



---

COLLEGE OF ENGINEERING  
AND COMPUTER SCIENCE

# Advanced Software Process

Part II: The Repeatable Process

7. *Software Configuration Management (I)*

Dave Garcia-Gomez  
Faculty / Lecturer  
Department of Computer Science

# Course Roadmap

## Part I: Software Process Maturity

- 1. A Software Maturity Framework
- 2. The Principles of Software Process Change
- 3. Software Process Assessment
- 4. The Initial Process

## Part II: The Repeatable Process

- 5. Managing Software Organizations
- 6. The Project Plan
- 7. Software Configuration Management (Part I)
- 8. Software Quality Assurance

## Part III: Defined Process

- 9. Software Standards
- 10. Software Inspections
- 11. Software Testing
- 12. Software Configuration Management (continued)
- 13. Defining the Software Process
- 14. The Software Engineering Process Group

## Part IV: The Managed Process

- 15. Data Gathering and Analysis
- 16. Managing Software Quality

## Part V: The Optimizing Process

- 17. Defect Prevention
- 18. Automating the Software Process
- 19. *Contracting for Software*
- 20. *Conclusion*

# Software Configuration Management (Part I)

- The Need for Configuration Management
- Software Product Nomenclature
- Basic Configuration Management Functions
- Baselines
- Configuration Management Responsibilities
- The Need for Automated Tools

# Software Configuration Management

- Change management is one of the fundamental activities of SE.
- Changes to the requirements drives the design, and design changes affect the code.
- For even modest-size projects, the number of people involved and the change volume generally require a formal change management system.
  - Software configuration management

# Software Configuration Management

- Since a key objective of the software process is to have change activity converge until the final product is stable enough to ship, the management of all changes is important.
- Focus on code control
  - Control of requirements and design changes is also critically important, but these functions can often be handled as enhancements to a code management system.

# Software Configuration Management

Change management, more commonly called software configuration management (SCM), is a set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made [Pressman 2005].

# The Need for Software Configuration Management

- The most frustrating software problems are often caused by **poor configuration management**.
- The problems are frustrating because they take time to fix, they often happen at the worst time, and they are totally unnecessary.
- **Configuration management** helps to reduce these problems **by coordinating the work products of the many different people who work on a common project**.
  - Without such control, their work will often conflict, resulting in problems.

# The Need for Software Configuration Management

## Problems:

- Simultaneous update
  - When two or more programmers work separately on the same program
- Shared code
  - When a bug is fixed in code shared by several programmers
- Common code
  - When common program functions are modified
- Versions
  - Evolutionary release

# The Need for Software Configuration Management

Control system to answer

- What is my current software configuration?
- What is its status?
- How do I control changes to my configuration?
- How do I inform everyone else of my changes?
- What changes have been made to my software?
- Do anyone else's changes affect my software?

# Software Product Nomenclature

- Since the role of **configuration management** is to control the development of the system elements as they are built and then combined into a full system, it is important to use **common system terminology**.
- The **design process** starts by **successively partitioning the system** until a satisfactory level of detail is reached.
  - At this point, implementation begins with these smallest pieces, which are progressively assembled and tested until the total system is completed.

# Software Product Nomenclature

## Figure 7.1

- System
  - The package of all the SW (and HW) that meets the user's requirements.
- Subsystem
  - Large systems can have many subsystems, such as communications, display, and processing.
- Product
  - Subsystem typically contain many products. For example, OS contains a control program, compilers, utilities, and so forth.
- Components
  - A control program could be made of such components as the scheduler and I/O controller.
- Module - Components consist of a number of modules.

# Tests Supporting Implementation Process

- Unit test
  - Test of each individual module
- Integration test
  - As the modules are integrated into components, products, subsystems, and systems, their interfaces and interdependencies are tested.
- Function test
  - When integration results in a functionally operable build, it is tested successfully in component test, product test, subsystem test, and finally in system test.
- Regression test
  - At each integration step (spin), a new build is produced.
  - This is first tested to ensure that it has not regressed.

# Basic Configuration Management Functions

- Once an initial product level has stabilized, a **first baseline** is established.
- With each successive set of enhancement, a new baseline is established in step with development.
- Each **baseline** is retained in a permanent database, **together with all the changes** that produced it.
- Configuration Management Overview [Figure 7.2]

# Basic Configuration Management Functions

- The **baseline** is thus the **official repository** for **the product**, and it contains the **most current version**.
- Only tested code and approved changes are put in the **baseline**, which is fully protected.
  - It is **the official source** that all the programmers use to ensure their work is consistent with that of everyone else.

# Basic Configuration Management Functions

Software Configuration Management (SCM)'s  
key tasks:

- Configuration control
- Change management
- Revisions
- Versions
- Deltas
- Conditional code

# Configuration Control

- The task of configuration control revolves around **one official copy of the code**.
- The simplest way to protect every system revision is to keep **a separate official copy of each revision level**.
- While this can take a lot of storage, the most serious issue concerns **code divergence**.

# Revisions

- Keeping track of revisions is an important task of configuration management.
- In large system, some programs, such as the control program, provide essential function for all the others.
- Once a component's modules have been integrated into a testable unit, it is then tested with the latest control program level.
- When new problems are found, previous tests can be rerun to trace the problem source.

# Revisions

- If an early driver is now reproduced using the latest versions of the **control program** modules, some of these modules will likely be different, so the prior tests will not be exactly reproduced.
- The answer is to **keep track of every change to every module and test case**.
- There is **one latest official version** and every prior version is identified and retained.
- These **obsolete copies** can then be used to assist in tracing problems.

# Derivations

- The ability to determine **what has changed** is one of the most powerful SW testing aids.
- Derivations [Figure 7.3]
- A **derivation record** would show that Test A was run on control program level 116 and the rerun was attempted on level 117.
- Changes X and Y could then be identified readily.

# Derivations

- Information maintained in derivation records:
  - The revision level of each module
  - The revision levels of the tools used to assemble, compile, link, load, and execute the programs and tests
  - The test cases used and their revision levels
  - The test data employed
  - The files used
  - The software and hardware system configuration, including peripherals, features, options, allocations, assignments, and HW change levels
  - The operational procedures
  - If not a stand-alone test, a record of the job streams being executed.

# Versions

- Often, several **different functions** can be implemented by the same module with only **modest coding differences**.
  - Version [Figure 7.4]
- As a project progresses, many **versions** of individual work products will be created.
- The repository must be able to save all of these **versions** to enable **effective management of product releases** and to permit developers to go back to previous **versions** during **testing and debugging**.
  - Revision numbering scheme

# Deltas

- The use of **versions** solves the problem of different functional needs for the same module but introduces **multiple copies of the same code**.
  - The reason is that most of the code in the two memory management modules would be identical.
  - One, however, would have an additional routine to handle memory limits testing and mode switching.
  - Since they are stored as **separate modules**, however, there is no way to ensure that all the changes made to one are incorporated in the other.

# Deltas

- One way to handle this is to use **deltas**.
- This involves **storing the base module together with the changes required to make it into another version**.
- Deltas [Figure 7.5]
- Very useful in some situations, but has some disadvantages
  - Use the delta approach for **temporary variations**

# Conditional Code

- Another way to handle slight variations between modules is to use some form of conditional program construction.
- A source program would contain various different versions constructed depending on different conditions, but none would be included in the final system unless called for at system installation.
- This need for conditional code is often met by using system generation options.

# Conditional Code

- These might include conditionally selecting one of several optional modules depending on the particular condition that the program was used.
- All modules would be in the source library, but only one would be used at any time.
- Conditional Code [Figure 7.6]

# Baselines

- The **baseline** is the foundation for configuration management.
- It provides **the official standard** on which subsequent work is based and to which only authorized changes are made.
- After an **initial baseline** is established and frozen, every subsequent **change** is recorded as a **delta** until the **next baseline** is set.

# Baselines

- It is desirable to establish a **baseline** at an early point in every project.
- Establishing a **baseline too early**, however, will impose **unnecessary** procedures and slow the programmers' work.
- As long as the programmers can work on **individual modules with little interaction**, a code **baseline** is not needed.
- As soon as **integration** begins, however, **formal control** is essential.

# Baselines Scope

- **Baseline items** included for the implementation phase:
  - The current level of each module
  - The current level of each test case
  - The current level of each assembler, compiler, editor, or other tools used
  - The current level of any special test or operational data
  - The current level of all macros, libraries, and files
  - The current level of any installation or operating procedures
  - If the project involves operating system or control program changes, it may also be necessary to retain data on the computing system configuration and its change level.

# Baseline Control

- SCM must ensure appropriate **baseline control** while providing flexible service to the programmers.
- On the other hand, the **baseline** must be protected against unauthorized change; at the same time, the programmers should be able to readily modify and test their code.
- This **controlled flexibility** is accomplished by providing the programmers with **private working copies** of any part of the **baseline**.

# Baseline Control

- They can thus try out new changes, conduct tests, or make trial fixes without disturbing anyone else.
- When they are ready to incorporate their work into a new baseline, however, care must be taken to ensure that the new changes are compatible and that no new code causes regressions.

# Configuration Management Records

- To maintain control over all code changes, **several procedures** are used.
- First, every **change proposal** is documented and authorized before being made.
  - Sample **Change Request** Contents [Table 7.1]
- As the change is made, a record is kept of what was done.
  - Sample **Change Log** Contents [Table 7.2]
- The **results of each test** should be documented.
  - Sample **Test Report** Contents [Table 7.3]

# Configuration Management Records

- Detailed test records are particularly crucial when HW and SW changes are made simultaneously.
- Perhaps the single most important project control document is the problem report.
  - Sample Problem Report Contents [Table 7.4]

# Configuration Management Records

- Another important report is the **expect list**.
  - This is a detailed listing of every function and feature planned for every component in each new baseline.
  - All developers who work on program changes for a new baseline should maintain a **current list of all the functions, features, fixes, and interfaces** to be provided by their program when delivered to integration.
  - This information is made **available to all development groups**, so they know of any changes affecting them.

# Configuration Management Responsibilities

- To implement the necessary controls and procedure, a number of responsibilities need to be established.
- Depending on the size of the system and the number of people involved, they may be handled by a single individual, several people, or an entire organization.
- The basic responsibilities are:
  - Configuration manager
  - Module ownership
  - Change Control Board (CCB)

# Module Ownership

- Maintaining design integrity is a common problem in large systems with periodic enhancements.
- One simple but effective approach is to designate a programmer as owner of each module.
- Since there are often many more modules than programmers, one person generally owns several modules at one time.

# Module Ownership

- The module owner's **responsibilities** are:
  - Know and understand the module design
  - Provide advice to everyone who works on or interfaces with the modules
  - Serve as **technical control point** for all module modifications, including both enhancement and repair
  - Ensure module integrity by **reviewing all changes** and conducting periodic **regression tests**

# The Change Control Board

- On moderate to very large projects, a **central control mechanism** is needed to ensure that every change is properly considered and coordinated.
  - Change Control Board (CCB)
  - Configuration Control Board (CCB)
  - Members from development, documentation, test, assurance, maintenance and release
  - Its purpose is to ensure that every baseline change is properly considered by all concerned parties and that every change is authorized before implementation.

# The Change Control Board

- Information typically needed by a CCB on each proposed change:
  - Size
  - Alternatives
  - Complexity
  - Schedule
  - Impact
  - Cost
  - Severity
  - Relationship to other changes
  - Test
  - Resources
  - System impact
  - Benefits
  - Politics
  - Change maturity

# The Change Control Board

- If the change is to **fix a customer-reported problem**, other information may also be needed:
  - A definition of the problem the change is intended to fix
  - The conditions under which the problem was observed
  - A copy of the trouble report
  - A technical description of the problem
  - The names of any other programs affected

# The Change Control Board

- Additional information should also be available **prior to release:**
  - Final source and/or object code for the change
  - Final documentation changes
  - Evidence of successful inspection and test of the code and documentation
- The CCB not only determines whether the change should be made but also the conditions under which it can be released.

# CCB Problems

- Since the CCB must consider every change, its **workload** can become **excessive**, particularly during the final test and release phases of a large project.
- This is one reason why **multiple CCB's** are established for specific components or test phases.

# CCB Problems

- It is important to select the CCB members with care.
- They have the power to block any part of the project so these assignments should not be treated lightly.
- Generally, in fact, the project manager should personally chair the highest-level CCB.

# The SCM Organization

- A further question concerns the dividing line between configuration management and computer operations.
  - These two organizations must coordinate, control, and track every change so that between them they can reproduce every program or test environment.
  - Project-unique changes generally should be handled by configuration management and system-wide changes by computer operations.

# The Need for Automated Tools

- The final development stages of even modest-sized projects involve a lot of detail.
- Regardless of the care taken in product design, quality can be destroyed by sloppy last-minute changes.
  - Changes as object patches with the intent of later updating the source may cause serious problems if equivalent source changes are not later made and tested for every case.

# The Need for Automated Tools

- If any one change is overlooked or its consequences are not properly considered, unpleasant problems are likely in system test, acceptance test, or customer use.
- **Automated system** are thus generally needed to handle the large amount of detail.

# The Change Management System

- The change management system tracks the change requests, problem reports, CCB actions, and activity log entries.
- Data is recorded on what was done, by whom, when, what remains to be done, any special conditions, and current status.
  - The system should maintain a record of the changes not closed, the changes to a particular component, and the oldest outstanding changes.
  - The system tracks the mean time to change closure and maintains statistics on the change backlog.

# The Change Management System

- The change management system tracks the change requests, problem reports, CCB actions, and activity log entries.
- Data is recorded on what was done, by whom, when, what remains to be done, any special conditions, and current status.
  - The system should maintain a record of the changes not closed, the changes to a particular component, and the oldest outstanding changes.
  - The system tracks the mean time to change closure and maintains statistics on the change backlog.

# The Change Management System

- Change activity is also a powerful indicator of project status.
- With change data, management can pinpoint problems early and take timely action in those areas where the problems are most severe.
- Module Change Tracking [Figure 7.7]

# The System Library

- The system library stores the development work products, including the source and object code for every baseline and change, the test cases, and the development tools.
- It provides locks to prevent unauthorized changes, and it has the capability to build the various system configurations, test drivers, and test scenarios or buckets required by development.

# The System Library

- Primary functions of the **system library**:
  - Retrieving objects by name, date of creation, author, component, or development status
  - Identifying object relationship
  - Executing a make function, such as building a product release, building a private system copy, ...
  - Performing such service functions as checking status, verifying the presence of all build items, ...
  - Providing the required levels of control and protection
  - Gathering and reporting any desired statistical and accounting information

# References

Humphrey, Watts S., *Managing the Software Process*, The SEI Series in Software Engineering, Addison-Wesley, 1989. (29th Printing, May 2003) (ISBN 0-201-18095-2)



---

COLLEGE OF ENGINEERING  
AND COMPUTER SCIENCE

# Advanced Software Process

Part II: The Repeatable Process

*8. Software Quality Assurance*

Dave Garcia-Gomez  
Faculty / Lecturer  
Department of Computer Science

# Course Roadmap

## Part I: Software Process Maturity

- 1. A Software Maturity Framework
- 2. The Principles of Software Process Change
- 3. Software Process Assessment
- 4. The Initial Process

## Part II: The Repeatable Process

- 5. Managing Software Organizations
- 6. The Project Plan
- 7. Software Configuration Management (Part I)
- 8. Software Quality Assurance

## Part III: Defined Process

- 9. Software Standards
- 10. Software Inspections
- 11. Software Testing
- 12. Software Configuration Management (continued)
- 13. Defining the Software Process
- 14. The Software Engineering Process Group

## Part IV: The Managed Process

- 15. Data Gathering and Analysis
- 16. Managing Software Quality

## Part V: The Optimizing Process

- 17. Defect Prevention
- 18. Automating the Software Process
- 19. *Contracting for Software*
- 20. *Conclusion*

# Software Quality Assurance (SQA)

- Quality Management
- The Role of SQA
- Launching the SQA Program
- The SQA Plan
- SQA Considerations
- SQA People
- Independent Verification and Validation

# Quality Management

**Quality** is a **characteristic or attribute of something** [American Heritage Dictionary, quoted from Pressman 2005].

- **Measurable** characteristics
- Things we can compare to known **standards**
- **Quality**
  - Quality of design
    - The characteristics that designers specify for an item
  - Quality of conformance
    - The degree to which the design specifications are followed during manufacturing

# Quality Management

- Software quality
  - Quality of design

Encompasses requirements, specifications, and the design of the system
  - Quality of conformance

An issue focused primarily on implementation
- User satisfaction [Glass 1998]
  - Compliant product + **good quality** + delivery within budget and schedule

# Quality Management

- Quality control (variation control) involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it.
- Quality assurance (QA) consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control activities.

# Quality Management

- The goal of QA is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals.
- Quality management (often called software quality assurance, SQA) is an umbrella activity that is applied throughout the software process.

# Quality Management

- Quality management encompasses:
  - A SQA process
  - Specific QA and Quality Control (QC) tasks (including formal technical reviews and a multi-tiered testing strategy)
  - Effective SE practice (methods and tools)
  - Control of all SW work products and the changes made to them
  - A procedure to ensure compliance with software development standards (when applicable)
  - Measurement and reporting mechanisms.

# Software Quality Assurance

- One of the **critical challenge** for any quality program is to devise a **way for ordinary people** to **review the work of experts.**
- Management properly wants the best designers to design the products, so **Software Quality Assurance (SQA)** cannot have them.
- The need is to focus on those **SQA** methods that **permit the development work** to be reviewed by **people who are not primarily developers.**

# Software Quality Assurance

- The SQA role is to monitor the methods and standards the software experts use and to verify that they have properly applied their expertise.
- SQA is a valid discipline in its own right, and people can be SQA experts without being software design experts.
- This SQA expertise includes knowledge of statistical methods, quality control principles, the software process, and an ability to deal effectively with people in contentious situations.

# Quality Management

- “What is **not tracked** is **not done**.”
- Tracking is the role of **SQA**.
- Before establishing an **SQA** organization, it is essential to first decide how important **software quality** is to the organization.
- **SQA** is a management tool that must be properly used to be effective.
- The prime benefit of an **SQA** program is the assurance it provides management that **the officially established process is actually being implemented**.

# Quality Management

- SQA ensures
  - An appropriate **development methodology** is in place.
  - The projects use **standards and procedures** in their work.
  - **Independent reviews and audits** are conducted.
  - **Documentation** is produced to support maintenance and enhancement.
  - The documentation is produced during and not after development.
  - Mechanisms are in place and used to **control changes**.

# Quality Management

- Testing emphasizes all the high-risk product areas.
- Each software task is satisfactorily completed before the succeeding one is begun.
- Deviations from standards and procedures are exposed as soon as possible.
- The project is auditable by external professionals.
- The quality control work is itself performed against established standards.
- The SQA plan and the software development plan are compatible.

# The Benefits of SQA

- The reason for concern about **SW quality** are compelling.

Software has an enormous impact on almost everything we do, and this impact will only increase in the future.

