Integrating Requirements Engineering into Software Engineering Processes

Er. Rudra Nepal

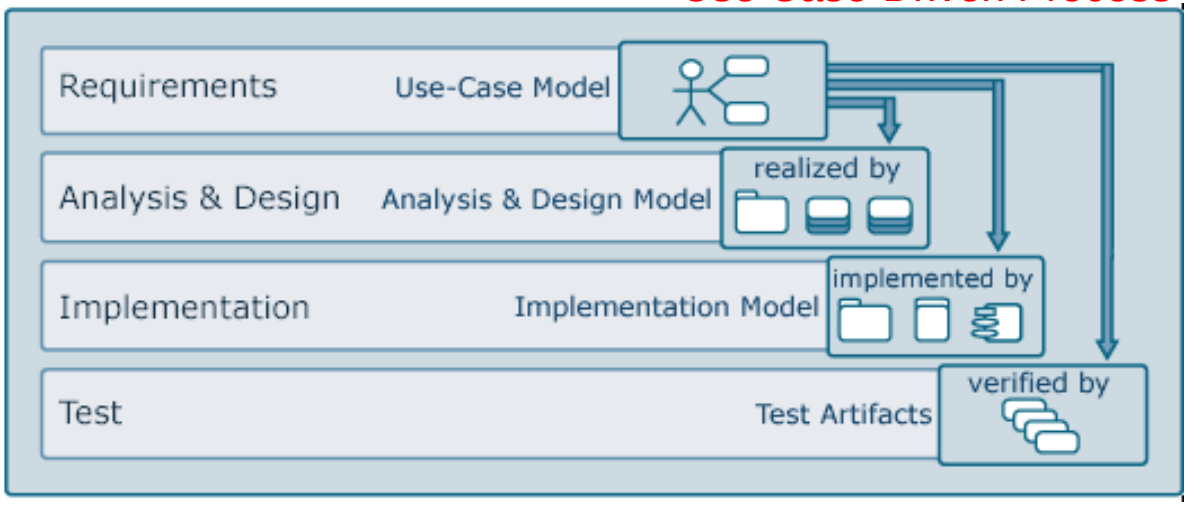
Table of Contents

Rational Unified Process

Rational Unified Process (RUP)

- One commercial implementation of the Unified Process
  - Developed by Jacobson, Booch, Rumbaugh
  - Evolved from the Objectory process and the earlier Ericsson approach
  - Now an IBM product[1]
- Vocabulary and concepts
  - Artefacts, roles, disciplines, activities
  - Use case-driven, architecture-centered, iterative, incremental, risk management-oriented
- RUP is a process framework (not a process on its own)
  - Intended to be customized to project needs

Use Case-Driven Process



[1] http://www-01.ibm.com/software/awdtools/rup/

RUP Vocabulary (1)

- Artefact
  - Data element produced during the development (document, diagram, report, source code, model…)
- Role
  - Set of skills and responsibilities
  - RUP defines 30 roles to be assigned (not all need to be fulfilled, depends on the project)
  - 4 role categories: analysts, developers, testers, and managers
- Activity
  - Small, definable, reusable task that can be allocated to a single role

RUP Vocabulary (2)

- Discipline
  - Set of activities resulting in a given set of artefacts
  - RUP includes 9 disciplines: engineering (business modeling, requirements, analysis and design, implementation, test, deployment) and support (configuration and change management, project management, environment)
  - Guidance for a discipline is provided as workflows: sequence of activities that produces a result of observable value

