best practices up to 10.
- Optimizing the "Circle of Life" by hitting the sweet-spot of practices that self-reinforce and become more than the sum of the parts (synergize).

XP Circle of Life



Cost-of-Change Curves



Flattening cost of change curve is both *enabled by* and *exploited by* Extreme Programming (XP)

The Four XP Values

- **Simplicity**
  - Simplest thing that could possibly work
  - YAGNI: you aren't going to need it
- **Communication**
  - Developers
  - Users
  - Customers
  - Testers
  - Code
- **Feedback**
  - Testing
  - Experimenting
  - Delivering
- **Courage**
  - Trust
  - History

The Four XP Variables

- **Quality**
  - **Internal high, fixed**
- **Cost**
  - **People-Time**
  - **Mythical Man-Month (F. Brooks)**
- **Schedule**
  - **Fixed-length, short iterations**
- **Scope**
  - **Negotiable**

Twelve XP Practices

1. **Planning Game**
2. **Short Releases**
3. **Simple Design**
4. **Testing**
5. **Refactoring**
6. **Pair Programming**
7. **Collective Ownership**
8. **Continuous Integration**
9. **On-site Customer**
10. **Sustainable Pace**
11. **Metaphor**
12. **Coding Standards**

1. Planning Game

- Release Planning: Define and estimate higher-level features down to about 5-10 days effort each. Customer lays features in fixed-length iteration schedule.
- Iteration Planning: Same, but to 3 or less days effort & detailed story cards within next iteration.
- Simple to steer project towards success.

2. Short Releases

- Deliver business value early and often
- Do not slip iteration release dates
  - adjust scope within an iteration, never time or quality
- Small, stable teams are predictable in short time-frames

3. Simple Design

- XP Mantra: "The simplest thing that could *possibly* work".
- Meet current, minimum business requirements only. Avoid anticipatory design.
- YAGNI – You Aren't Going to Need It

4. Testing

- Automated unit tests for every entity.
- Automated acceptance tests for every story / requirement.
- All unit tests pass 100% before checking in a feature.
- Test-First, in small increments:
  1. Write the test
  2. Prove it fails (red-bar)
  3. Code until it passes (green-bar)

5. Refactoring

- Refactoring: changing internal structure without changing external behavior
- Remove duplication. “Once and Only Once”, “Three strikes and your out”.
- Leaves code in simplest form.
- When change is hard, refactor to allow change to be easy, testing as you go, then add change.

6. Pair Programming

- Two heads are better than one, especially in an open lab environment (colocation)
- Earliest possible code inspections
- Earliest possible brainstorming
- Better quality at lower cost
- Driver/Navigator
- Peer pressure reinforces discipline

7. Collective Ownership

- Interchangeable programmers
- Team can go at full speed
- Can change anything, anytime, without delay

8. Continuous Integration

- Avoids "versionitis" by keeping all the programmers on the same page
- Integration problems smaller, taken one at a time
- Eliminates traditional, high-risk integration phase

9. On-site Customer

- Customer/User liaisons are team-members
- Available for priorities, clarifications, to answer detailed questions
- Reduces programmer assumptions about business value
- Shows stakeholders what they pay for, and why

10. Sustainable Pace

- Tired programmers make more mistakes
- Better to stay fresh, healthy, positive, and effective
- XP is for the average programmer, for the long run

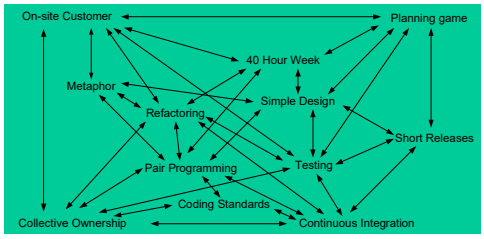
11. Metaphor

- Use a "system of names"
- Use a common system description
- Helps communicate with customers, users, stakeholders, and programmers

12. Coding Standards

- All programmers write the same way
- Rules for how things communicate with each other
- Guidelines for what and how to document

XP Practices Support Each Other



Source: Beck, Extreme Programming Explained: Embrace Change, 1999

Practices to Start With

- Talking to, instead of about, people, in their language, considering their perspective
  - Customer, developer, mgmt., Q/A, user, finance, marketing, sponsor
- Frequent Integration (Config. Mgmt., Check-in > daily)
- Testing (Unit, Integration, System, Feature)
- Release Management (build-box, sandboxes, labeled releases, migrations)

See www.balancedagility.com

How to Explore

Web

- Agile Alliance: www.agilealliance.org
- Scrum: www.controlchaos.com
- Don Well's XP Introduction: *Extreme Programming: A Gentle Introduction* www.extremeprogramming.org

How Not to Get Started

1. Read some
2. Discuss some
3. Start an approach without advice from those with previous experience
4. Draw conclusions from experience

- Can work this way, but its risky
- Often fails to define and leverage success criteria. Often unrealistic expectations.
- Inexperience decreases chances of success

How Best to Get Started

- Get help from experienced people for:
  - Readiness assessments
  - Approach selection
    - ✓Pilot / skunkworks vs. changing existing process
    - ✓Mission-critical vs. stand-alone
    - ✓Selective best practices vs. complementary set vs. all best practices
  - Measurement and success criteria
  - Identifying and delivering targeted training, mentoring, coaching, project management / stewardship

Agile Best Practice Adaptations

- How long should iterations and releases be?
- How does development work with QA?
- How do our stakeholders work with multiple customers?
- How should our teams be structured?
- How do we work with regulatory agencies?
- How does this work with legacy systems?
- How does this work with Use Cases and RUP?
- How do we ensure architectural vision and usage.

Agile Summary

- Agile = try, inspect, adapt, repeat
- Highly focused, empowered teams
- Collaborate with all stakeholders
- Optimize and automate feedback
- Deliver real value early and often
- Use feedback to evaluate, ruthlessly prioritize, and re-plan
- Delivers high quality, ensures flexibility
- Evaluate business value of everything

Agile Future

- Agile in most dev. orgs, in few IT orgs.
- Agile is here to stay, past early adopters, into early majority
- "Agile" is loosing meaning
- XP is developer-focused, now Q/A friendly, needs to become customer/user friendly
- Scrum is still "pure", but there are now tools… CMM and RUP were "pure" to start…
- All camps need to sell business value, in business terms, financial terms, risk terms