- Air traffic control
- Air defense control
- Weather forecast
- Banking
- Emergency, police

# The Need for SQA

- When **quality** is vital, some **independent checks** are necessary, not because people are untrustworthy **but because they are human.**
- The issues with software are not whether checks are needed, **but who does them and how.**

# The Goal of SQA

- Broadly stated, the goals of SQA are:
  - To improve software quality by appropriately monitoring both the software and the development process that produces it
  - To ensure full compliance with the established standards and procedures for the software and the software process
  - To ensure that any inadequacies in the product, the process, or the standards are brought to management's attention so these inadequacies can be fixed

# The Goal of SQA

- To be effective, SQA needs to work closely with development.
- SQA need to understand the plans, verify their execution, and monitor the performance of the individual tasks.
- If the development people view SQA as the enemy, it will be hard for them to be effective.
- The key is an SQA attitude of cooperation and support.

# The Role of SQA

- The people responsible for the software projects are the only ones who can be responsible for quality.
- The role of SQA is to monitor the way these groups perform their responsibilities.

# SQA Responsibilities

- SQA can be effective when they report through an independent management chain, when they are properly staffed with competent professionals, and when they see their role as supporting the development and maintenance personnel in improving product quality.
- When all these conditions are met, SQA can help remove the major inhibitors to producing quality software.

# SQA Responsibilities

- Review all **development and quality plans** for completeness
- Participate as **inspection moderators** in design and code inspections
- Review all **test plans** for adherence to **standards**
- Review **a significant sample of all test results** to determine adherences to **plans**
- Periodically **audit SCM (Conf. Mgmt.)** performance to determine adherence to standards
- Participate in all **project quarterly and phase reviews** and register non-concurrence if the appropriate **standards and procedures** have not been reasonably met

# SQA Responsibilities

- Example Items for SQA Review [Table 8.1]

# SQA Functions

- In establishing an **SQA function**, the **basic organizational framework** should include the following:
  - Quality Assurance (QA) practices
  - Software project planning evaluation
  - Requirements evaluation
  - Evaluation of the design process
  - Evaluation of coding practices
  - Evaluating the SW integration and test process
  - In-process evaluation of the management and project control process
  - Tailoring of Quality Assurance (QA) procedures.

# SQA Functions

- Sample SQA Tasks by Program Phase  
[Table 8.2 ]

# SQA Reporting

- The one simple rule on SQA reporting is that it not be under the software development manager.
- SQA should report to a high-enough management level to have some chance of influencing priorities and obtaining the resources and time to fix the key problems.
- Reporting level, however, is a trade-off.
- Since there is no simple solution that meets all needs, a specific reporting level decision should be made for each organization.

# Launching the SQA Program

- The essential first step in establishing an SQA function is to **secure top management agreement** on its goals.
  - Since **the senior managers** must resolve all major SQA issues, they must agree in advance on the basis for doing so.
  - If they do not, SQA cannot be effective.

# Launching the SQA Program

- Eight steps for launching an SQA program:
  - Initiate the SQA program
  - Identify SQA issues
  - Write the SQA plan
  - Establish standards
  - Establish the SQA function
  - Conduct training and promote the SQA program
  - Implement the SQA plan
  - Evaluate the SQA program

# Launching the SQA Program

- In producing the SQA plan, a statistically sound sampling approach is essential.
- It is generally not practical for SQA to review every development action or product item, so the plan should identify the sampling system that will most effectively use the available SQA resources.

# Launching the SQA Program

- Possible sampling methods
  - Ensure that all required design and code inspections are performed, and participate (possibly as monitor) **in a selected set**
  - Review all inspection reports and analyze those **outside of established control limits**
  - Ensure that all required tests are performed and test reports produced
  - Examine a **selected set of test reports** for accuracy and completeness
  - Review all module test results and further study the data on those modules with test histories that are **outside of established control limits**

# The SQA Plan

Each development and maintenance project should have a **Software Quality Assurance Plan (SQAP)** that specifies **its goals**, **the SQA tasks** to be performed, **the standards** against which the development work is to be measured, and **the procedures** and organizational structure.

# The SQA Plan

- Software Quality Assurance Plan (SQAP)
  - Purpose
  - Reference documents
  - Management
  - Documentation
  - Standards, practices, and conventions
  - Reviews and audits
  - SW configuration management
  - Problem reporting and corrective action
  - Tools, techniques, and methodologies
  - Code control
  - Media control
  - Supplier control
  - Records collection, maintenance, and retention

# The SQA Plan

- The section on **standards, practices, and conventions** specifies a minimum content of:
  - Documentation standards
  - Logic structure standards
  - Coding standards
  - Commentary standards

# The SQA Plan

- **Suggested SQAP Documentation [Table 8.3]**
  - The SW Requirements Specification
  - The SW Design Description
  - The SW Verification and Validation Plan
  - The SW Verification and Validation Report
  - User Documentation
  - Other

# The SQA Plan

- The Software Quality Assurance Plan (SQAP) section on reviews and audits should describe both technical and the managerial reviews and audits to be conducted.
  - The IEEE standard includes the items shown in Table 8.4.

# The SQA Plan

- Examples of SQAP Reviews and Audits [Table 8.4]
  - Software Requirements Review
  - Preliminary Design Review
  - Critical Design Review
  - Software Verification and Validation Review
  - Functional Audit
  - Physical Audit
  - In-Process Audits
  - Managerial Reviews

# SQA Considerations

- Many SQA organizations fail to have much impact on software quality.
- Common reasons to fail:
  - SQA organization are rarely staffed with sufficiently experienced or knowledgeable people.
  - The SQA management team is often not capable of negotiating with development.
  - Senior management often backs development over SQA on a large percentage of issues.
  - Many SQA organizations operate without suitably documented and approved development standards and procedure.
  - Software development groups rarely produce verifiable quality plans.

# SQA People

- Getting good people into SQA is one of the most difficult problems software managers face.
- One effective solution is to require that all new development managers be promoted from SQA.
- This would mean that potential managers would spend six months to a year in SQA before being promoted into management back in their home departments.
  - Extreme measure, but effective

# SQA People

- For SQA to be effective, they must have **good people** and **full management backing**.

# Independent Verification and Validation

- In DoD contracts, it is often common to have a separate **Independent Verification and Validation (IV&V)** organization involved.
- Its role is to provide an **independent monitor of the development or maintenance** organization's performance.
- While there can easily be confusion regarding the relative roles of **IV&V** and **SQA**, the distinction should be clear.

# Independent Verification and Validation

- Development management uses SQA to monitor its own organization and to ensure that established standards and procedures are followed.
- IV&V does essentially the same thing for the customer.
- One of the important IV&V role is to ensure that the customer's needs are adequately reflected in the work.

# Independent Verification and Validation

- Another important difference between IV&V and SQA is that IV&V can and should **capitalize on the existence of SQA**.
- If SQA is working effectively, IV&V need **not duplicate** its work, and if it is not, IV&V must **not try to replace it**.
- Their role is to highlight this shortcoming and to get it fixed

# Independent Verification and Validation

- The crucial role of IV&V is to ensure that the right skills and attitudes are in place.
- While they should also review the standards and procedures, they must look beyond them to see if a first-class SW engineering job is being done and if the key risks and feasibility issues are being addressed.

# Independent Verification and Validation

- SQA and IV&V Activities [Table 8.5]
  - DeMillo has studied the performance of several DoD contractors on the respective roles of SQA and IV&V.
  - SQA is more involved in the internal working of the contracting organization, while IV&V tends to look more at application-related issues.

# References

Humphrey, Watts S., *Managing the Software Process*, The SEI Series in Software Engineering, Addison-Wesley, 1989. (29th Printing, May 2003) (ISBN 0-201-18095-2)



---

COLLEGE OF ENGINEERING  
AND COMPUTER SCIENCE

# Advanced Software Process

Part III: The Defined Process

## *9. Software Standards*

Dave Garcia-Gomez  
Faculty / Lecturer  
Department of Computer Science

# Course Roadmap

## Part I: Software Process Maturity

- 1. A Software Maturity Framework
- 2. The Principles of Software Process Change
- 3. Software Process Assessment
- 4. The Initial Process

## Part II: The Repeatable Process

- 5. Managing Software Organizations
- 6. The Project Plan
- 7. Software Configuration Management (Part I)
- 8. Software Quality Assurance

## Part III: Defined Process

- 9. Software Standards
- 10. Software Inspections
- 11. Software Testing
- 12. Software Configuration Management (continued)
- 13. Defining the Software Process
- 14. The Software Engineering Process Group

## Part IV: The Managed Process

- 15. Data Gathering and Analysis
- 16. Managing Software Quality

## Part V: The Optimizing Process

- 17. Defect Prevention
- 18. Automating the Software Process
- 19. *Contracting for Software*
- 20. *Conclusion*

# Part III: The Defined Process

- Once organizations have mastered the **basic capabilities** described in Part II, the performance of their projects will have sufficiently **stabilized** to permit orderly process improvement.
- The priority needs are then to **improve from Level 2 to Level 3**.
- This improvement will establish a **more consistent and uniform process across the organization** and provide a coherent framework for organized learning.

# Part III: The Defined Process

- Part III describes the **key topics** that Level 2 organizations must address to **advance to Level 3**.
- A more detailed listing of the **total set of improvement actions** is included in **Appendix A**.

# Part III: The Defined Process

- The subjects requiring priority management attention at this point (from 2 to 3) are:
  - Standards
  - Software inspections
  - Testing
  - Advanced configuration management topics
  - Process models and architecture
  - Software Engineering Process Group (SEPG)

# Part III: The Defined Process

## Ch.9 – Software Standards

- Benefits of standards, establishment of standards program

## Ch.10 – Software Inspections

- What are software inspections, how an inspection program is initiated, inspection guidelines

## Ch.11 – Software Testing

- Principles of testing, methods, planning, management

## Ch.12 – Software Configuration Management (II)

- Ch.7 basics, management of requirements and design changes, configuration auditing

## Ch.13 – Defining the Software Process

- Process models, process architectures

## Ch.14 – The SE Process Group

- Considerations involved in establishing an SEPG

# Software Standards

- Software Standards
- Definitions
- The Reasons for SW Standards
- Benefits of Standards
- Examples of Some Major Standards
- Establishing SW Standards
- Standards versus Guidelines

# Software Standards

- A standard is a rule or basis for comparison that is used to assess the size, content, value, or quality of an object or activity.
- In software, two kinds of standards are used:
  - Describes the nature of the object to be produced
  - Defines the way the work is to be performed
- Other standards
  - Languages, coding conventions, commenting, change flagging, error reporting

# Software Standards

- Procedures are closely related to standards.
- For example, there are standards for software reviews and audits as well as procedures for conducting them.
  - A review standard specifies review contents, preparatory materials, participants, responsibilities, and the resulting data and reports.
  - The procedure for conducting the review describes how the work is actually to be done, by whom, when, and what is done with the results.

# Software Standards

- Representative Software Standards [Table 9.1]
  - Software quality assurance plans
  - Software development notebooks
  - Software development plans
  - Software reviews and audits
  - Software requirements
  - Software design documentation
  - Software test plans
  - Software quality assurance reviews
  - Software configuration management
  - Problem reporting/corrective action
  - Software documentation

# Software Standards

- Representative Software Procedures [Table 9.2]
  - Auditing Software development notebooks
  - Reviewing a software development plan
  - Conducting software reviews
  - Conducting software audits
  - Reviewing software requirements
  - Reviewing software design documents
  - Reviewing software test plans
  - Auditing the software testing process
  - Conducting SQA reviews
  - Performing Software Configuration Management (SCM)
  - Auditing Software Configuration Management systems
  - Handling problem reporting/corrective action
  - Auditing problem reporting/corrective action systems
  - Reviewing software documentation

# Software Standards

- There are many different ways to establish standards for an organization.
- TRW software development policies, procedures, and standards [Figure 9.1 ]
  - TRW has also published a guidebook of DoD regulations, specifications, and standards.
  - Although it is somewhat out of date, the guidebook does provide a helpful overview of a number of military software standards.

# Software Standards

- TRW software development policies, procedures, and standards [Figure 9.1]

# Definitions

- Before discussing **standards** and **procedures**, it is necessary to establish some **definitions**.
- Software **terms** can be grouped into **categories**, with the meaning understood in the textbook, as follows:

# Definitions

- Authoritative direction on what is to be done:
  - Policy  
A governing principle, typically used as the basis for regulations, procedures, or standards, and generally stated by the highest authority in the organization
  - Regulation  
A rule, law, or instruction, typically established by some legislative or regulatory body
  - Specification  
The precise and verifiable description of the characteristics of a product  
A process specification define a method, procedure, or process to be used in performing a task.  
Specifications are produced by technical experts.

# Definitions

- The characterization of how a task is to be performed or the required characteristics of the result:
  - Guideline  
A suggested practice, method, or procedures
  - Procedure  
A defined way to do something
  - Standard  
A rule or basis for comparison that is used to assess size, content, or value, typically established by common practice or by a designated standards body

# Definitions

- The **ways** in which tasks are accomplished:
  - Convention
    - A **general agreement** on practices, methods, or procedures
  - Method
    - A **regular, orderly procedure** or **process** for performing a task
  - Practice
    - A **usual procedure** or **process**, typically a matter of habit or tacit agreement
  - Process
    - A **defined way** to perform some activity

# Definitions

- The focus in this chapter is on:

- Guidelines

A suggested practice, method, or procedure, typically issued by some authority.

- Procedures

A defined way to do something, generally embodied in a procedures manual.

- Standards

A rule or basis for comparison that is used to assess size, content, or value, typically established by common practice or by a designated standards body.

# The Reasons for Software Standards

- Standards are needed when many people, products, or tools must coexist.
- Standards are essential for establishing common support environments, performing integration, or conducting system test.
  - The fact that everyone knows and understands a common way of doing the same tasks makes it easier for the professionals to move between projects, reduces the need for training, and permits a uniform method for reviewing the work and its status.

# The Reasons for Software Standards

- Standards also promote the consistent use of better tools and methods.
- When everyone uses common coding conventions and commenting guidelines, for example, it is practical for programmers to review each others' work and easier for them to understand it.
- This both facilitates design and code inspections and improves the maintainability of the finished product.

# Benefits of Standards

- While there is *little quantitative evidence* that supports the use of **standards**, most experienced software managers can cite **at least one standard that was key to a program's success.**
- While **standards** alone will **not** make the difference between project success and failure, **they clearly help.**

# Benefits of Standards

- Thayer (1982) has surveyed SW managers to determine their views on the **key software problems** and their most effective solutions.
  - Of all the solutions listed, the **use of enforcement of standards and procedures** ranked first.

# Examples of Some Major Standards

- IBM's Federal Systems Division (FSD) has established a family of **standards for software development** [Table 9.3 Software Design Practices ].
- The **standards** have been an important contributor to the high quality and productivity of IBM's FSD programming group.
  - A set of code management standards
    - Programming languages, coding standards and conventions, computer product support SW, hierarchical program control library, SW development environment
  - Conventions
    - Rules for naming programs, designating variables, and for writing comments

# Examples of Some Major Standards

- IBM's Federal Systems Division (FSD) Software Design Practices [Table 9.3 ]
  - Systematic programming practices
    - Logical expression
    - Program expression
    - Program design
    - Program design verification
  - Systematic design practices
    - Data design
    - Modular design
  - Advanced design practices
    - Software system specification
    - Real-time design

# Establishing Software Standards

- Before establishing an aggressive standards development program, it is wise to formulate an overall plan that considers the available standards, the priority needs of the organization, the status of the projects, the available staff skills, and the means for standards enforcement.

# Establishing Software Standards

- While it is important to establish standards, it is also **important** to concentrate on those standards that can be implemented in a reasonable period and that will provide the most immediate benefit to the organization.

# Establishing Software Standards

- The establishment of standards starts with an examination of the organization's standards and procedures needs.
- This should be considered in three categories, and an effort should be made to maintain a balance of emphasis among them:
  - Management and planning standards and procedures
  - Development process standards and methods
  - Tool and process standards

# Establishing Software Standards

- Management and planning standards and procedures
  - Configuration management
  - Estimating and costing
  - Software Quality Assurance
  - Status reporting

# Establishing Software Standards

- Development process standards and methods
  - Requirement
  - Design
  - Documentation
  - Coding
  - Integration and test
  - Reviews, walkthroughs, and inspections

# Establishing Software Standards

- Tool and process standards
  - Product naming
  - Size and cost measures
  - Defect counting and recording
  - Code entry and editing tools
  - Documentation systems
  - Languages
  - Library system

# Establishing Software Standards

- The standard development process
- Maintaining standards
- Enforcing standards

# The Standards Development Process

- Standards development involves the following steps:
  - Establish a **standards strategy** that defines priorities and recognizes prior work
  - Distribute, review, and maintain this **strategy**
  - Select **individuals or small working groups** to develop the **top-priority standards**
  - This development effort should build on **prior work** where available, define the areas of **applicability**, specify the **introduction strategy**, and propose an **enforcement plan**.

# The Standards Development Process

- The **draft standards** should be widely distributed and reviewed.
- The standards should be **revised** to incorporate the review comments and then **re-reviewed** if the changes are extensive.
- The **standards** should initially be **implemented in a limited test environment**.
- Based on this test experience, the standards should again be **reviewed and revised**.
- **Implement and enforce the standards** across the defined areas of applicability.
- **Evaluate the effectiveness of the standards** in actual practice.

# Maintaining Standards

- Standards must be kept current.
- Standards should be modified and adjusted based on the experience in using and enforcing them, on the changes in available technology, and on the varying needs of the projects.
- If the standards are not maintained, they will gradually become less pertinent to working conditions and enforcement will become progressively less practical.

# Maintaining Standards

- If not corrected, the standard will ultimately become a bureaucratic procedure that takes time without adding value.
- The responsibility for maintaining each standard should be assigned to an individual or group.
- This could be the SW Engineering Process Group, or some other group that has the technical capability and the management charter to handle such technical functions.

# Enforcing Standards

- Standards enforcement is the basic role of the SQA organization.
- They do this with a mix of reviews and tests.
- Exhaustive reviews are most appropriate when automated tools can be used to support the monitoring process or when the standard is so critical that no single deviation is acceptable.
- Otherwise, statistical samples are sufficient unless major problems are encountered.

# Enforcing Standards

- Statistical reviews are used for all other standards and procedures.
- While the level of sampling should be determined for each case, it is often possible for SQA to provide useful help to development as part of the review.
- The effective use of statistical reviews requires that SQA have control over which cases to review and management's support in follow-up actions when the review finds problems.

# Standards Versus Guidelines

- A standard is appropriate when **no further judgment** is needed.
- **Standardization** makes sense when **items are arbitrary** and **must be done uniformly** or when there is **one clearly best alternative**.
  - The definition of coding or naming conventions
  - The selection of a programming language
  - The use of common design methods

# Standards Versus Guidelines

- There are many possible selections, and there may be no clear right or wrong choice, but they must all be done the same way by everyone.
  - Appropriate subjects for standardization

# Standards Versus Guidelines

- Similarly, some **standards** are essential to the maintenance of business or technical control.
  - Standard cost categories
  - Reporting forms
  - Change approval procedures
  - Schedule checkpoints

# Standards Versus Guidelines

- The particular choices are somewhat arbitrary, but standard methods are needed because the support of several different approaches would be expensive and confusing.