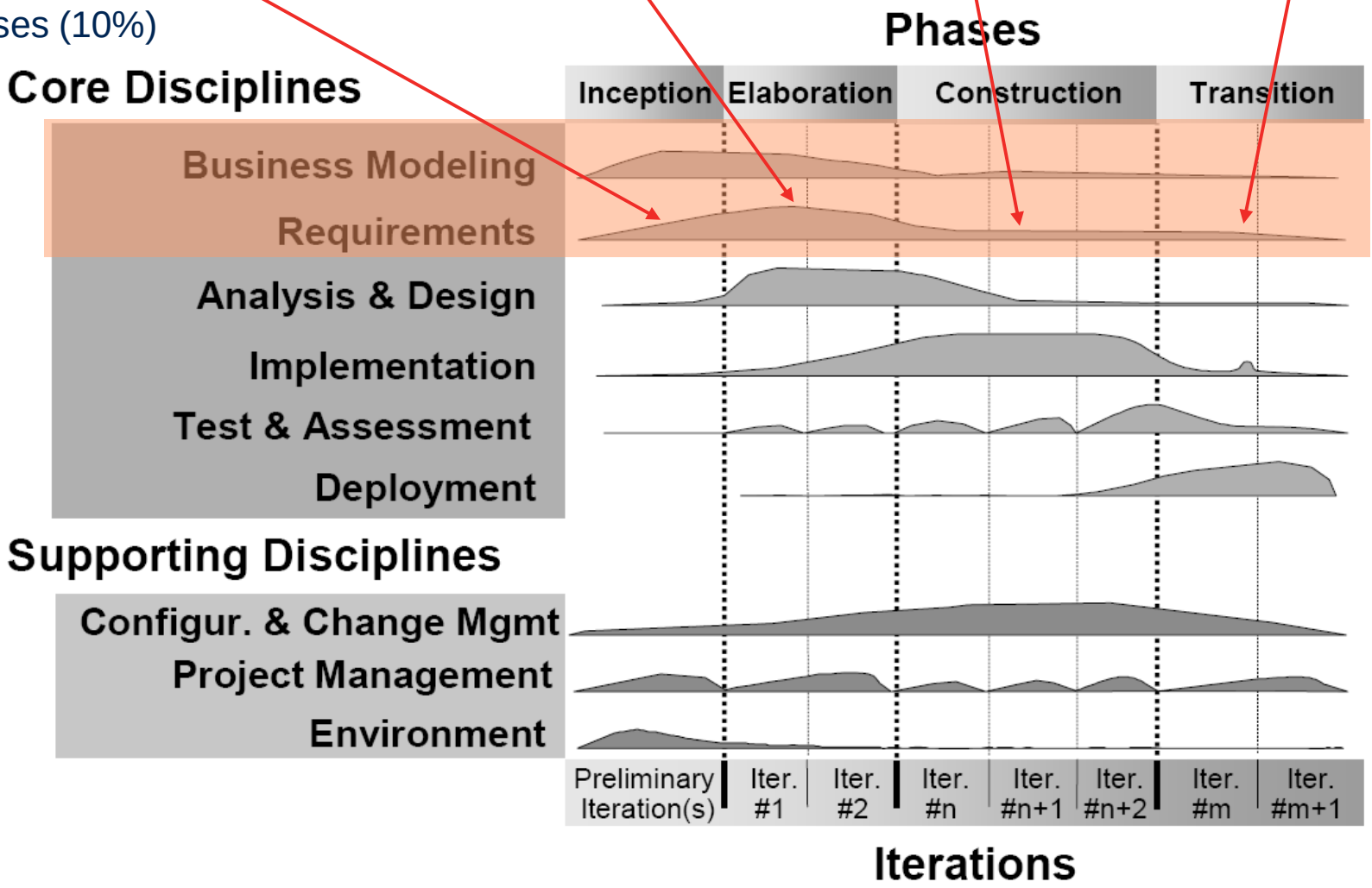
Disciplines, Phases, and Iterations

Identify most of the use cases to define scope, detail critical use cases (10%)

Detail the use cases (80% of the requirements)

Identify and detail remaining use cases

Track and capture requirements changes



Inception Phase

- Overriding goal is obtaining buy-in from all interested parties
  - Initial requirements capture
  - Cost-benefit analysis
  - Initial risk analysis
  - Project scope definition
  - Defining a candidate architecture
  - Development of a disposable prototype
  - Initial use case model (10%-20% complete)
  - First pass at a domain model

Elaboration Phase

- Requirements Analysis and Capture
  - Use Case Analysis
    - Use Cases (80% written and reviewed by end of phase)
    - Use Case Model (80% done)
    - Scenarios
      - Sequence and Collaboration Diagrams
      - Class, Activity, Component, State Diagrams
  - Glossary (so users and developers can speak common vocabulary)
  - Domain Model
    - To understand the problem: the system's requirements as they exist within the context of the problem domain
  - Risk Assessment Plan revised
  - Architecture Document

Construction Phase

- Focus is on implementation of the design
  - Cumulative increase in functionality
  - Greater depth of implementation (stubs fleshed out)
  - Greater stability begins to appear
  - Implement all details, not only those of central architectural value
  - Analysis continues, but design and coding predominate

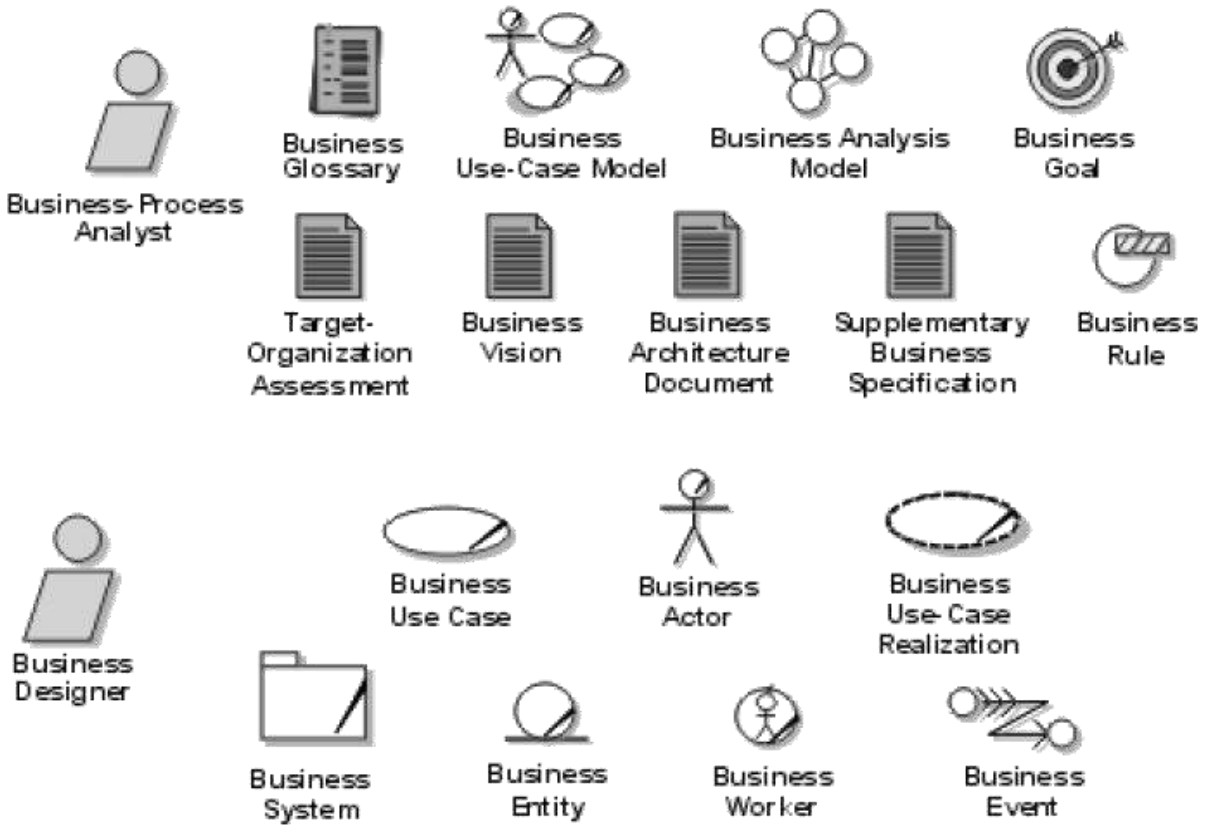
Transition Phase

- The transition phase consists of the transfer of the system to the user community
  - Includes manufacturing, shipping, installation, training, technical support, and maintenance
  - Development team begins to shrink
  - Control is moved to maintenance team
  - Alpha, Beta, and final releases
  - Software updates
  - Integration with existing systems (legacy, existing versions…)