# Standards Versus Guidelines

- There are also many cases in which standards are totally inappropriate.
  - Typically these are cases involving technical judgment.
    - The specification of limits on module size
    - Clearly it is not wise to establish a standard unless there is convincing evidence that it is always the right thing to do.

# Standards Versus Guidelines

- In questionable cases, a **guideline** should be used or the **standard** should have some **well-known and practical escape provisions**.
- While such approaches have much **the same effect** as **standards**, they recognize that **exceptions occasionally make sense**.

# References

Humphrey, Watts S., *Managing the Software Process*, The SEI Series in Software Engineering, Addison-Wesley, 1989. (29th Printing, May 2003) (ISBN 0-201-18095-2)



---

COLLEGE OF ENGINEERING  
AND COMPUTER SCIENCE

# Advanced Software Process

Part III: The Defined Process

*10. Software Inspections*

Dave Garcia-Gomez  
Faculty / Lecturer  
Department of Computer Science

# Course Roadmap

## Part I: Software Process Maturity

- 1. A Software Maturity Framework
- 2. The Principles of Software Process Change
- 3. Software Process Assessment
- 4. The Initial Process

## Part II: The Repeatable Process

- 5. Managing Software Organizations
- 6. The Project Plan
- 7. Software Configuration Management (Part I)
- 8. Software Quality Assurance

## Part III: Defined Process

- 9. Software Standards
- 10. Software Inspections
- 11. Software Testing
- 12. Software Configuration Management (continued)
- 13. Defining the Software Process
- 14. The Software Engineering Process Group

## Part IV: The Managed Process

- 15. Data Gathering and Analysis
- 16. Managing Software Quality

## Part V: The Optimizing Process

- 17. Defect Prevention
- 18. Automating the Software Process
- 19. *Contracting for Software*
- 20. *Conclusion*

# Software Inspections

- Types of Reviews
- Inspection Objectives
- Basic Inspection Principles
- The Conduct of Inspections
- Inspection Training
- Reports and Tracking
- Other Considerations
- Initiating an Inspection Program
- Future Directions

# Software Inspections

- Software inspections provide a powerful way to improve the quality and productivity of the SW process.
- This chapter provides overview and additional details are included in *Appendix C*.
- The software inspection is a peer review of a programmer's work to find problems and to improve quality.

# Software Inspections

- The fundamental objective of inspections is to improve the quality of programs by assisting programmers to recognize and fix their own errors early in the software engineering process.
- With large-scale, complex programs, a brief inspection by competent co-workers invariably turns up mistakes the programmers could not have found by themselves.
- An error often starts with an early misconception that is repeated in the design, the code, the documentation, and even the testing.

# Software Inspections

- Inspections help to motivate better work.
- When programmers know their work will be critically examined by their peers in an inspection, they are encouraged to work more carefully either to avoid being embarrassed by sloppy mistakes or through the pride of exhibiting a quality work product.
- By enlisting others in identifying their errors, programmers actually end up doing better work themselves.

# Software Inspections

- Inspections are not magic and they should not be considered a replacement for testing.
- But all software organizations should use inspections or similar technical review methods in all major aspects of their work.
  - Requirements, design, implementation, test, maintenance, and documentation

# Types of Reviews

- There are many different kinds of reviews and many different names for them, so it is important to draw some distinction.
- Types of Reviews [Table 10.1]
  - Management review
  - Technical review
  - Software inspection
  - Walkthrough

# Types of Reviews

- Management and technical reviews are generally conducted for management and typically provide information for management action.
- Inspections and walkthroughs, on the other hand, are peer examinations aimed at assisting the procedures in improving their work.

# Types of Reviews: Objectives

- Management review
  - Ensure progress
  - Recommend corrective action
  - Ensure proper allocation of resources
- Technical review
  - Evaluate conformance to specifications and plans
  - Ensure change integrity

# Types of Reviews: Objectives

- Software inspection
  - Detect and identify defects
  - Verify resolution
- Walkthrough
  - Detect defects
  - Examine alternatives
  - Forum for learning

# Types of Reviews

- Inspection is to examine work technically and provide the producers with an independent assessment of those product areas where improvements are needed.
- Walkthroughs are generally less formal and are often conducted in an educational format.
- Inspections, on the other hand, generally have a formal format, attendance is specified, and data is reported on the results.

# Types of Reviews

- Types of work products for inspections
  - Requirements
  - High-level design
  - Detailed design
  - Implementation
  - Test cases
  - Documentation

# Types of Reviews

- While there is almost no limit on what can be inspected, there is a question of cost.
- Where the cost of inspections does not seem warranted, a less formal walkthrough process is generally adequate.
- Technical reviews can also be used for such items as development and test plans.

# Inspection Objectives

- To find errors at the earliest possible point in the development cycle
- To ensure that the appropriate parties technically agree on the work
- To verify that the work meets predefined criteria
- To formally complete a technical task
- To provide data on the product and the inspection process

# Inspection Objectives

- Inspections also provide a host of secondary benefits:
  - Inspections ensure that associated workers are technically aware of the product.
  - Inspections help to build an effective technical team.
  - Inspections help to utilize the best talents in the organization.
  - Inspections provide people with a sense of achievement and participation.
  - Inspections help the participants develop their skills as reviewers.

# Basic Inspection Principles

- The inspection process follows certain basic principles:
  - The inspection is a formal, structured process with a system of checklist and defined roles for the participants.
  - Generic checklists and standards are developed for each inspection type and, where appropriate, they are tailored to specific project needs.
  - The reviewers are prepared in advance and have identified their concerns and questions before the inspection starts.

# Basic Inspection Principles

- The focus of the inspection is on identifying problems, not resolving them.
- An inspection is conducted by technical people for technical people.
- The inspection data is entered in the process database and used both to monitor inspection effectiveness and to track and manage product quality.

# The Conduct of Inspections

- Inspection should be conducted at every point in the development or maintenance process at which intermediate products are produced.
- Because they are time consuming, involve people from several groups, and use scarce resources, they must be planned well in advance.
- To guarantee that they are done, inspections must be an explicit part of every project plan.

# The Conduct of Inspections

## Basic Set of Inspections [Table 10.2]

Phase	Inspections	Walkthroughs	Technical Reviews
Requirements		Detailed requirements	Initial requirements
Plans			Development plan
Development	Detailed design Code	System design High-level design	
Publications		Draft publication Final publication	
Test		Test implementation	

# Inspection Participants

- The moderator (or inspection leader)
- The producers
  - The person(s) responsible for doing the work being inspected
- The reviewers (or inspectors)
- The recorder (or scribe)

# Inspection Participants

- While many more people may be interested in the inspection results, the purpose of the inspection is to assist the producers in improving their work.
- This can best be done by limiting attendance to five or six reviewers.
- The key point of this attendance list is that only technical peers attend.

# Inspection Participants

- The moderator is not the manager of the work being reviewed, and neither are any of the other participants.
- The inclusion of managers changes the inspection process and distorts the participants' objectivity.
- Regardless of the manager's behavior, the participants will feel that **it is they who are being reviewed rather than the product.**

# Inspection Participants

- Inspection data is gathered to see how well the project is progressing, not to evaluate the people.
- Since reviewers are human, however, they are subject to error, and managers need to study the inspection data to see where improvements are needed.

# Preparing for the Inspection

- The full inspection process consists of a preparation phase, the inspection itself, and some post-inspection activity.
  - The producers and their manager decide that the product is ready for inspection.
  - The inspection participants are identified, the inspection entry criteria are prepared, and the supporting materials are produced for the opening meeting with the entire inspection group.

# Preparing for the Inspection

- The moderator opens this meeting with a brief statement of the subject to be inspected, the inspection objectives, and, if needed, an overview of the inspection process.
- The moderator then provides a copy of the inspection package to each of the participants.
- Following this introductory meeting the reviewers individually prepare for the inspection.
- During preparation the reviewers record their time and the errors identified on the error log form.

# The Inspection and Post-Inspection Actions

- In conducting the inspection meeting, the moderator first checks to see if all participants are prepared and obtains copies of any preparation reports not already submitted.
- The producer(s) then review each major error either to clarify why it is not an error, to understand what the reviewer(s) meant, or to accept it.
- Pertinent data on each error is recorded.

# The Inspection and Post-Inspection Actions

- After discussing all major errors, the product is briefly reviewed to identify any other areas of confusion or concern.
- Based on the inspection results, and after asking the reviewers for their views, the moderator decides whether a re-inspection is required.
  - Sample Re-inspection Criteria [Table 10.3]

# The Inspection and Post-Inspection Actions

- Sample Re-inspection Criteria [Table 10.3]
  - Inspection rates unusual:
    - Inspection time per LOC too short
    - Inspection time per LOC too long
    - Too many errors per programmer hour
    - Too few errors per programmer hour
  - Error data out of line:
    - Too many minor errors, and too few major errors (preoccupied with details)
    - Too many major errors
    - Unusual error distribution
    - Too low a percent of errors found during preparation
  - Other:
    - Any module with more than N errors (N set in project plan)
    - Any module with persistently high error rates
    - The reviewers suggest a reinspection
    - The moderator suggest a reinspection
    - The testers suggest a reinspection
    - The module contains uninspected changes.

# The Inspection and Post-Inspection Actions

- Following the inspection, the **producer(s)** fixes the identified problems and either **reviews** the corrections with the moderator or in a **re-inspection**.
- As the final inspection action, the **moderator** makes sure that **the inspection results** and **data** are inserted in **the process database** and that **management** is **informed** that the inspection has been successfully **completed**.

# Inspection Training

- Moderator training courses are absolutely essential.
- The moderators need a complete grounding in the principles and methods of inspection before they can do a competent job.
- This gives them the basic skills and helps to provide the self-confidence needed to lead such a potentially contentious activity.

# Inspection Training

- As for the participants, training is also highly desirable.
- If a competent moderator is available, however, the software professionals can often learn how to be inspectors by participating in well-run inspections.

# Inspection Training

- Courses should teach inspection principles and provide practice sessions with the checklists and methods involved.
- After the initial moderator training needs have been met, it is then desirable to broaden the training program to include all potential inspection participants.

# Reports and Tracking

- There are several reasons for **gathering data** and **making reports** on the inspection process.
  - It is essential to track **inspection completions** to ensure they are done as required.
  - Much can be learned about **inspection effectiveness** from a brief study of the data gathered.
  - **Inspection metrics** and **data gathering** are discussed in more detail in Chapter 15.
  - Inspection Report [Table 10.4]
  - Inspection Summary [Table 10.5]

# Reports and Tracking

- Inspection Report [Table 10.4]
  - Project
  - System name
  - Moderator
  - Meeting type (overview, reinspection, requirements, design, code)
  - Number of inspections, inspection duration
  - Total number of reviewers, inspection prep time
  - Total lines inspected, pages of diagrams
  - Disposition (accept, conditional, reinspect)
  - Reviewers
  - Producers
  - Recorder

# Reports and Tracking

- Inspection Summary [Table 10.5]
  - Project, System name
  - Moderator
  - Meeting type (overview, reinspection, requirements, design, code)
  - Major errors, minor errors (function, interface, data, logic, i/o, performance, maintenance, standards, documentation, human factors, syntax, other)
  - Distribution (project manager, QA, process group, producers, review coordinator)

# Other Considerations

- Since well-run inspections require intense concentration from all participants, they can be very tiring.
- As a result, inspection sessions should generally not exceed about two hours.
- Inspections involving the same people should not be scheduled back-to-back since the participants will often be too tired to be fully productive.

# Other Considerations

- One inspection session a day is generally all that is advisable for any one individual.
- The moderator should check on this at the time the inspection is scheduled

# Other Considerations

- It is also helpful to assign some inspectors to specific product areas for the project duration.
- If the assignment is done early in the design phase and maintained for the entire project, the inspector's growing product knowledge will greatly facilitate inspection productivity and quality.
- It is wise to include some new reviewers in each inspection, so that they may not lose their ability to see the problems.

# Other Considerations

- It is **rarely desirable** to cover two different designs at the same time.
  - The work should be **split into two inspections**, even if each will take less than one hour.
- It is also wise to **focus on the quality of the result being produced** rather than on the number of errors being found.
  - Errors are a fact of life that must be expected.
  - The need is to **improve the methods and tools** the SW professionals use so the most prevalent error causes can be reduced or eliminated.

# Other Considerations

- Some types of errors cannot be found very efficiently by inspections.
- If, for example, a large number of minor errors were found during preparation, it is wise to discontinue the inspection and have the producers desk check their work more carefully before starting again.
  - It is often helpful to ask one or more of the more experienced reviewers to assist the producers while they recheck.
  - When reviewers become embroiled in minor details, they often overlook more important problems.

# Initiating an Inspection Program

- Inspections have been installed successfully in many organizations with very positive results.
- The way the inspection program is introduced, however, can have an important impact on its effectiveness.

# Initiating an Inspection Program

- The AT&T Bell Laboratories introduced inspections in conjunction with an extensive program of education and consultation (reported in 1984, 1986).
  - Select a location and a key project for the initial effort.
  - Introduce the concept of inspections to the managers and key professionals in a three- to five-hour overview session.
  - Form a working team with one or two project members to determine training requirements, develop the needed forms and procedures, and establish the introduction plan.

# Initiating an Inspection Program

- If trained moderators are not available, conduct a **two- to three-day moderator training class**.
- A **two-day developer workshop** is held to introduce the methods and obtain the support of the professional personnel who will use them.
- After a couple of months' experience in conducting inspections, a **management seminar** is held to outline the results and emphasize the need for continuing management support.
- Periodically, the inspection program is **assessed** and any indicated changes made.

# Initiating an Inspection Program

- After the initial project has obtained some early success, inspections are introduced throughout the organization.
- The final implementation step is to incorporate the use of inspections into the organization's official development process and establish an SQA monitoring program to advise management whenever the established procedures are not followed.

# Inspection Costs

- The effectiveness of inspections depends on the time and effort spent in preparing for and conducting them.
- Optimum inspection rates depend on the type of product involved and the skill and experience of the people doing them.
- Sample Rates for Software Inspections [Table 10.6]
- Relative Inspection Rates with Experience [Table 10.7]

# Inspection Benefits

- There are many examples of inspection quality and productivity benefits, and there are no documented cases of poor experience.
- This, of course, is partly due to people's natural reluctance to write about project failure.

# Inspection Benefits

- The cases in which inspections have **not been effective** have generally had **errors in the way they were conducted**.
- Either the preparation was **not adequate**, too **many people** were involved, the **wrong people** attended, or **too much material** was covered at one time.
- The biggest single problem is generally the combination of **management inattention** and **schedule pressure**.

# Inspection Benefits

- It is becoming clear, however, that **inspections can be highly effective** and that they should be widely used in software development and maintenance.
- The textbook includes many successful stories.
- COBOL program quality before and after code inspection [Table 10.8]
  - A dramatic improvement that can be expected at the initial use of inspection
- Error prevention or detection probabilities [Table 10.9]
  - What techniques could have been used to prevent or detect errors prior to program shipment?

# Inspection Benefits

- Clearly, **inspections** are an important way to **find errors**.
- Not only are they **more effective than testing** for finding many types of problems, but they also **find them earlier** in the program when the cost of making the corrections is far less.
- **Inspection** should be a **required part of every well-run software process**.
  - **Inspections** should be used for every software design, every program implementation, and every change made either during original development, in test, or in maintenance.

# Future Directions

- While inspections are highly cost-effective with the quality of the programs generally produced today, they are also labor-intensive.
  - Each inspection requires the concentrated involvement of a number of talented software professionals who together review each element of the product's design and implementation.
  - While newer, more productive techniques will likely be found, some form of inspections will undoubtedly be needed as long as SE remains a human-intensive process.

# References

Humphrey, Watts S., *Managing the Software Process*, The SEI Series in Software Engineering, Addison-Wesley, 1989. (29th Printing, May 2003) (ISBN 0-201-18095-2)



---

COLLEGE OF ENGINEERING  
AND COMPUTER SCIENCE

# Advanced Software Process

Part III: The Defined Process

*11. Software Testing*

Dave Garcia-Gomez  
Faculty / Lecturer  
Department of Computer Science

# Course Roadmap

## Part I: Software Process Maturity

- 1. A Software Maturity Framework
- 2. The Principles of Software Process Change
- 3. Software Process Assessment
- 4. The Initial Process

## Part II: The Repeatable Process

- 5. Managing Software Organizations
- 6. The Project Plan
- 7. Software Configuration Management (Part I)
- 8. Software Quality Assurance

## Part III: Defined Process

- 9. Software Standards
- 10. Software Inspections
- 11. Software Testing
- 12. Software Configuration Management (continued)
- 13. Defining the Software Process
- 14. The Software Engineering Process Group

## Part IV: The Managed Process

- 15. Data Gathering and Analysis
- 16. Managing Software Quality

## Part V: The Optimizing Process

- 17. Defect Prevention
- 18. Automating the Software Process
- 19. *Contracting for Software*
- 20. *Conclusion*

# Software Testing

- Software Testing Definitions
- Software Testing Principles
- Types of Software Tests
- Test Planning
- Test Development
- Test Execution and Reporting
- Test Tools and Methods
- Real-Time Testing
- The Test Organization

# Software Testing

- Software testing is defined as the execution of a program to find its faults.
- While more time typically is spent on testing than in any other phase of software development, there is considerable confusion about its purpose.
- Many software professionals, for example, believe that tests are run to show that the program works rather than to learn about its faults.

# Software Testing

- Dijkstra, 1969
  - “Program testing can be used to show the presence of bugs, but never their absence!”
- In fact, testing is an inefficient way to find and remove many types of bugs.
- Software organizations can often improve product quality while at the same time reducing the amount of time they spend on testing.
  - Software inspections
- In spite of its limitations, however, testing is a critical part of the software process.

# Definitions [Myers 1976]

- Testing
  - The process of executing a program (or part of a program) with the intention of finding errors
- Verification
  - An attempt to find errors by executing a program in a test or simulated environment; It is now preferable to view verification as the process of proving the program's correctness.
- Validation
  - An attempt to find errors by executing a program in a real environment
- Debugging
  - Diagnosing the precise nature of a known error and then correcting it; Debugging is a correction and not a testing activity

# Definitions [Pressman 2005]

- Verification
  - Verification refers to the set of activities that ensure that software correctly implements a specific function.
  - Are we building the product right?
- Validation
  - Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
  - Are we building the right product?

# Definitions [Pressman 2005]

Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, DB review, algorithm analysis, development testing, usability testing, qualification testing, and installation testing.

# The Seven Types of Software Tests

- Unit or module tests
- Integration tests
- External function tests
- Regression tests
- System tests
- Acceptance tests
- Installation tests

# Testing Methods

Two basic ways of **constructing tests**:

- **White box tests**
  - Examine the internal design of the program and require that the tester has **detailed knowledge of its structure**
- **Black box tests**
  - Are designed without knowledge of the program's internals and are generally based on the functional requirements

# Software Testing Principles

- Software testing presents a problem in economics.
- With large systems it is almost always true that more tests will find more bugs.
- The question is not whether all the bugs have been found but whether the program is sufficiently good to stop testing.
- This trade-off should consider the probability of finding more bugs in test, the marginal cost of doing so, the probability of the users encountering the remaining bugs, and the resulting impact of these bugs on the users.