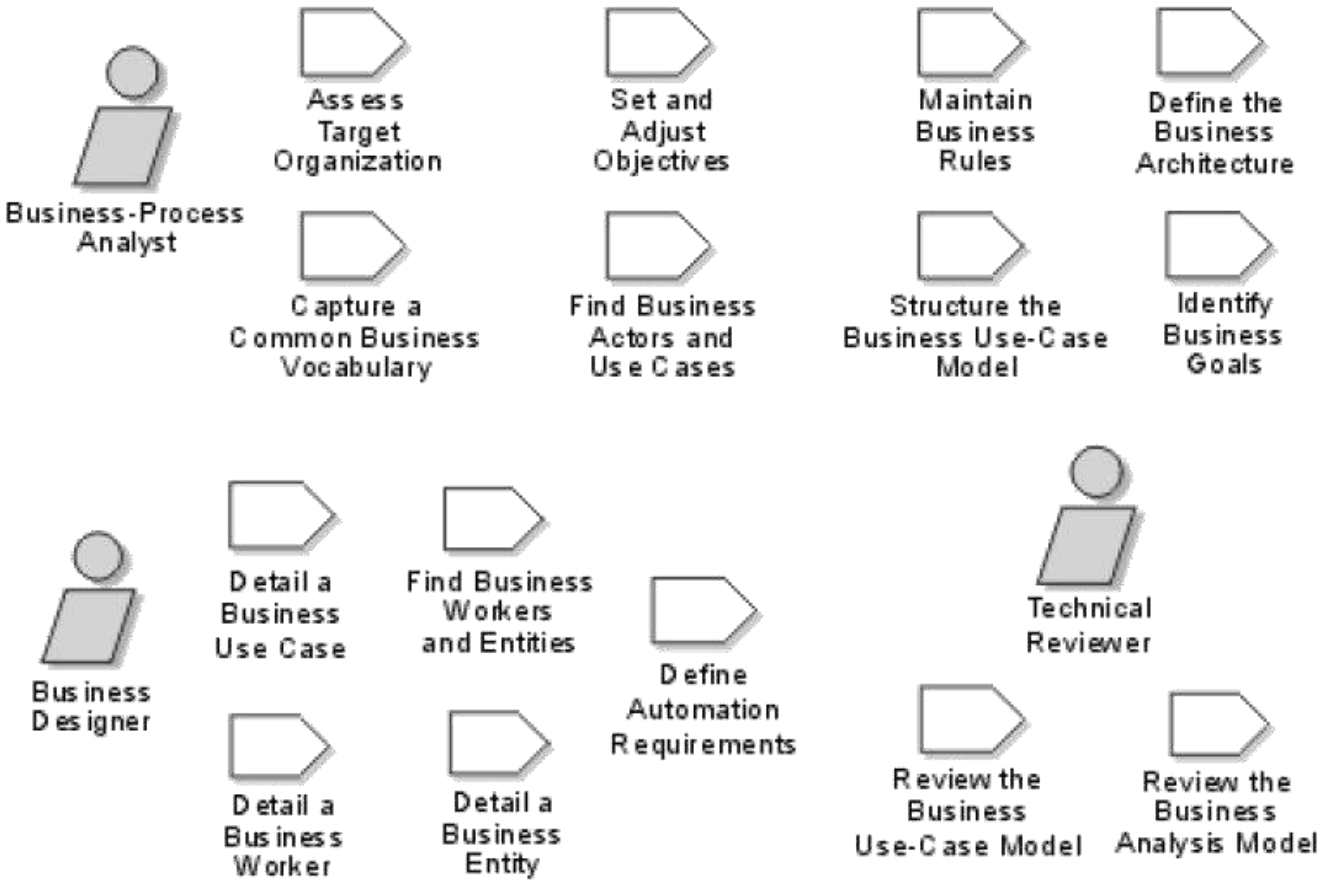
Business Modeling Discipline

- Objectives
  - Understand the structure and the dynamics of the organization in which a system is to be deployed (the target organization)
  - Understand current problems in the target organization and identify improvement potential
  - Ensure that customers, end users, and developers have a common understanding of the target organization
  - Derive the system requirements needed to support the target organization
- Explains how to describe a vision of the organization in which the system will be deployed and how to then use this vision as a basis to outline the process, roles, and responsibilities

Business Modeling Discipline – Artefacts



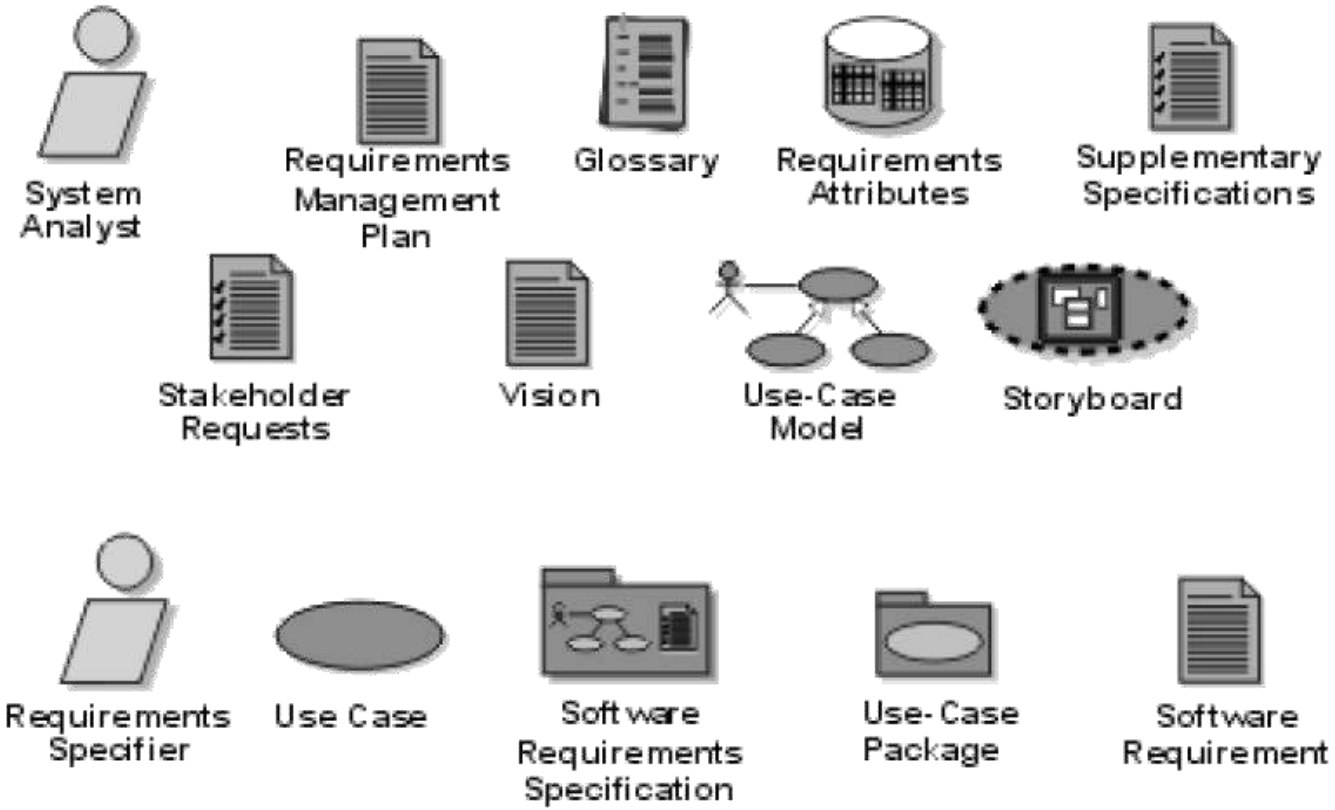
Business Modeling Discipline – Roles



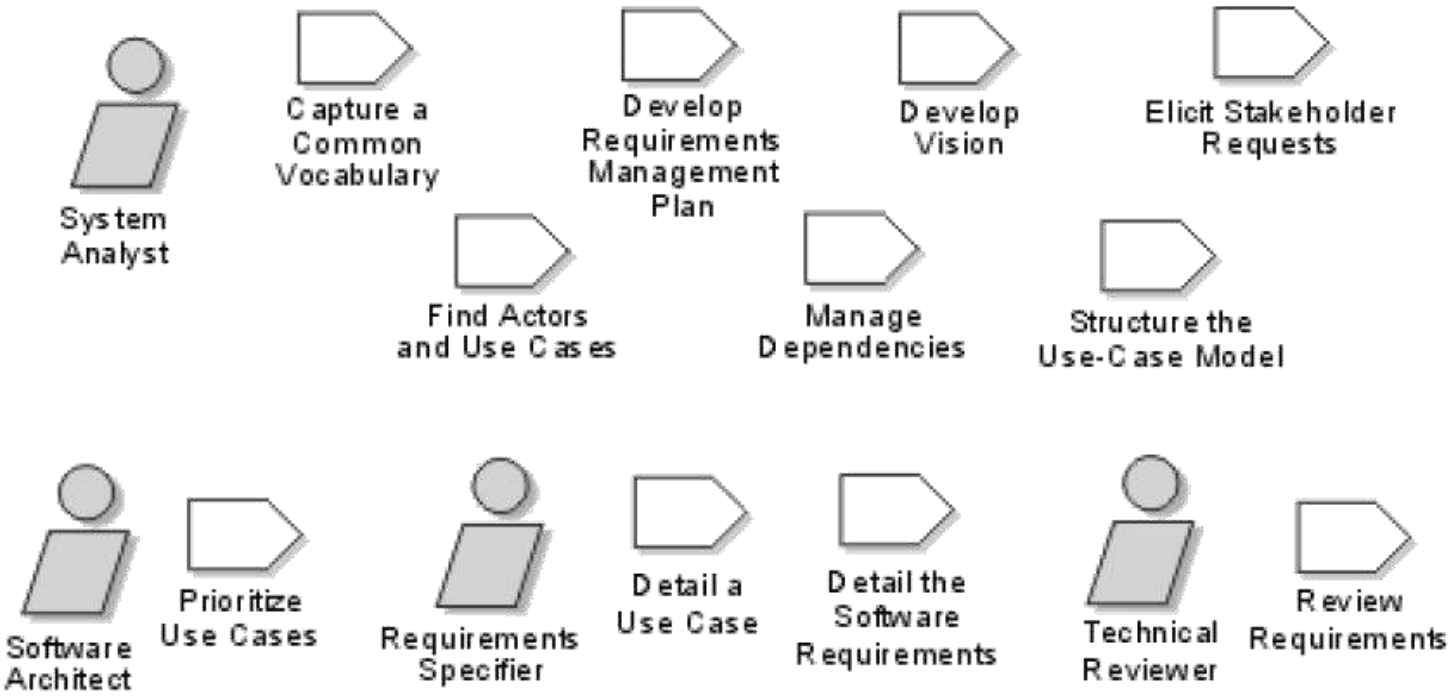
Requirements Discipline

- Establish and maintain agreement with the customers and other stakeholders on what the system should do
- Provide system developers with a better understanding of the system requirements
- Define the boundaries of the system
- Provide a basis for planning the technical contents of iterations
- Provide a basis for estimating the cost and time to develop the system

Requirements Discipline – Artefacts



Requirements Discipline – Roles



Requirements Discipline – Tasks

- Includes the following tasks
  - List candidate requirements
    - Candidate features that could become requirements
  - Understand system context
    - Based on business model, domain model or simple glossary
  - Capture functional requirements
    - Develop use cases and user interface support of use cases
  - Capture non-functional requirements
    - Tied to use cases or domain concepts
    - Defined as supplementary requirements
  - Validate requirements

Other Disciplines – Engineering (1)

- Analysis and Design Discipline
  - Transform the requirements into a design of the system-to-be
  - Evolve a robust architecture for the system
  - Adapt the design to match the implementation environment
- Implementation Discipline
  - Define the organization of the implementation
  - Implement the design elements
  - Unit test the implementation
  - Integrate the results produced by individual implementers (or teams), resulting in an executable system

Other Disciplines – Engineering (2)

- Test Discipline
  - Find and document defects in software quality
  - Provide general advice about perceived software quality
  - Prove the validity of the assumptions made in design and requirement specifications through concrete demonstration
  - Validate that the software product functions as designed
  - Validate that the software product functions as required (that is, the requirements have been implemented appropriately)
- Deployment Discipline
  - Ensure that the software product is available for its end users

Supporting Disciplines

- Configuration & Change Management Discipline
  - Identify configuration items
  - Restrict changes to those items
  - Audit changes made to those items
  - Define and manage configurations of those items
- Project Management Discipline
  - Manage a software-intensive project; manage risk
  - Plan, staff, execute, and monitor a project
- Environment Discipline
  - Provide the software development organization with the software development environment – both processes and tools – that will support the development team
  - This includes configuring the process for a particular project, as well as developing guidelines in support of the project