# The Axioms of Testing

- A **good test case** is one that has a **high probability of detecting a previously undiscovered defect**, not one that shows that the program works correctly.
- One of the most difficult problems in **testing** is knowing **when to stop**.
- It is impossible to test your **own** program.
- A necessary part of every **test case** is a description of **the expected output**.
- Avoid **non-reproducible** or on-the-fly testing

# The Axioms of Testing

- Write test cases for invalid as well as valid input conditions
- Thoroughly inspect the result of each test
- As the number of detected defects in a piece of software increases, the probability of the existence of more undetected defects also increases.
- Assign your best programmers to testing
- Ensure that testability is a key objective in your software design

# The Axioms of Testing

- The design of a system should be such that each module is integrated into the system only once.
- Never alter the program to make testing easier (unless it is a permanent change)
- Testing, like almost every other activity, must start with objectives.

# The Proper Role of Testing

- An examination of even relatively simple programs demonstrate that **exhaustive testing is generally impossible.**
- **Test design** thus reduces to the judicious selection of a small subset of conditions that will reveal the characteristics of the program.

# Types of Software Tests

- When **unit tests** are done on a **white box** basis, they are essentially **path tests**.
- The idea is to **focus on a relatively small segment of code** and aim to **exercise a high percentage of the internal paths**.
- A **path** is an **instruction sequence that threads through the program** from initial entry to final exit.
  - The simplest approach is to ensure that **every statement is exercised at least once**.
  - A more stringent criterion is to require **coverage of every path** within a program.

# Unit Testing (White Box)

- One disadvantage of white box testing is that the tester may be biased by previous experience.
- The tests are often designed by the programmers who produced the code since they may be the only ones who understand it.
- Unfortunately, those who created the program's faults are least likely to recognize them.

# Unit Testing (White Box)

- Another limitation of white box testing is coverage.
- Even if each instruction is tested and every branch traversed in all directories, many other possible combinations may still be overlooked.
  - All possible combinations of every loop
  - All possible values for every parameter
  - All allowed and un-allowed but possible values for all variables

# Unit Testing (White Box)

- While its disadvantages are significant, white box testing generally has **the highest error yield** of all testing techniques.
- In fact, in a retrospective study, Thayer and Lipow (1978) state that **comprehensive path and parameter testing** could potentially have caught **72.9 percent** of all the problems encountered by the users of one large program.

# Integration Testing

- The proper approach to integration depends on both the kind of system being built and the nature of the development project.
- With a new system, for example, there is no foundation on which to assemble and to run newly developed program fragments.
- The initial problem is thus to establish a test framework on which to run these various elements.

# Integration Testing Approaches

- On very large systems it is often wise to do integration testing in several steps.
- Such systems generally have several relatively large components that can be built and integrated separately before combination into a full system.
- Table 11.1 Top-down and Bottom-up Integration [Myers 1976]

# Integration Testing Approaches

- In **bottom-up testing** the modules are individually tested, using **specially developed drivers** that provide the needed system functions.
- As more modules are integrated, these **drivers** are replaced by the modules that perform those functions.
- The **disadvantage** of **bottom-up testing** are **the need for drivers** and **the amount of testing required before functional testing** is possible.

# Integration Testing Approaches

- Top-down testing is essentially a prototyping philosophy.
- The initial tests establish a basic system skeleton from the top and each new module adds capability.
- The problem is that the functions of the lower-level modules that are not initially present must be simulated by program stubs.
  - Often requires very sophisticated stubs
- With top-down testing it may be difficult or impossible to test certain logical conditions such as error handling or special checking until most of the system has been integrated.

# System Build

- Since integration is a process of incrementally building a system, there is often a need to have special groups do this work.
  - In building large software system, build experts often integrate the components in the system builds (spins), maintain configuration management control, and distribute the builds back to development for module and component test.
  - These experts work with development to establish an integration plan and then build the drivers and integrate the system.

# Function Testing (Black Box)

- Functional (or black box) tests are designed to exercise the program to its external specifications.
- The testers are typically not biased by knowledge of the program's design and thus will likely provide tests that resemble the user's environment.
  - The two most typical problems with black box testing are the need for explicitly stated requirements and the ability of such tests to cover only a small portion of the possible test conditions.

# Regression Testing

- Testing is both **progressive** and **regressive**.
- The **progressive** phase introduces and tests new functions, uncovering problems in the newly added or modified modules and in their interface with the previously integrated modules.
- The **regressive** phase concerns the effects of the newly introduced changes on all the previously integrated code.

# Regression Testing

- Problems arise **when errors made in incorporating new functions affect previously tested functions.**
- In large software systems these **regression problems** are common.

# Regression Testing

- Regression testing is particularly important in software maintenance.
- The basic regression testing approach is to incorporate selected test cases into a regression test bucket that is run periodically in an attempt to detect regression problems.
  - In many software organizations regression testing consists of re-running all the functional tests every few months.
  - This delays the discovery of regression problems and generally results in significant rework after every regression run.

# System Test

- The purpose of system tests is to find those cases in which the system does not work as intended, regardless of the specifications.
- Regardless of what the contracts says, if the system does not meet the user's real needs, **every one loses**.
  - Some areas that need special system test attention are **operational errors** and **intentional misuse**.
  - With the increasingly pervasive use of sophisticated computing systems in modern society, we must also be more concerned about the problems of **intentional misuses** by vandals, criminals, and even terrorists.

# System Test

- The program objectives should specify what is to be done for the end users.
- When these objectives are not specific, neither the testing nor the system design and implementation can be accurately planned.
- When systems are built without good objectives, they must often be rebuilt before they can be used.

# System Test

- System test planning can be done by a special organization with a reasonably direct link to the end users, possibly through a users group or by close contact with selected key users.
- System test planning will often uncover problems that have been undiscovered throughout the entire development process.
- Categories of System Tests [Table 11.2]

# System Test Categories

- Load/stress
- Volume
- Configuration
- Compatibility
- Security
- Performance
- Installability
- Reliability/Availability
- Recovery
- Serviceability
- Human factors

# Acceptance and Installation Tests

- After all development testing is completed, it is often advisable to **try the system in a real user's environment.**
- Even though such **field tests** generally require special support, the results are invariably worth far more than the added costs.
- Often when software is developed under contract, some form of **acceptance testing** is required **in a real or simulated user environment.**

# Acceptance and Installation Tests

- When this is not the case, the developers can often arrange with **selected users** to act as special “**beta test**” sites in return for special installation support and early program availability.
- If end user testing is not practical, it may be possible to try the system in an internal application environment.

# Test Planning

- Test planning starts with an overall development plan that defines the **functions, roles, and methods** for all test phases.
- Test Plan Review Checklist [Table 11.3]

# The Test Files

- It is helpful to establish a development file system to retain this information both for the test plan in general and for each test and test case.
- These files should be defined and provisions made to retain them as part of the configuration management plan.
- Such files provide a useful way to relate all the material on each test and help to establish a test DB.

# The Test Files

- This file should also contain the specifications, design, documentation, review history, test history, test instructions, anticipated results, and success criteria for each test case.

# Test Success Criteria

- Establishing the **test success criteria** is probably the most difficult part of test planning, due to both the developer's attitudes and to their general lack of quantitative experience.
- The **developer's** viewpoint is that a successful test is one that executes completely without encountering a problem.
- The **testers** measure success, however, by bug found.

# Test Success Criteria

- Each test should be planned **against yield criterion (bugs per test run)** that is determined from prior experience.
- The **testers'** goal should then be to increase test case yield, while the **developers'** is to produce code that will reduce test yield.

# Test Success Criteria

- When the testers acknowledge that they cannot meet their yield objectives, their work should be technically reviewed to see if it is adequate.
- The testers should determine why they did not catch any bug that is later found and how to improve their tests so they would catch similar problems in the future.

# Test Development

- After developing the **test plan**, the next job is to produce the **test cases**.
- The decision on what **test cases** to develop is complicated by two factors.
  - Full test coverage is generally impossible.
  - There is **no proven comprehensive method** for selecting the highest yielding test conditions.

# Test Development

- From an experiment, Myers (1976) found that, even with very experienced programmers, the likelihood of their producing tests that covered a reasonably complete set of all test combinations was quite small.
- Test Results for Triangle Program [Table 11.4]
  - The Triangle Program read three integers representing the sides of a triangle and printed out whether the triangle was scalene, isosceles, or equilateral.
  - The percentage of programmers (test design) who checked for specific bugs

# Test Development

- Because even experienced software professionals **make errors and overlook conditions**, it is essential to hold **walkthroughs of the test plans and inspections of the test cases**.
- Unit Test Review Checklist [Table 11.5]

# Test Coverage Techniques

- One approach to the **test coverage** problem is to **view the program as a graph** with nodes and to ensure that these nodes and the paths between them are adequately covered by the tests.
- The adequacy of **test coverage** can be roughly assessed by comparing **the number of test paths** executed with the complexity of the program graph.

# Path Selection In Unit Test

- While there are **no simple rules** for selecting **testing paths** to achieve adequate **test coverage**, some **general guidelines** can help:
  - Pick defined (in the requirements) functional paths that simply connect from entrance to exit
  - Pick additional paths as small variations, favoring short paths over long ones, simple paths over complicated ones, and logically reasonable paths over illogical ones.
  - Next, to provide coverage, pick additional paths that have no obvious functional meaning

# Path Coverage in Functional Testing

- For functional or black box testing, one technique for selecting tests is based on functional paths.
  - Here the program functions are considered as a set of transaction flows that form functional paths through the program.
  - These transaction flows can be reduced to graphs of nodes and paths much as in white box testing, and the tests are again designed to achieve coverage of these nodes and paths.
  - These then form the paths to be covered by the functional tests in much the way as with unit tests.

# Path Selection

- With a defined set of **functional paths** it is necessary to decide **which paths to cover**.
- While there are **no guidelines for functional path selection**, it is generally a good idea to start by **considering the data** handled by the program.
- Functional Testing Data Selection Guidelines [Table 11.6]

# Path Selection

- While there is no magic way to select a sufficient set of practical tests, the objective is to **test reasonably completely all valid classes for normal operations** and to **exhaustively test behavior under unusual and illegal conditions**.
- One way to look at the adequacy of a test is to determine the degree to which **all the requisite conditions have been covered**.

# A Suggested Path Selection Technique

- Prather (1987) and Myer (1979) also have suggested **techniques for handling path selection**.
- The general approach is to establish **equivalence classes** based on the various parameters and conditions for the program.
- For each parameter or condition, an **equivalence class** would cover all valid values.
- An **equivalence class** would also be established for each invalid class, such as above range or below range.

# Test Case Design Guidelines

- Since the **objective of test case design** is to **catch errors** rather than to prove the program works, it is wise to consider the **common types of errors**.
- Major errors [Goodenough and Gerhart 1975]
  - Missing control flow paths
  - Inappropriate path selection
  - Inappropriate or missing action

# Test Case Design Guidelines

- Even though there are **no simple rules for test case design**, some **general guidelines** can be helpful:
  - Do it wrong
  - Use wrong combinations
  - Don't do enough
  - Do nothing
  - Do too much

# Test Case Design Guidelines

- In addition, some more overall considerations are:
  - Don't forget the normal cases!
  - Don't go overboard with cases.
  - Don't ignore structure.
  - Remember that there is more than one kind of test.

# Test Reports

- At the conclusion of each test, a **test report** should be produced.
- The **amount of detail** included **depends on the purpose of the test** and **the audience for the report**.
- In the case of **acceptance tests**, a considerable **amount of detail** is generally required, including the qualification of the test observers, **detailed data on test results**, and the plans to resolve the trouble reported.

# Test Reports

- Sample Test Incident Form [Table 11.7]
- Sample Bug Report Form [Table 11.8]

# Bug Classification

- As part of the test reporting process, it is desirable to have standard definitions for bug severity.
- This allows the follow-up actions to be prioritized and the critical items to be tracked.
- Bug Severity Classification – Military System [Table 11.9]
- Bug Severity Classification [Beizer 1983] [Table 11.10]

# Test Analysis

- Since tests are intended to gather data on programs, the resulting information should be analyzed and used to make decisions.
- In general the evidence for jungle-like conditions can be found by examining the test results and the characteristic of the problem found.

# Test Tools and Methods

- Test Tools [Table 11.11]
  - A list of available testing tools in a document TRW prepared for the Air Force on “Software Testing and Evaluation

# Real-Time Testing

- Real-time testing can be particularly difficult because the development work is done on a host system and then compiled for execution on a target system.
- Typically a reasonable set of test and debug facilities is available for the host environment but the target system is generally much more sparsely equipped.

# The Test Organization

- Bug guilt [Beizer 1983]
- Programmers are inherently incapable of effectively testing their own programs.
- Not only do they feel guilty about the existence of bugs, but they are biased by the creative work they have done.
- One remedy is to separate testing from program design and implementation.

# The Test Organization

- Such **special test groups** often assume test responsibility after unit testing.
  - This generally makes sense because **white box unit tests** generally require an intimate knowledge of the program's internal structure.
  - **Unit test standards** and **walkthroughs of unit test plans** can help the programmers do a reasonably effective job of unit testing their own code.

# The Test Organization

- Following unit test it is generally advisable to transfer test responsibility to a dedicated test group.
  - This group's role is to find as many bugs as possible.
  - As they gain experience, such groups can become extraordinarily effective at finding those odd pathological cases that will only come up once in a billion executions.

# References

Humphrey, Watts S., *Managing the Software Process*, The SEI Series in Software Engineering, Addison-Wesley, 1989. (29th Printing, May 2003) (ISBN 0-201-18095-2)



---

COLLEGE OF ENGINEERING  
AND COMPUTER SCIENCE

# Advanced Software Process

## The Agile Process

### *Overview*

Dave Garcia-Gomez  
Faculty / Lecturer  
Department of Computer Science

# Contents

- Key Practices of Agile Process
- The Agile Manifesto
- Principles behind the Agile Manifesto
- Agile Development Methods

# Key Practices of Agile Process

- Iterative development
- Risk-driven and client-driven
- Time-boxing
- Evolutionary development
- Adaptive planning
- Incremental delivery
- Evolutionary delivery
- ...

# Iterative Development

Iterative development is an approach to building software (or anything) in which the overall lifecycle is composed of several iterations in sequence.

- Each iteration is a self-contained mini-project composed of activities such as requirements analysis, design, programming, and test.
- The goal for the end of an iteration is an iteration release, a stable, integrated and tested partially complete system.

<http://guide.agilealliance.org/guide/iterative.html>

# Risk-Driven and Client-Driven Iterative Planning

ID methods promote a combination of **risk-driven** and **client-driven priorities**.

**Risk-driven iterative development** chooses the **riskiest, most difficult elements** for the early iterations.

**Client-driven iterative development** implies that the choice of features for the next iteration comes from the **client – whatever they perceive as the highest business value** to them.

# Timeboxed Iterative Development

All the modern ID methods (including Scrum, XP, and so forth) either require or strongly advise **timeboxing** the iterations.

Fixed time for each iteration

**Timebox:** A timebox is a previously agreed period of time during which a person or a team works steadily towards completion of some goal. Rather than allow work to continue until the goal is reached, and evaluating the time taken, the timebox approach consists of stopping work when the time limit is reached and evaluating what was accomplished. Timeboxes can be used at varying time scales. Time scales ranging from one day to several months have been used. The critical rule of timeboxed work is that work should stop at the end of the timebox, and review progress: has the goal been met, or partially met if it included multiple tasks?

Source: <http://guide.agilealliance.org/guide/timebox.html>

# Evolutionary and Adaptive Development

Evolutionary iterative development implies that the requirements, plan, estimates, and solution evolve or are refined over the course of the iterations, rather than fully defined and “frozen” in a major up-front specification effort before the development iterations begin.

Evolutionary methods are consistent with the pattern of unpredictable discovery and change in new product development.

# Evolutionary and Adaptive Development

Adaptive development implies that elements adapt in response to feedback from prior work – feedback from users, tests, developers, and so on.

The intent is the same as evolutionary development, but the name suggests more strongly the feedback-response mechanism in evolution.

# Incremental Delivery

Incremental delivery is the practice of repeatedly delivering a system into production (or the marketplace) in a series of expanding capabilities.

The practice is promoted by IID and Agile methods.

Incremental deliveries are often between three and twelve months.

<http://guide.agilealliance.org/guide/incremental.html>

# Evolutionary Delivery

Evolutionary delivery is a refinement of the practice of incremental delivery in which there is a vigorous attempt to capture feedback regarding the installed product, and use this to guide the next delivery.

# Agile Development

Agile development methods apply timeboxed iterative and evolutionary development, adaptive planning, promote evolutionary delivery, and include other values and practices that encourage agility

- rapid and flexible response to change.

# The Agile Manifesto

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

# Principles behind the Agile Manifesto

1. Our highest priority is to **satisfy the customer** through **early and continuous delivery** of valuable software.
2. **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
4. **Business people and developers must work together daily** throughout the project.
5. **Build projects around motivated individuals**. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.

# Principles behind the Agile Manifesto

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

# Agile Development Methods

- Scrum
- XP
- Evo
- Crystal
- Agile Modeling
- (Agile) Unified Process (Agile UP)
- Dynamic Solutions Delivery Model (DSDM)
- Feature-Driven Development (FDD)
- Lean Development
- ...

# References

- Craig Larman. Agile & Iterative Development: A Manager's Guide. Addison-Wesley, Pearson Education, 2004. (11th Printing, Aug. 2009) (ISBN-10: 0-13-111155-8, ISBN-13: 978-0-13-111155-4)
- Agile Alliance: <http://www.agilealliance.org/>
  - <http://www.agilemanifesto.org/>
  - <http://www.agilemanifesto.org/principles.html>
  - <http://guide.agilealliance.org/subway.html>
- Agile Modeling and RUP – Scott Ambler: <http://www.agilemodeling.com/>
- Agile, XP Method – Martin Fowler: <http://www.martinfowler.com/>
- Agile, XP Method – Robert Martin: <http://www.objectmentor.com/>
- Extreme Programming – Don Well: <http://www.extremeprogramming.org/>
- Extreme Programming – Ron Jeffries: <http://xprogramming.com/index.php>
- Agile Development – Artern Marchenko :  
<http://agilesoftwaredevelopment.com/>



---

COLLEGE OF ENGINEERING  
AND COMPUTER SCIENCE

# Advanced Software Process

The Agile Process

*Scrum*

Dave Garcia-Gomez

Faculty / Lecturer

Department of Computer Science

# Contents

- Scrum Overview & Scrum Practices
- Scrum Lifecycle
- Work-products
- Roles
- Practices
- Sample Projects & History

# Scrum Overview

Scrum has practices that capture key adaptive and agile qualities.

Scrum's distinctive emphasis among the methods is its strong promotion of

- self-directed teams
- daily team measurement
- avoidance of prescriptive process.

# Scrum Overview

Scrum's practices include:

- self-directed and self-organizing team
- no external addition of work to an iteration, once chosen
- daily stand-up meeting with special questions
- usually 30-calendar day iterations
- demo to external stakeholders at end of each iteration
- each iteration, client-driven adaptive planning

# Method Overview - Classification

Scrum on the **cycles** and **ceremony** scale (Figure 7.1)

Scrum is uniquely precise on the length of iterations: usually 30 calendar days, a more-or-less common length compared to other IID methods.

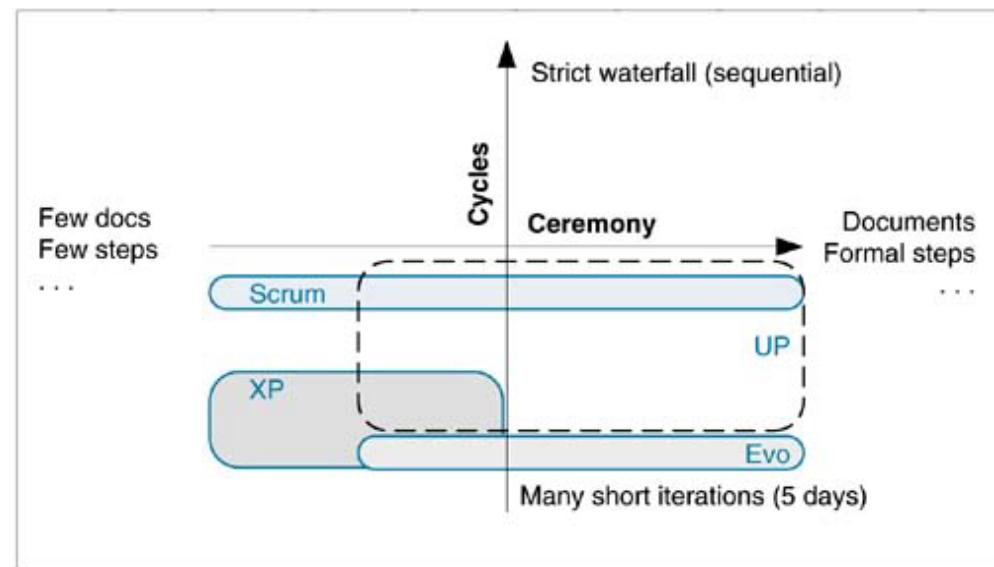
Shorter is legal, but 30-day iterations are encouraged.

Scrum: 30 days (4w)

XP: 1 – 3 weeks

UP: 2 – 6 weeks

Evo: 1 – 2 weeks



# Method Overview - Classification

Scrum is **flexible on the ceremony scale**; discussion of what and how many work products is outside its scope, as is how much rigor is required.

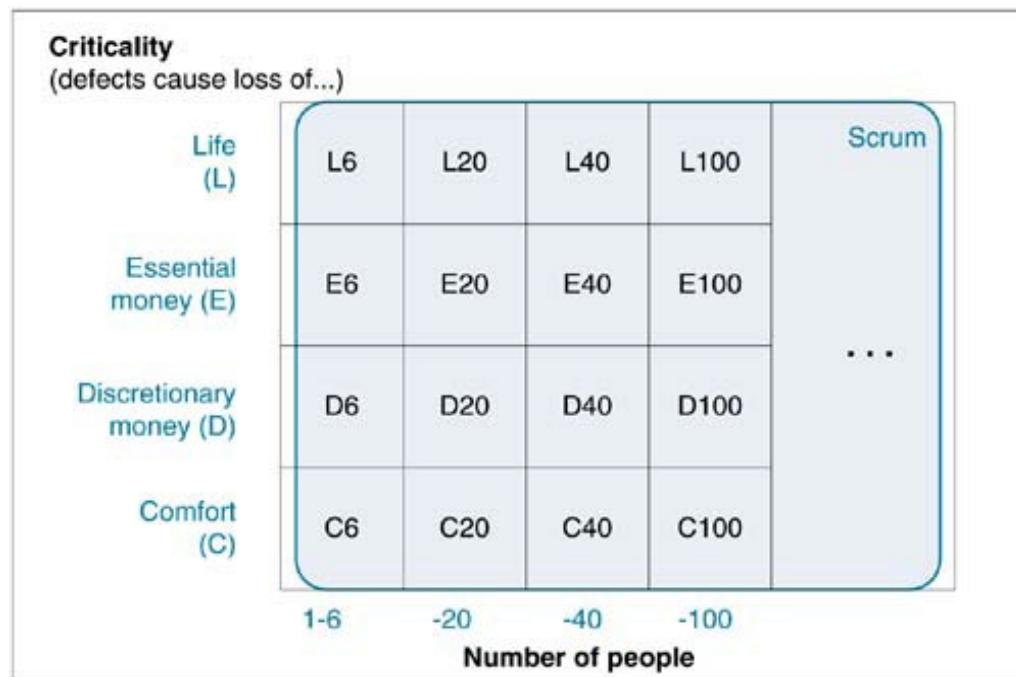
As a guiding principle, the Scrum founders would say, **“as little ceremony as possible.”**

Also on a Scrum project, **the whole team – not a manager – will decide how much is appropriate.**

# Method Overview - Classification

In terms of scope on the Cockburn scale, Scrum covers the cells shown in the following figure.

Scrum on the Cockburn scale (Figure 7.2)



A method selector = a function of (criticality, size, priority): how much formal process (ceremony, cycles) a software project requires: Cockburn scale (criticality, size)

- L: Loss of Life
- E: Loss of Essential Money
- D: Loss of Discretionary Money
- C: Loss of Comfort

Reference:  
<http://alistair.cockburn.us/Cockburn+Scale>

# Method Overview - Classification

Scrum is complementary enough to other practices that it may be applied across all domains of software applications, from life-critical to more casual – and it has.

Although one Scrum team should be seven or less, multiple teams may form a project.

- Since Scrum practices include working in a common project room, it scales via a “scrum of scrums”
- Where small teams work together and hold a daily stand-up meeting, and representatives from each those teams likewise meet daily.

# Method Overview - Introduction

Scrum is an IID method that emphasizes a set of project management values and practices, rather than those in requirements, implementation, and so on.

As such, it is easily combined with or complementary to other methods.

A key Scrum theme is its emphasis on empirical rather than defined process.

# Lifecycle

The Scrum lifecycle is composed of four phases:

Pre-game

- Planning
- Staging

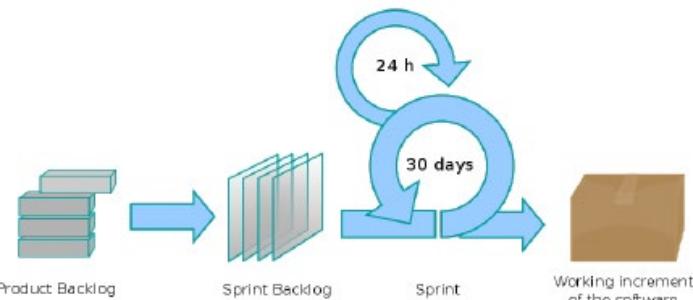
Development

Release

PLANNING	PRE-GAME	STAGING	DEVELOPMENT	RELEASE
<b>Purpose:</b> <ul style="list-style-type: none"><li>- establish the vision, set expectations, and secure funding</li></ul> <b>Activities:</b> <ul style="list-style-type: none"><li>- write vision, budget, initial Product Backlog and estimate items</li><li>- exploratory design and prototypes</li></ul>	<b>Purpose:</b> <ul style="list-style-type: none"><li>- identify more requirements and prioritize enough for first iteration</li></ul> <b>Activities:</b> <ul style="list-style-type: none"><li>- planning</li><li>- exploratory design and prototypes</li></ul>	<b>Purpose:</b> <ul style="list-style-type: none"><li>- implement a system ready for release in a series of 30-day iterations (Sprints)</li></ul> <b>Activities:</b> <ul style="list-style-type: none"><li>- Sprint planning meeting each iteration, defining the Sprint Backlog and estimates</li><li>- daily Scrum meetings</li><li>- Sprint Review</li></ul>	<b>Purpose:</b> <ul style="list-style-type: none"><li>- operational deployment</li></ul> <b>Activities:</b> <ul style="list-style-type: none"><li>- documentation</li><li>- training</li><li>- marketing &amp; sales</li><li>... etc.</li></ul>	

# Workproducts, Roles, and Practices

## Workproducts:



<b>WORKPRODUCTS (non-software)</b>	<b>Requirements</b>	<b>Design</b>
<b>Product &amp; Release Backlog</b>	All items for the product. The Release Backlog is a subset.  Includes features, use cases, enhancements, defects, technologies.	
	<b>Implementation</b>	<b>Test &amp; Verification</b>
<b>Project Management</b>	<b>Configuration &amp; Change Management Environment</b>	
<b>Backlog Graph</b>	Estimate of work remaining versus days.	
<b>Sprint Backlog</b>	Tasks for the iteration. Granularity 4-16 hours.	
<b>Product &amp; Release Backlog</b>		

# Workproducts, Roles, and Practices

## Roles:

### Chickens vs. Pigs

- Chickens: involve, consult on the project and are informed of its progress.
- Pigs: committed to the project and accountable for its outcome

Source:

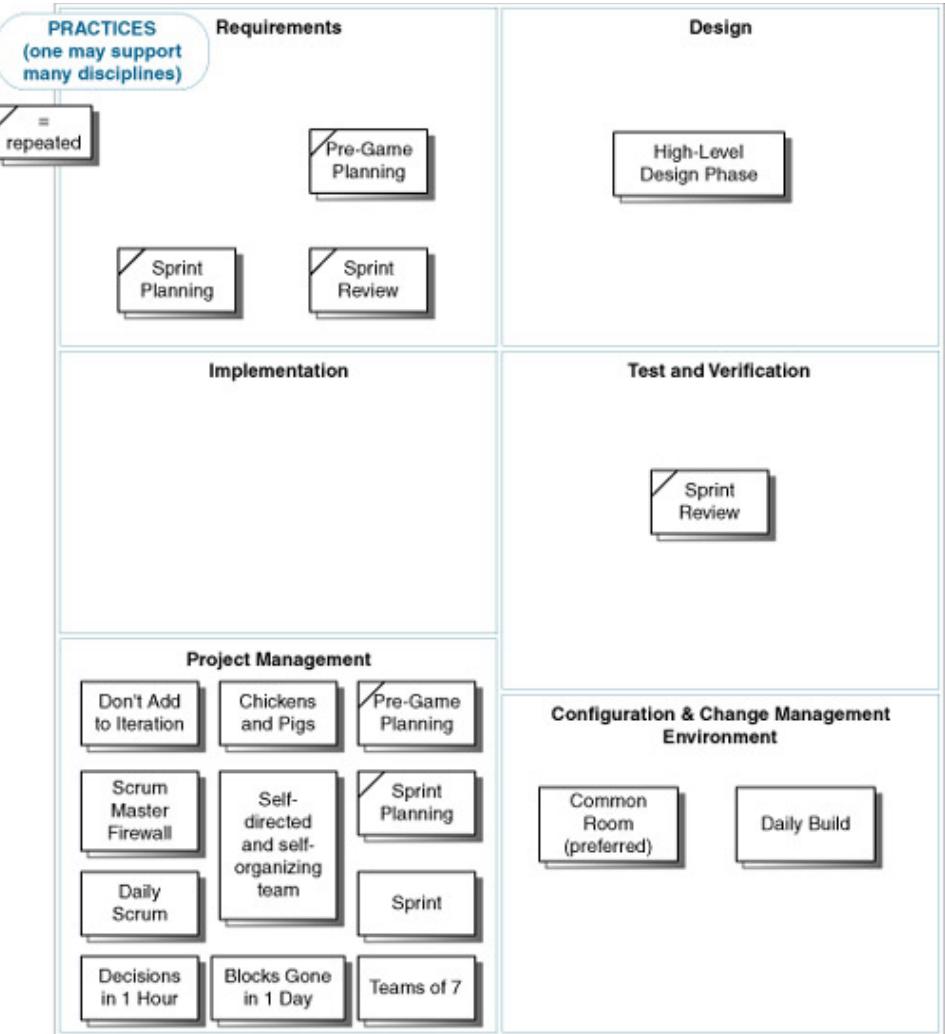
<http://www.implementingscrum.com/2006/09/11/the-classic-story-of-the-pig-and-chicken/>



ROLES	Customer	Development
	 <b>Product Owner</b> <ul style="list-style-type: none"><li>- one person who is responsible for creating and prioritizing the Product Backlog</li><li>- chooses the goals (from the Product Backlog) for the next Sprint</li><li>- along with other stakeholders, reviews the system at the end of each Sprint</li></ul>	 <b>Scrum Team</b> <ul style="list-style-type: none"><li>- work on the Sprint (iteration) Backlog</li><li>- there is explicitly no other title than "team member"</li></ul>
	 <b>Scrum Master</b> <ul style="list-style-type: none"><li>-50% developer, not just management</li><li>-knows and reinforces the project and iteration vision and goals</li><li>- ensures Scrum values and practices followed</li><li>- mediates between Management and Scrum Team</li><li>- listens to progress and removes impediments</li><li>- conducts the Daily Scrum</li><li>- conducts the Sprint Review (demo)</li></ul>	 <b>Chickens</b> <ul style="list-style-type: none"><li>- everyone else can observe, but not interfere or speak during an iteration</li></ul>

# Workproducts, Roles, and Practices

## Practices:



# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Pre-game planning and staging	During Pre-game Planning, all stakeholders can contribute to creating a list of features, use cases, enhancements, defects, and so forth, recorded in the Product Backlog. One Product Owner is designated its owner, and requests are mediated through her. During this session, at least enough work for the first iteration is generated, and likely much more. Starting at these meeting and evolving over time, is identification of the Release Backlog, the subset of the Product Backlog that will make the next operational or product release. .

# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Sprint planning	<p>Before the start of each iteration – or Sprint – two consecutive meetings are held. In the first, stakeholders meet to refine and re-prioritize the Product Backlog and Release Backlog, and to choose goals for the next iteration, usually driven by highest business value and risk. In the second meeting, the Scrum Team and Product Owner meet to consider how to achieve the requests, and to create a Sprint Backlog of tasks (in the 4–16 hour range) to meet the goals. If estimated effort exceeds resources, another planning cycle occurs. As the iteration proceeds, the Sprint Backlog is updated, often daily during the early part of the iteration, as new tasks are discovered. As a history of many Sprint Backlogs grows, the team improves their creation of new ones.</p>

# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Sprint	Work is usually organized in 30-calendar-day iterations; each is called a Sprint.
Self-directed and self-organizing teams	During an iteration, management and the Scrum Master do not guide the team in how to fulfill the iteration goals, solve its problems (other than to make decisions when requested, and remove reported blocks), nor plan the order of work. The team is empowered with the authority and resources to find their own way, and solve their own problems. This hands-off approach for 30 days, except to provide resources and remove blocks, is perhaps the most personally challenging aspect for management adopting Scrum.

# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Scrum meeting	Each workday at the same time and place, hold a meeting with the team members in a circle, at which the same special Scrum questions are answered by each team member.
Don't add to iteration	During an iteration, management does not add work to the team or individuals. Uninterrupted focus is maintained. In the rare case something has to be added, something else should ideally be removed. But, before each new iteration, the Product Owner and Management have the right and responsibility to re-prioritize the Product Backlog, and indicate what to do in the next iteration, as long as the work request estimates don't exceed the resources.

# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Scrum master firewall	<p>The Scrum Master looks out to ensure the team is not interrupted by work requests from other external parties, and if they occur, removes them and deals with all political and external management issues. The Scrum Master also works to ensure Scrum is applied, removes reported blocks, provides resources, and makes decisions when requested. She also has to take initiative when she sees during the meeting that someone isn't completing work, if the team doesn't speak up.</p>
Decision in one hour	<p>Blocks reported at the Scrum Meeting that require decisions by the Scrum Master are ideally decided immediately, or within one hour. The value of “bad decisions are better than no decisions, and they can be reversed” is promoted.</p>

# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Decision in one hour	The Scrum Master looks out to ensure the team is not interrupted by work requests from other external parties, and if they occur, removes them and deals with all political and external management issues. The Scrum Master also works to ensure Scrum is applied, removes reported blocks, provides resources, and makes decisions when requested. She also has to take initiative when she sees during the meeting that someone isn't completing work, if the team doesn't speak up.
Decision in one hour	Blocks reported at the Scrum Meeting that require decisions by the Scrum Master are ideally decided immediately, or within one hour. The value of “bad decisions are better than no decisions, and they can be reversed” is promoted.

# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Blocks gone in one day	Blocks reported at the Scrum Meeting are ideally removed before the next meeting.
Chickens and pigs	<p>During the Scrum Meeting, only the Scrum Team can talk (the pigs). Anyone else can attend, but should remain silent (the chickens), even the CEO.</p> <p>An exception is management (e.g., CEO) feedback on survival points or explanation of the business relevance of the team's work.</p> <p>The Scrum needs to be a vehicle for communicating the product vision and organization goals.</p> <p>From this story: A pig and chicken discussed the name of their new restaurant. The chicken suggested Ham n' Eggs. "No thanks," said the pig, "I'd be committed, but you'd only be involved!"</p>

# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Teams of seven	Scrum can scale to large projects, but recommends one team have a maximum of seven members. Larger projects are multi-team. Scrum of scrums if large.
Common room (preferred)	Ideally, the team work together in a common project room, rather than separate offices or cubes. Separate, private space is still available for other activities. However, teams composed of geographically spread members, participating by speakerphone in the Daily Scrum, have reported success.
Daily build	At least one daily integration and regression test across all checked-in code for the project. The XP practice of Continuous Integration is even better.

# Workproducts, Roles, and Practices

## Core Practices:

Practice	Description
Sprint review	<p>At the end of each iteration, there is a review meeting (maximum of four hours) hosted by the Scrum Master. The team, Product Owner, and other stakeholders attend. There is a demo of the product. Goals include informing stakeholders of the system functions, design, strengths, weaknesses, effort of the team, and future trouble spots.</p> <p>Feedback and brainstorming on future directions is encouraged, but no commitments are made during the meeting. Later at the next Sprint Planning meeting, stakeholders and the team make commitments.</p> <p>“Power Point” presentations are forbidden. Preparation emphasis is on showing the product.</p>

# Workproducts, Roles, and Practices

## The Scrum Meeting: Details

The **Scrum Meeting** – or **scrum** – is the **heartbeat** of Scrum and the project.

Each **workday at the same time and place**, hold a meeting with the team members standing in a circle, at which time the **same special questions are answered by each member**:

1. What have you done since the last Scrum?
2. What will you do between now and the next Scrum?
3. What is getting in the way (blocks) of meeting the iteration goals?

# Workproducts, Roles, and Practices

## The Scrum Meeting: Details

Larman (2004-2009) added two more questions that have been useful:

4. Any tasks to add to the Sprint Backlog? (missed tasks, not new requirements)
5. Have you learned or decided anything new, of relevance to some of the team members? (technical, requirements, ...)

Continuously improving and learning – vital to agile development

No other discussion is allowed beyond the three (or five) questions.

- If other issues need discussion, secondary meetings immediately after the Scrum Meeting occur, usually with subsets of the team.  
“We need to talk about that. Let's meet after the Scrum.”

# Workproducts, Roles, and Practices

## The Scrum Meeting: Details

- The **Scrum Meeting** provides a daily forum to **update tasks**, and **surface and remove impediments**.
- The meeting is ideally held **in a stand-up circle** to encourage **brevity**.
- On average, **15 or 20 minutes** for **7–10 people**.
  - Longer meetings are common near the start of an iteration.
- Non-team members (**chickens**) are **outside the circle**.
- It is held next to a **whiteboard** at which all the **tasks** and **blocks** are written when reported.
  - The Scrum Master erases blocks only once they've been removed.

# Workproducts, Roles, and Practices

## Workproducts

In addition to the workproducts (illustrated before), Scrum allows any other workproducts of value to the project.

For example, it can be combined with some UP practices, and one can create a Vision or Risk List, using UP terminology.

# Workproducts, Roles, and Practices

## Workproducts

### Sample Product Backlog (Figure 7.3)

	A	B	C	D	E	F
1	Product Backlog					
2						
3	Requirement	Num	Category	Status	Pri	Estimate
4	log credit payments to AR	17	feature	underway	5	2
5	process sale-simple cash scenario	232	use case	underway	5	60
6	slow credit payment approval	12	issue	not started	4	10
7	sales commission calculation	43	defect	complete	4	2
8	lay-away plan payments	321	enhance	not started	3	20
9	PDA sale capture	53	technology	not started	1	100
10	process sale-credit pmt scenario	235	use case	underway	5	30

Note that all conceivable items go in the **backlog** and are prioritized by the Product Owner. The **estimates** (in person-hours of effort) start as rough guidelines, refined once the team commits to an item.

# Workproducts, Roles, and Practices

## Workproducts

### Sample Sprint Backlog (Figure 7.4)

	A	B	C	D	E	F	G	H	I
1	Sprint Backlog								
2	Task Description	Originator	Responsible	Status	Hours of work remaining				
3					6	7	8	9	10
4					362	322	317	317	306
5	Meet to discuss the goals and	JM	JM/SR	Completed	20	10	0	0	0
6	Move Calculations out of	TL	AW	Not Started	8	8	8	8	8
7	Get GEK Data		TN	Completed	12	0	0	0	0
8	Analyse GEK Data - Title		GP	In Progress	24	20	30	25	20
9	Analyse GEK Data - Parcel		TK	Completed	12	12	12	12	12
10	Define & build Database		BR/DS	In Progress	80	80	75	60	52

New estimates are allowed to increase above the original estimate.

The simplest (and thus preferred) tool is a spreadsheet; Sutherland uses a customized version of the open-source GNU GNATS tracking tool, with a Web interface.

Workers daily update the Sprint Backlog - Individuals are responsible for daily updates estimating the time remaining for their tasks.

Note the daily estimate of work remaining for each task; these columns also show the date (e.g., 6 of Jan) and total hours remaining on each day (e.g., Jan 6, 362 hours).

It is updated daily by the responsible members or by a daily tracker who visits each member.

# Workproducts, Roles, and Practices

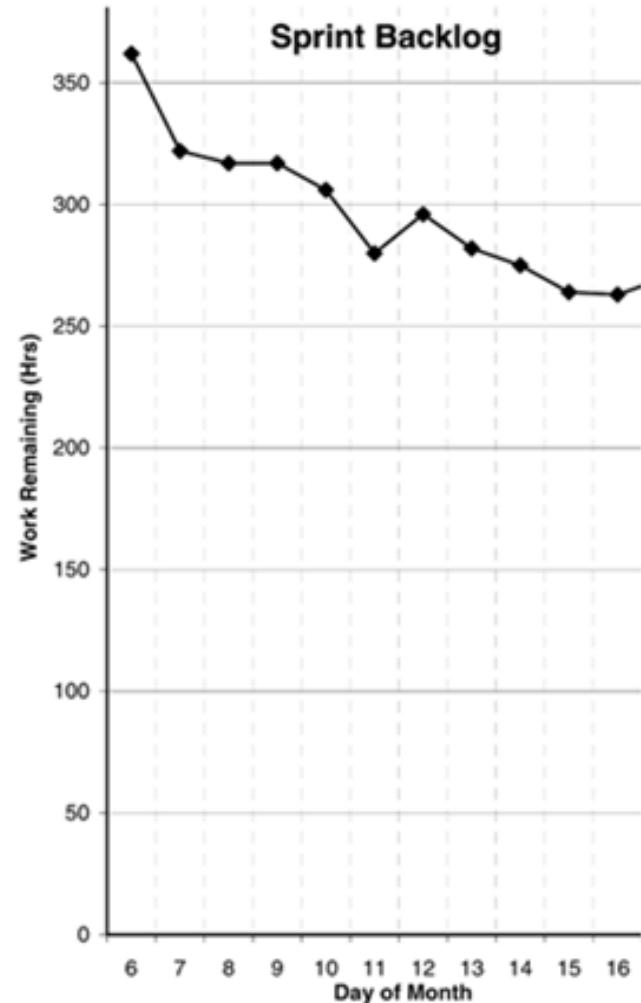
## Workproducts

Figure 7.5. sample Backlog Graph

It is a visual summary of estimated task hours remaining in the Sprint Backlog.

In Scrum, this is considered the most critical project data to track.

Recommended: Post an updated version of this each day on the wall by the Scrum meeting.



# Workproducts, Roles, and Practices

## Other Practices and Values

- Scrum Master reinforces **vision**
  - She needs to daily share and clarify the overall project vision, and goals of the Sprint, perhaps at the start of the Scrum meeting.
- Replace **ineffective Scrum Master**
  - The manager/Scrum Master is the servant of the developers, not vice versa.
  - If Scrum Master is **not removing blocks promptly**, acting as a firewall, and providing resources, the Scrum founders encourage replacing the Scrum Master.

# Values

## Openness

- The openly accessible **Product Backlog** makes visible the work and priorities.
- The **Daily Scrums** make visible the overall and individual status and commitments.
- **Work trend** and **velocity** are made visible with the **Backlog Graph**.

# Values

## Respect

- Or, **team responsibility** rather than **scapegoating**.
- The individual members on a team are respected for their **different strengths and weaknesses**, and not singled out for iteration failures.
- The **whole team** rather than a manager, **through self-organization and direction**, adopts **the attitude of solving “individual” problems through group exploration of solutions**, and is given the authority and resources to react to challenges, such as **hiring a specialist consultant to compensate for missing expertise**.

# Values

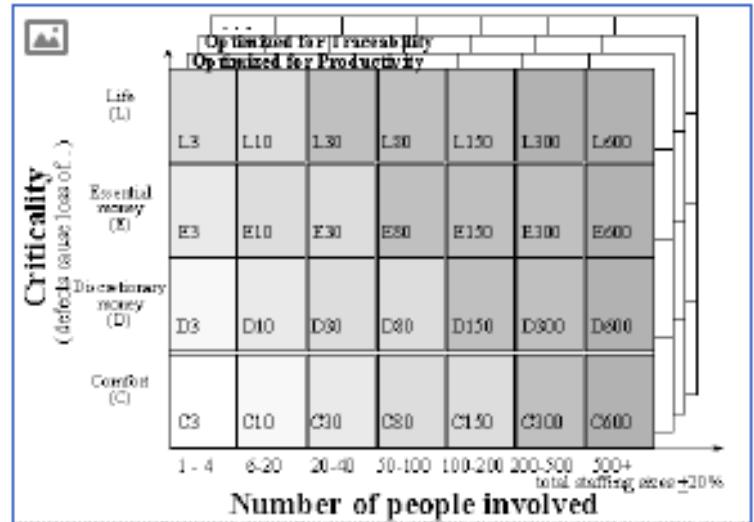
## Courage

- Management has the courage to plan and guide adaptively and to trust individuals and the team by avoiding telling them how to get the iteration done.
- The team has the courage to take responsibility for self-direction and self-management.

# Sample Projects

The following projects had significant Scrum influence:

- Large— IDX Web-enabled benefits suite
  - One year, 330 people across multiple related projects, an E300 project on the Cockburn scale, [SB02]
- Medium— Caremark
  - Four months, 20 people, an E20 project, [SB02]
- Small— Individual Personal NewsPage
  - One month, eight people, a C20 project [SB02]



"Criticality" is defined by the sentence "A defect could cause loss of":

# Process Mixtures

## Scrum + XP

The **Scrum** practice of a demo to external stakeholders at the end of each iteration enhances XP's feedback and communication goals.

- The **Scrum Backlog** and **progress tracking** approaches are **minor variations of XP** practices, and so simple that they are well within the **XP** spirit of “**do the simplest thing that could possibly work.**”

# Process Mixtures

## Scrum + XP

Scrum's 30-day timeboxed iteration length is not completely consistent with XP, which prefers shorter – even one-week – iterations.

# History

The roots of **Scrum** are found in a well-known article summarizing **common best practices** in **10 innovative Japanese companies**, “The New New Product Development Game,” Harvard Business Review, Jan 1986, by Takeuchi and Nonaka.

It introduced the terms **Sashimi (slices)** for IID, and **Scrum for the adaptive and self-directed team practices**. The name was taken from the game of **rugby**, for the **adaptive team behavior** moving a ball up the field.

# History

**Jeff Sutherland** is one of the Scrum creators and was VP at Easel Corporation in 1994 when he introduced some of its practices.

- He was influenced by a report on a hyper-productive project at Borland Corporation that effectively used **structured daily meetings** [Coplien94].
- In 1995 **Ken Schwaber** worked with **Sutherland** at Easel on the formalization of Scrum.
- Their results were described in a workshop paper [Schwaber95].
- In 1996 **Sutherland** joined Individual Inc., and asked **Ken Schwaber** to assist in the adoption of Scrum ideas.
- **Schwaber** refined and extended Scrum, in collaboration with **Sutherland**, into the versions ultimately described in [BDSSS98] and [SB02].

# References

- Craig Larman. Agile & Iterative Development: A Manager's Guide. Addison-Wesley, Pearson Education, 2004. (11th Printing, Aug. 2009) (ISBN-10: 0-13-111155-8, ISBN-13: 978-0-13-111155-4)
- Agile Alliance: <http://www.agilealliance.org/>
  - <http://www.agilemanifesto.org/>
  - <http://www.agilemanifesto.org/principles.html>
  - <http://guide.agilealliance.org/subway.html>
- Agile Modeling and RUP – Scott Ambler: <http://www.agilemodeling.com/>
- Agile, XP Method – Martin Fowler: <http://www.martinfowler.com/>
- Agile, XP Method – Robert Martin: <http://www.objectmentor.com/>
- Extreme Programming – Don Well: <http://www.extremeprogramming.org/>
- Extreme Programming – Ron Jeffries: <http://xprogramming.com/index.php>
- Agile Development – Artern Marchenko :  
<http://agilesoftwaredevelopment.com/>



---

COLLEGE OF ENGINEERING  
AND COMPUTER SCIENCE

# Advanced Software Process

## The Agile Process

### *XP*

Dave Garcia-Gomez  
Faculty / Lecturer  
Department of Computer Science

# Contents

- Extreme Programming (XP)
- Values
- Core Practices
- XP Lifecycle
- Work-products
- Roles
- Practices

# Extreme Programming (XP)

## Values

Extreme Programming (XP) is a well-known agile method.

- XP emphasizes:
  - Collaboration
  - Quick and early software creation
  - Skillful development practices
- XP is founded on four values:
  - Communication
  - Simplicity
  - Feedback
  - Courage

# Extreme Programming (XP)

## Core Practices

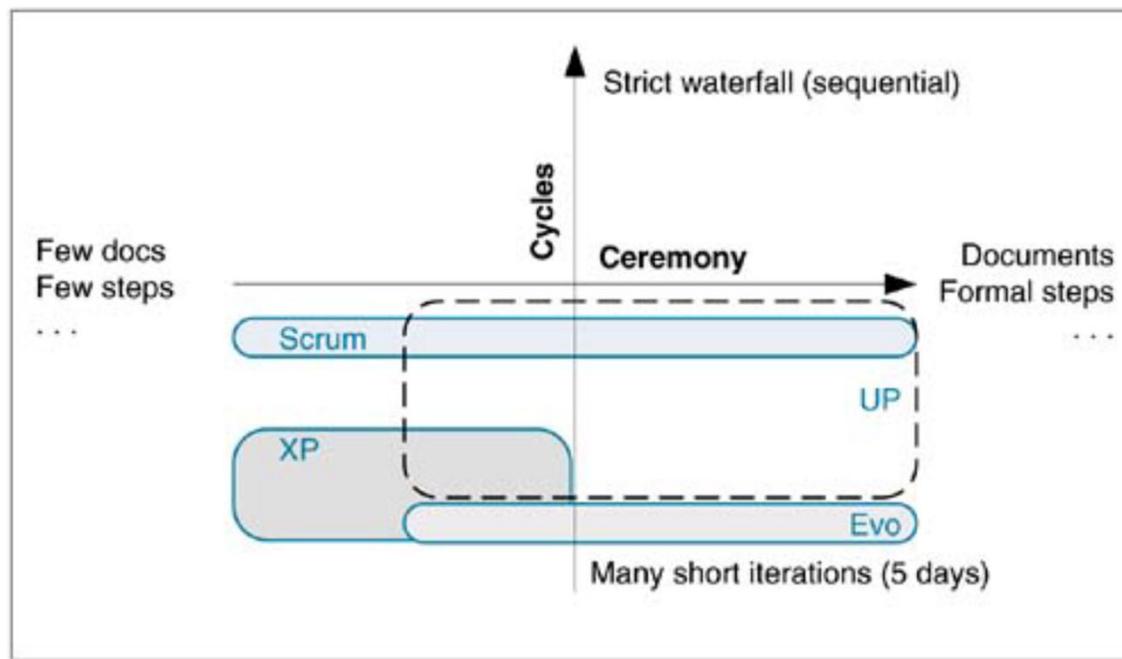
In addition to IID, XP recommends **12 core practices**:

- 1. Planning game
- 2. Small, frequent releases
- 3. System metaphors
- 4. Simple design
- 5. Testing
- 6. Frequent factoring
- 7. Pair programming
- 8. Team code ownership
- 9. Continuous integration
- 10. Sustainable pace
- 11. Whole team together
- 12. Coding standards

# Extreme Programming (XP)

## Classification

XP on the cycle and ceremony scale.



XP is low on the ceremony scale; it has only a small set of predefined, informal workproducts, such as paper index cards for summarizing feature requests, called story cards.

For average projects, the recommended length of a timeboxed iteration is between one and three weeks – somewhat shorter than for UP or Scrum.

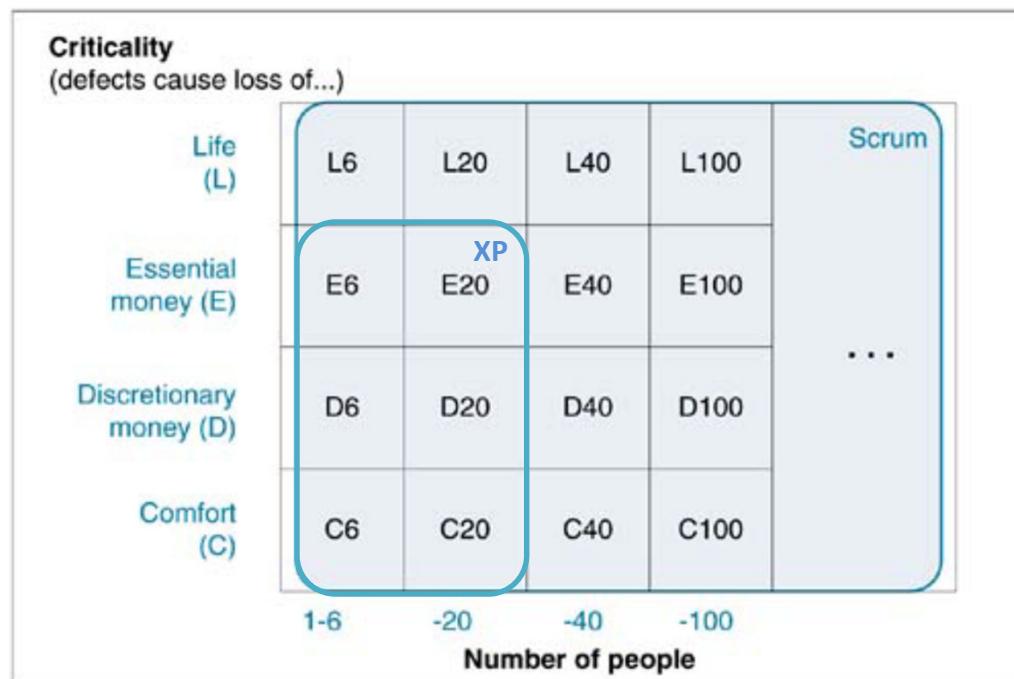
- Scrum: 30 days (4 w)
- XP: 1 – 3 weeks
- UP: 2 – 6 weeks
- Evo: 1 – 2 weeks

# Extreme Programming (XP)

## Classification

In terms of scope on the Cockburn scale, XP covers the cells shown in the following figure.

## XP on the Cockburn scale



A method selector = a function of (criticality, size, priority): how much formal process (ceremony, cycles) a software project requires: Cockburn scale (criticality, size)

- L: Loss of Life
- E: Loss of Essential Money
- D: Loss of Discretionary Money
- C: Loss of Comfort

Reference:

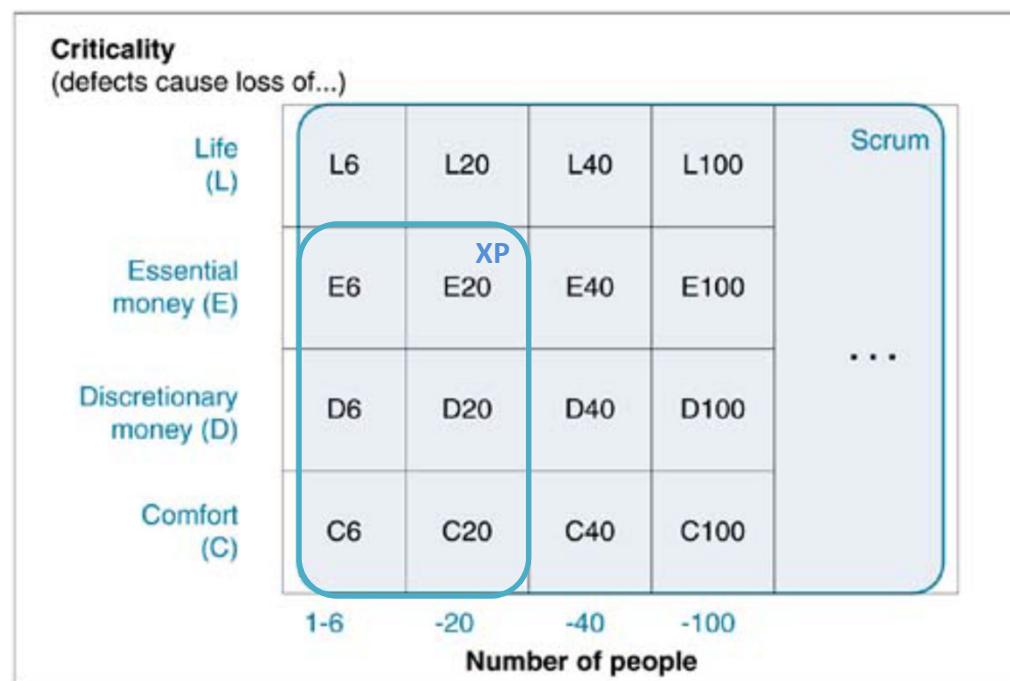
<http://alistair.cockburn.us/Cockburn+Scale>

# Extreme Programming (XP)

## Classification

In terms of scope on the Cockburn scale, XP covers the cells shown in the following figure.

### XP on the Cockburn scale



A refreshing quality of the original XP description was the statement of **known applicability**: It had been proven on projects involving roughly **10 developers or fewer**, and **not proven for safety-critical systems**. Nevertheless, it has been more recently applied with larger teams.

# Extreme Programming (XP)

- XP, created by Kent Beck [Beck 00], is an IID method that stresses customer satisfaction through:
  - rapid creation of high-value software
  - skillful and sustainable software development techniques
  - flexible response to change
- It is aimed at relatively small team projects, usually with delivery dates under one year.
- Iterations are short – usually one to three weeks.

# Extreme Programming (XP)

As the word ‘**programming**’ suggests, it provides **explicit methods for programmers**, so they can more confidently respond to changing requirements, even late in the project, and still produce quality code.

These include **test-driven development**, **refactoring**, **pair programming**, and **continuous integration**, among others.

In contrast to most methods, some XP practices truly are adopted by developers - they sense its **practical programmer-relevant techniques**.

# Extreme Programming (XP)

XP is very **communication** and **team-oriented**:

- Customers, developers, and managers form a **team working in a common project room** to quickly deliver software with high business value.

# XP Lifecycle

## XP Lifecycle

- Exploration Phase
- Planning Phase
- Iterations to Release Phase
- Productionizing Phase
- Maintenance Phase

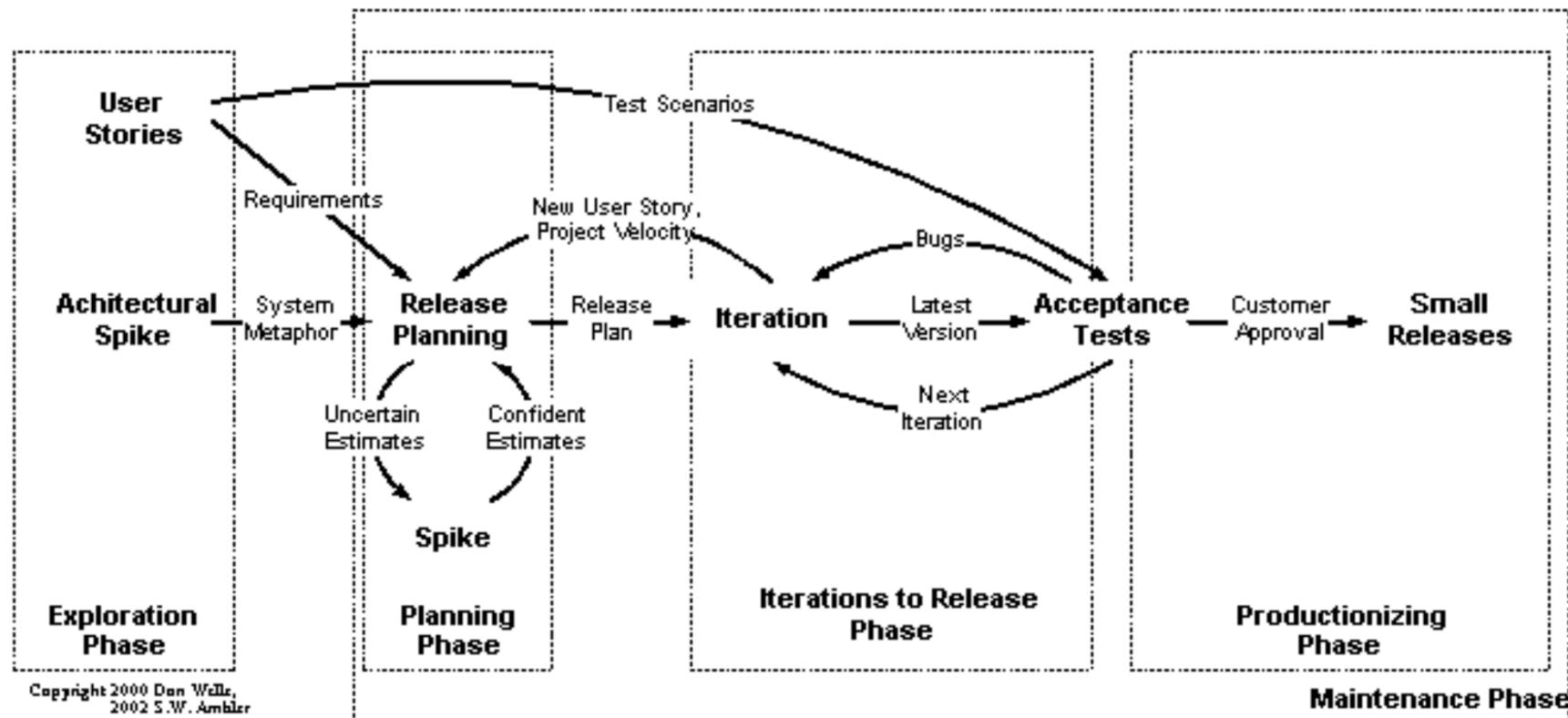
# XP Lifecycle

## XP Lifecycle

EXPLORATION	PLANNING	ITERATIONS TO FIRST RELEASE	PRODUCTIONIZING	MAINTENANCE
<b>Purpose:</b> <ul style="list-style-type: none"><li>- Enough well-estimated story cards for first release.</li><li>- Feasibility ensured.</li></ul>	<b>Purpose:</b> <ul style="list-style-type: none"><li>- Agree on date and stories for first release.</li></ul>	<b>Purpose:</b> <ul style="list-style-type: none"><li>- Implement a tested system ready for release.</li></ul>	<b>Purpose:</b> <ul style="list-style-type: none"><li>- Operational deployment</li></ul>	<b>Purpose:</b> <ul style="list-style-type: none"><li>- Enhance, fix.</li><li>- Build major releases</li></ul>
<b>Activities:</b> <ul style="list-style-type: none"><li>- prototypes</li><li>- exploratory proof of technology programming</li><li>- story card writing and estimating</li></ul>	<b>Activities:</b> <ul style="list-style-type: none"><li>- Release Planning Game</li><li>- story card writing and estimating</li></ul>	<b>Activities:</b> <ul style="list-style-type: none"><li>- testing and programming</li><li>- Iteration Planning Game</li><li>- task writing and estimating</li></ul>	<b>Activities:</b> <ul style="list-style-type: none"><li>- documentation</li><li>- training</li><li>- marketing</li><li>- ...</li></ul>	<b>Activities:</b> <ul style="list-style-type: none"><li>- May include these phases again, for incremental releases.</li></ul>

# XP Lifecycle

## XP Project Lifecycle



# XP Lifecycle

## XP Project Lifecycle Definitions

**Metaphor**: a conceptual framework; a logical architecture of the system; a shared story or description of how the system works.; a description of how you intend to build your system. The metaphor is defined during an architectural spike early in the project (during the 1st iteration or pre-iteration).

**Architectural spike**: intends to identify areas of high risk, to get started with estimating them correctly.

**Spike**: end-to-end, but very thin; depth-first, from the top to the bottom, then closed-loop.

**Project velocity**: a measure of project progress.

# XP Lifecycle

1. Like many projects, XP can start with **exploration**. Some story cards (features) may be written, with rough estimates.
2. In the **Release Planning Game**, the customers and developers **complete the story cards and rough estimates**, and then **decide what to do for the next release**.

# XP Lifecycle

3. For the next iteration, in the **Iteration Planning Game**, customers **pick stories to implement**. They choose stories – and thus **steer the project** – based on current status, and their latest priorities for the release.
  - Developers then break the stories into many **short, estimated tasks**.

# XP Lifecycle

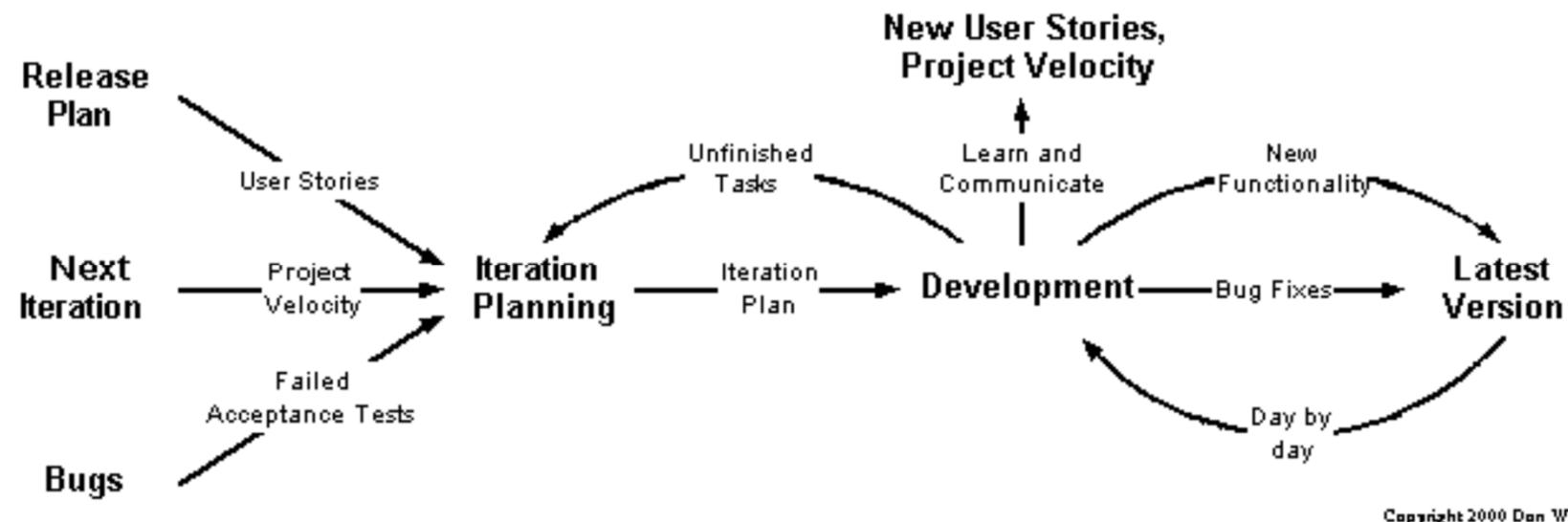
- Finally, a **review** of the **total estimated task-level effort** may lead to **readjustment** of the chosen stories, as XP **does not** allow overworking the developers with more than they can do based on “family-friendly” work days, such as an eight-hour day.
- Overtime is seriously discouraged in XP; it is viewed as a **sign of a dysfunctional project, increasingly unhappy people, and dropping productivity and quality.**

# XP Lifecycle

4. Developers implement the stories within the agreed timeboxed period, continually collaborating with customers (in the common project room) on tests and requirement details.
5. If not finished for release, return to step 3 for the next iteration.

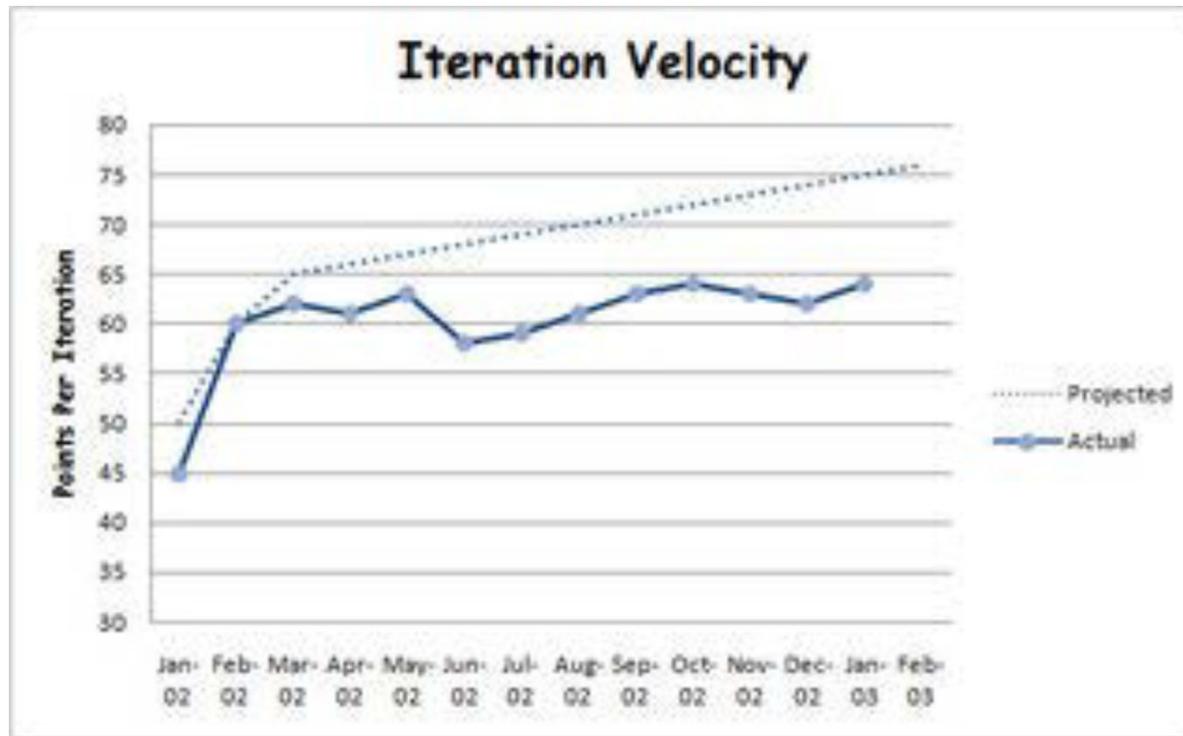
# XP Lifecycle

## The XP Iteration Lifecycle



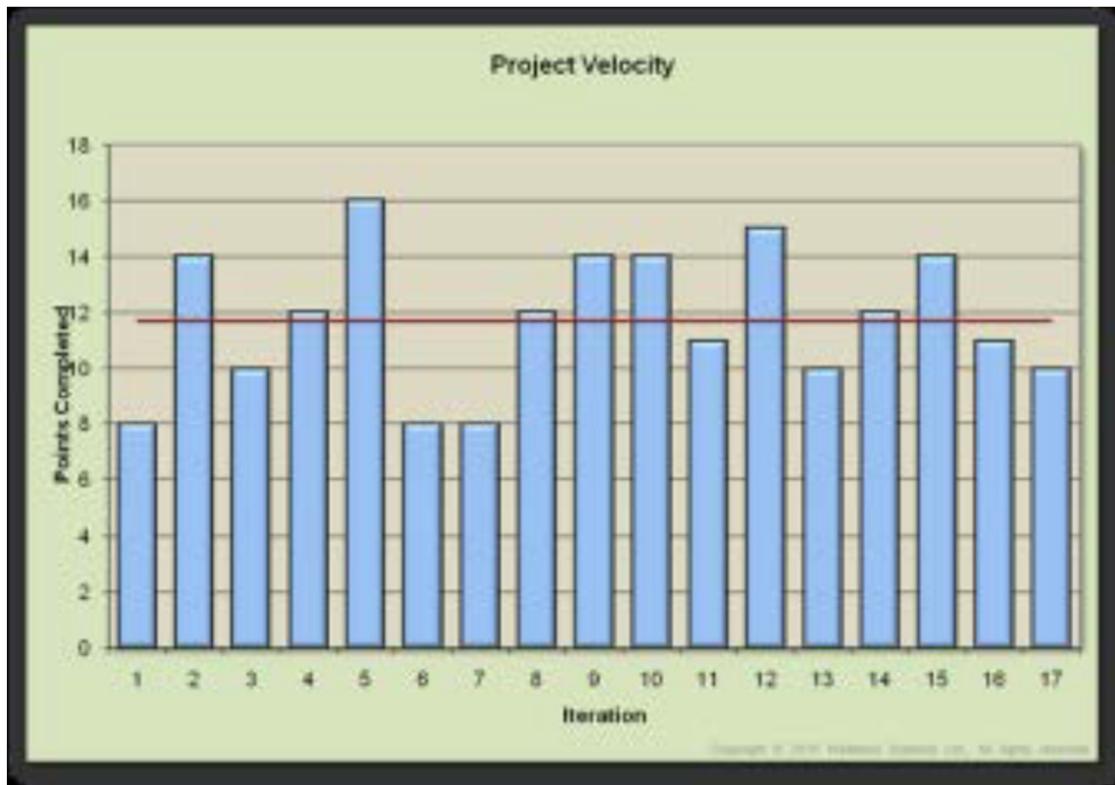
# XP Lifecycle

## The XP Iteration Lifecycle



# XP Lifecycle

## The XP Iteration Lifecycle



**Project velocity:** a measure showing how much move fast, and to which direction the project is going; not simply measure how much code is written, but measure the value from the customer's view; how much user stories are completed during an iteration and estimation; the ratio between estimated time and real calendar time.

# XP Lifecycle

## The XP Iteration Lifecycle

Story	Estimate (Points)	Completed?
Customer Searches for Book by Author	2	Yes
Customer Searches for Book by Category	3	Yes
Customer Searches for Book by Title	2	Yes
Customer Searches for Book by ISBN	1	Yes
Preferred Customer receives Discount	2	No

# XP Lifecycle

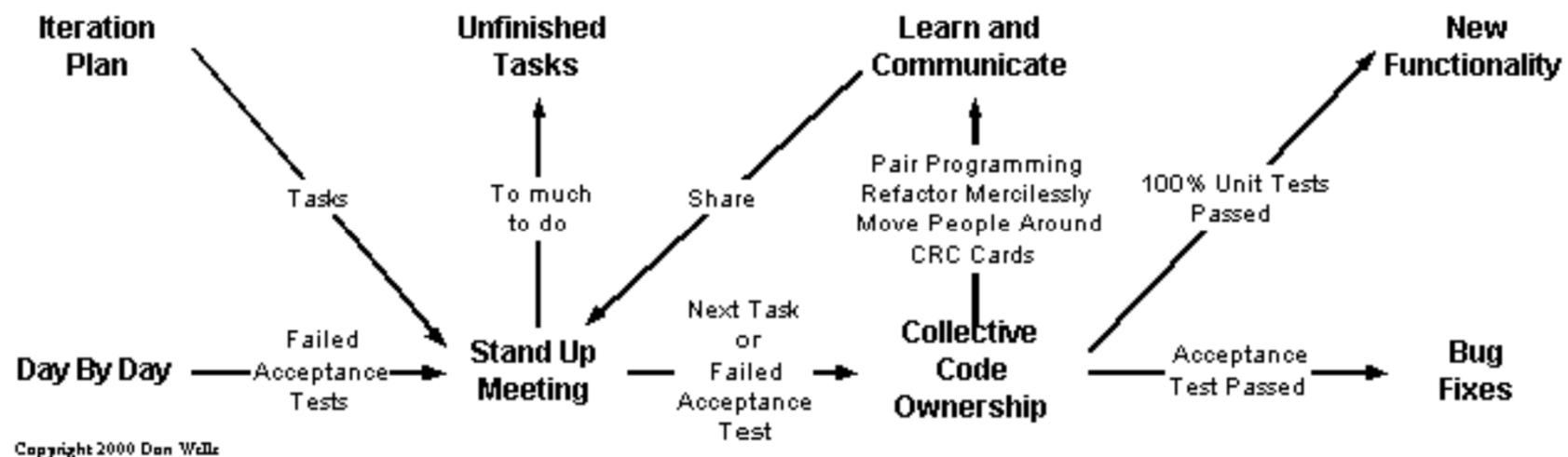
## The XP Iteration Lifecycle



**Project velocity:** a measure showing how much move fast, and to which direction the project is going; not simply measure how much code is written, but measure the value from the customer's view; how much user stories are completed during an iteration and estimation; the ratio between estimated time and real calendar time.

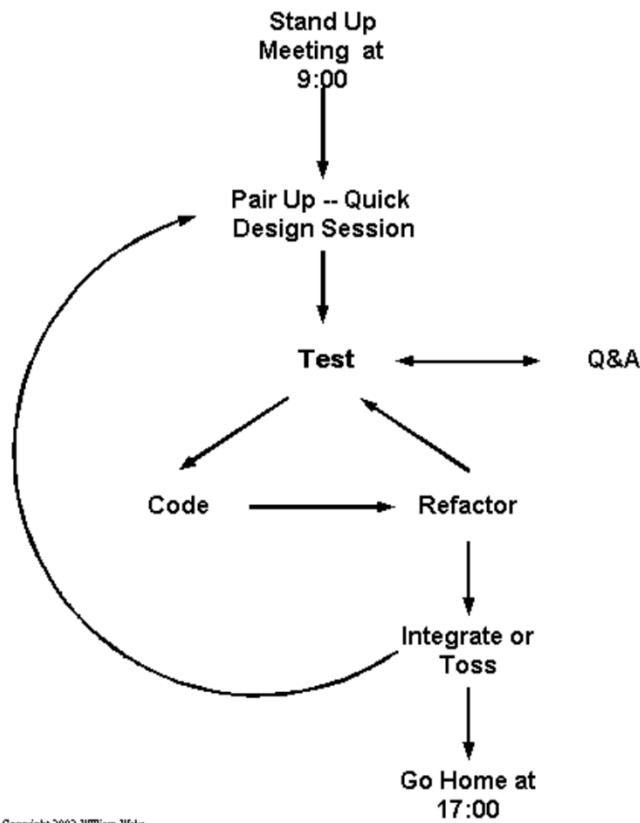
# XP Lifecycle

## The XP Development Lifecycle



# XP Lifecycle

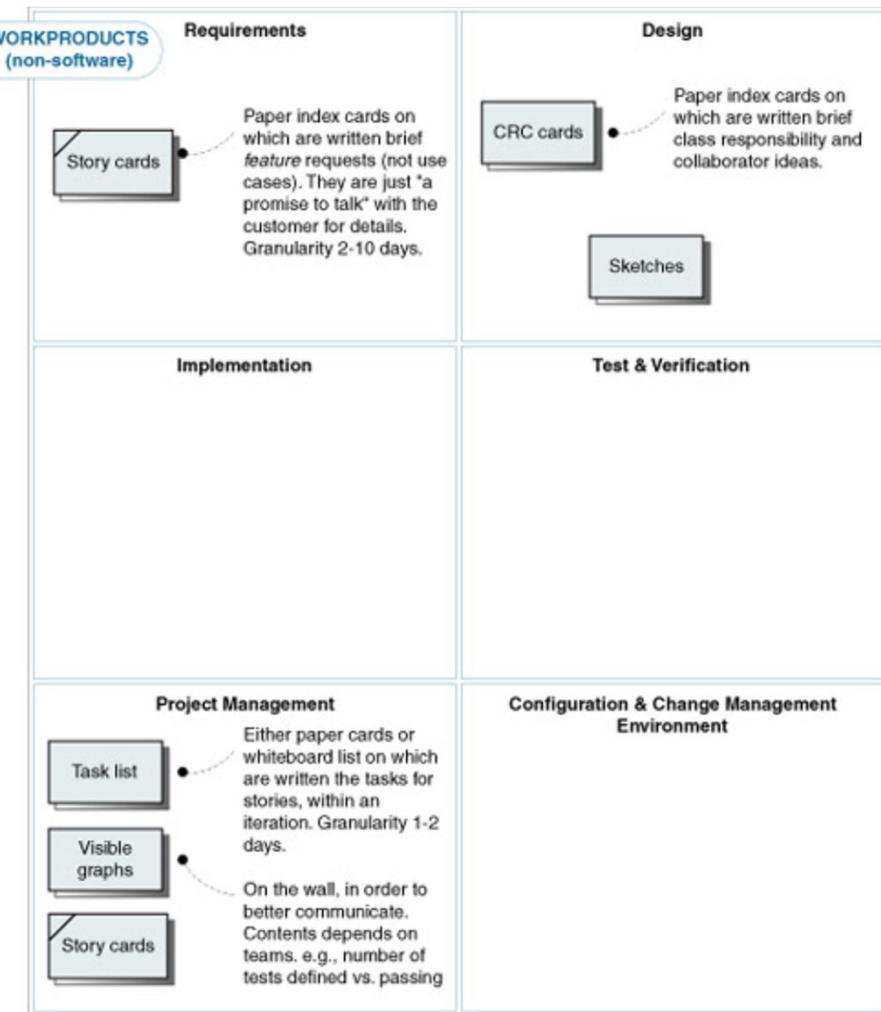
## Typical Day of an XP Developer



Pair programming is a programming technique in which two programmers work together at one computer. One programmer (driver) types in code, while the other programmer (observer, navigator, reviewer) reviews the code.

# Workproducts

# XP Lifecycle



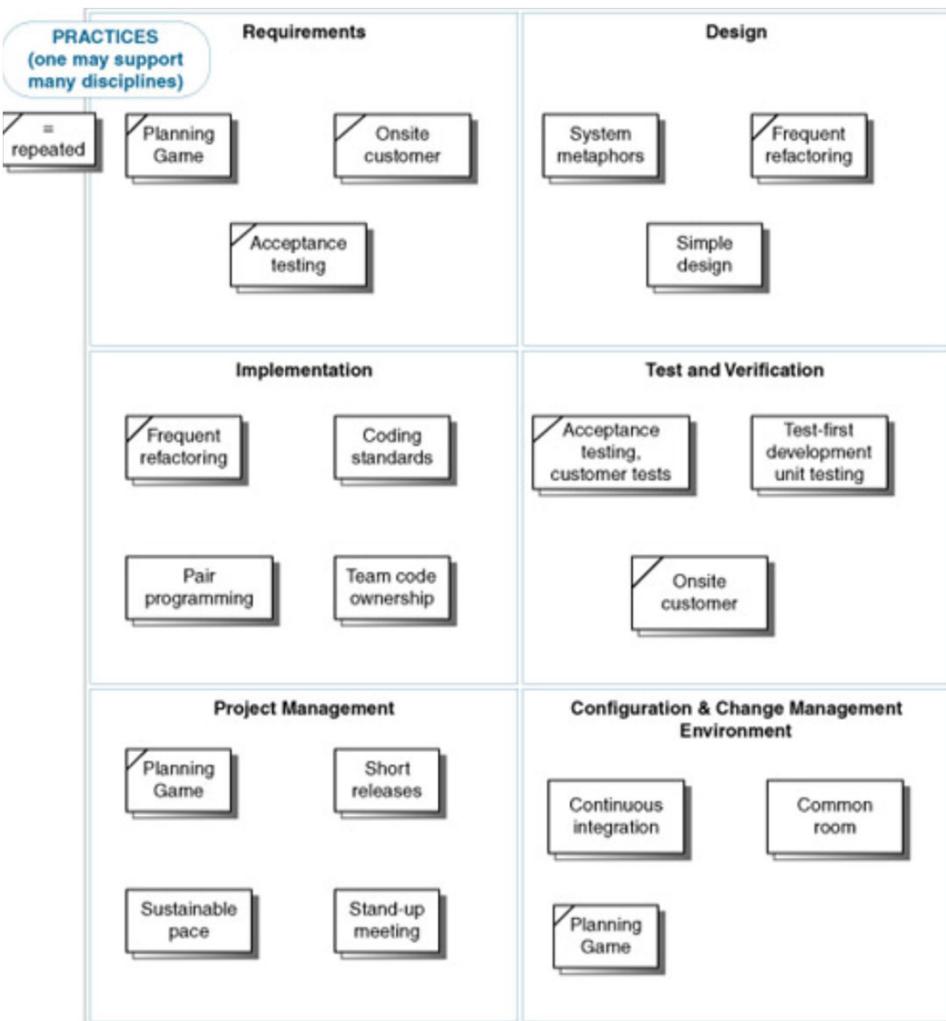
# XP Lifecycle

## Roles



# Practices

# XP Lifecycle



# Story Cards

- A **handwritten** note on a **paper index card**.
- During the **Planning Game**, many of these are written.
- Sample story card
  - Find available classes, upload a course material
- The **story cards record user stories: features, fixes, or nonfunctional requirements** that the user wants.
- **Stories** are usually in the **one-day to three-week** range of estimated duration.

# Stories, Use Cases & Features

- Contrary to some misunderstanding, XP stories are not use cases or scenarios.
- They usually represent features.
- Note that XP prefers a feature-driven approach to describing requirements rather than the use-case-driven approach that UP promotes.

# Stories, Use Cases & Features

- A **feature** is a service that the system provides to fulfill customers' needs.
- A **use case** describes a series of interactions between users and the system to accomplish their goals. A use case describes how users and the system interact to realize the identified feature.

# Stories and Communication

In XP, oral communication is preferred, and the story card purpose is not to detail the user story, but to jot a summary, make references to other documents, and in general, to view the card as “a promise to talk” (in Alistair Cockburn's words) with the customer who wrote it, by the developers implementing it.

- Since whole team together is an XP practice, the card donor should be readily available.

# Stories and Communication

XP coaches vary on their advice regarding granularity for estimation.

- Some say stories can be in the two-day to two-week range of effort.
- Others recommend stories be estimated in units of one, two, or three weeks, but not in finer person-day units.

# Stories for Estimation

XP coaches vary on their advice regarding granularity for estimation.

- Some say stories can be in the two-day to two-week range of effort.
- Others recommend stories be estimated in units of one, two, or three weeks, but not in finer person-day units.

# Embrace Change

- Onsite customer proxies
- Customer on call
- Embrace change
  - The overarching attitude that XP promotes is to embrace rather than fight change, in the requirements, design, and code, and be able to move quickly in response to change.

# Task List

- During an **Iteration Planning Game**, the team **convenes around a whiteboard**, and generates a **list of tasks** for all the stories chosen for the iteration.
- Another popular alternative is to **generate individual task cards**.
- Once a task is chosen by a **volunteer**, they enter an **effort estimate** (in ideal engineering hours) – tasks should be in the **1–2 day range**.

# Volunteering

- Only by volunteering (accepted responsibility)
- Tasks are not assigned to people.
- Rather, during the Iteration Planning Game, people choose or volunteer for tasks.
- This leads to a higher degree of commitment and satisfaction in the self-accepted responsibility.

# Light Modeling

## Very light modeling

- XP encourages programming very early and does take to “extreme” the avoidance of up-front design work.
- Any more than 10 or 20 minutes of design thinking (e.g., at the whiteboard with sketches or notes) before programming is considered excessive.
- Contrast this with Scrum or the UP, for example, where a half-day of design thought near the start of an iteration is acceptable.

# Minimal or “Just Enough” Documentation

- With the goal of getting to code fast, XP discourages writing unnecessary requirements, design, or management documents.
- The use of small paper **index cards** is preferred for **jotting brief descriptions**, as are verbal communication and elaboration.
- Note that the practice of “**avoiding documentation**” is compensated by the presence of an **onsite customer**.
- XP is **not anti-documentation**, but notes it has a cost, perhaps better spent on programming.

# Measurement Visible Graphs - Communication

- Metrics
  - XP recommends **daily measurement of progress** and **quality**. (Measure at least one thing.)
  - It **does not mandate** the exact metrics, but to “**use the simplest ones that could work.**”
    - Examples: numbers of completed tasks and stories, success rate of tests
- Visible wall graphs
  - The collected metrics are **daily updated on wall graphs** for all to **easily see - easily communicate.**

# Tracking

## Tracking and Daily Tracker

- The **regular collection of task and story progress metrics** is the responsibility of a **tracker**.
- This is done with a **walk-about to all the programmers**, rather than email.
  - Ron Jeffries (one of the XP founders) said, “*XP is about people, not computers.*”
- **Test metrics** can be **automatically collected by software**.

# Common Project Room

## Common project room

- XP projects are run in a **common project room** rather than separate offices.
  - **Pair programming tables** are in the center of the room, and the walls are clear for whiteboard and poster work.
  - Of course, people may have a private space for private time, but **production software development** is a **team sport** in XP.

## Daily stand-up meeting

- As in Scrum, there is a **daily short stand-up** (to keep it short) meeting of status.

# Agile Project Room

The project room walls are exposed (no furniture against them) and covered in whiteboards and static-cling-sheet whiteboard material.



# Ideal Engineering Hours (IEH)

## Ideal Engineering Hours (IEH)

- Task estimates – and possibly story estimates – are done in terms of IEH, or uninterrupted, dedicated, focused time to complete a task.

## Story estimates

- To estimate larger stories, some XP practitioners recommend using only coarse values of one, two, or three week durations rather than IEH or day-level estimates.

# Sample XP Projects

The following projects had significant XP influence:

- Large – Atlas leasing system
  - Three years, 60+ people, Java technologies, an E100 project [Schuh01]
- Medium – Orca security incident-response
  - One year, 25 people, a D40 project [Morales02]
- Small – C3 payroll
  - One year, 10+ people, an E20 project [C3Team98]

# Adoption Strategies

XP recommends adoption like this:

1. Pick the worst project or problem.
2. Apply XP until solved.
3. Repeat.

# Adoption Strategies

If all the XP practices **cannot be swallowed at once**, Beck recommends starting with:

- whole team together in a common project room
- test-first development
- acceptance tests written/owned by customers
- Planning Game
- pair programming

# History

- In the mid-1980s, Kent Beck and Ward Cunningham worked together in a research group at Tektronix.
  - They founded the idea of **CRC cards** and (the seminal contribution of) design patterns, while building many Smalltalk systems.
  - The roots of XP come from this collaboration.

Class:	
Responsibilities:	Collaborators:

# History

- In the C3 Project of Chrysler, Beck introduced the majority of practices that became XP, and brought in Ron Jeffries to daily lead and coach the team.
- Martin Fowler was also invited for some consulting.
- Primarily led by the vision of Beck, the XP practices coalesced on this project.
- Beck says that at its heart, XP is expressing what he learned with and from Cunningham.

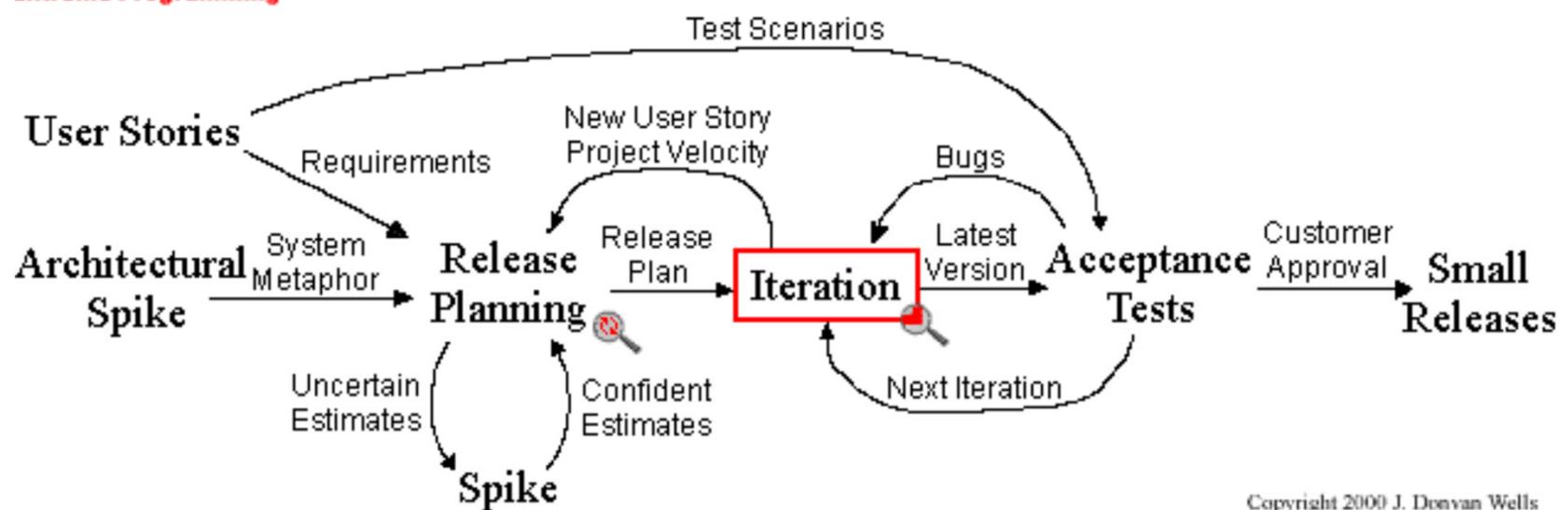
# Sample XP Practices

Don Wells: XP Rules

<http://www.extremeprogramming.org/rules.html>



## Extreme Programming Project



Copyright 2000 J. Donvan Wells

# Sample XP Practices

## Don Wells: XP Rules



### The Rules of Extreme Programming

#### Planning

- User stories are written.
- Release planning creates the release schedule.
- Make frequent small releases.
- The project is divided into iterations.
- Iteration planning starts each iteration.



#### Managing

- Give the team a dedicated open work space.
- Set a sustainable pace.
- A stand up meeting starts each day.
- The Project Velocity is measured.
- Move people around.
- Fix XP when it breaks.

#### Designing

- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

#### Coding

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Set up a dedicated integration computer.
- Use collective ownership.

#### Testing

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

Let's review the values of Extreme Programming (XP) next.

[ExtremeProgramming.org home](#) | [XP Map](#) | [XP Values](#) | [Test framework](#) | [About the Author](#)

Copyright 1999 Don Wells all rights reserved

# References

- Craig Larman. Agile & Iterative Development: A Manager's Guide. Addison-Wesley, Pearson Education, 2004. (11th Printing, Aug. 2009) (ISBN-10: 0-13-111155-8, ISBN-13: 978-0-13-111155-4)
- Agile Alliance: <http://www.agilealliance.org/>
  - <http://www.agilemanifesto.org/>
  - <http://www.agilemanifesto.org/principles.html>
  - <http://guide.agilealliance.org/subway.html>
- Agile Modeling and RUP – Scott Ambler: <http://www.agilemodeling.com/>
- Agile, XP Method – Martin Fowler: <http://www.martinfowler.com/>
- Agile, XP Method – Robert Martin: <http://www.objectmentor.com/>
- Extreme Programming – Don Well: <http://www.extremeprogramming.org/>
- Extreme Programming – Ron Jeffries: <http://xprogramming.com/index.php>
- Agile Development – Artern Marchenko :  
<http://agilesoftwaredevelopment.com/>