RUP Best Practices

- RUP is a set of best practices
  - Guidelines for creating good documents, models...
  - Focus on having the right process (neither too heavy nor insufficient)
- Iterative development
  - Each iteration considers the 9 disciplines to some degree
- Requirements management to establish an understanding of customer / project
  - With use cases, requirements management tools and processes
- Component based architecture
  - Promotes reusability, unit testing
- Visual modeling (using UML)
- Continuous verification of quality (reviews, metrics, indicators…)
- Change management (baselines, change requests…)

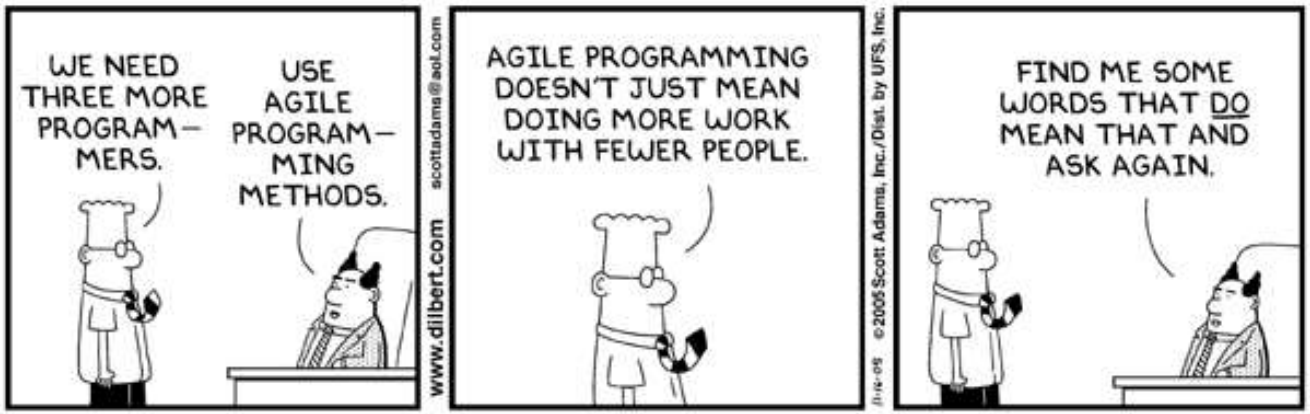
RUP and Agility?

- There is no contradiction between these two terms
  - A definition of agile processes can lead to a cumbersome process…
  - A definition of rich processes can lead to an agile process…

- Customizable software process engineering frameworks
  - Rational Method Composer
    - A tool for defining and monitoring your own development process
      - http://www-01.ibm.com/software/awdtools/rmc/
  - Eclipse Process Framework (EPF) – an Eclipse Project
    - Predefined and extensible roles, tasks, and development styles
    - Integration of tools - Metamodel-based
      - http://www.eclipse.org/epf/

Agile Methods

Agile Methods



© Scott Adams, Inc./Dist. by UFS, Inc.

© Scott Adams, Inc./Dist. by UFS, Inc.

Agile Manifesto (http://agilemanifesto.org)

Values of Agile Methods (1)

- Individuals and Interactions vs. Process and Tools
  - Individuals create value, therefore focus on them
  - Without a skilled craftsman, the best tools are useless
- Working Software vs. Comprehensive Documentation
  - A heavy process generates exhaustive documentation with all its drawbacks: ambiguity of language, cost of preparation, cost of maintenance to keep it up-to-date…
  - These documents are an illusion of progress
  - In agile methods, a single criterion measures the progress of a project: working software!
  - Documentation is a concrete support that helps produce software

Values of Agile Methods (2)

- Customer Collaboration vs. Contract Negotiation
  - If negotiation protects you more or less from financial risk, it can cause project failure and result in endless trials where everybody loses eventually
  - We must abandon the war with the customer / supplier and think as one team with a common goal: a successful project
- Responding to Change vs. Following a Plan
  - A predefined plan tends to make us unresponsive to events that occur during the project
  - Agile methods are designed to accommodate change, ensuring an accurate and adaptive strategic plan

Principles of the Agile Manifesto (1)

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

Principles of the Agile Manifesto (2)

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

Examples of Agile Approaches

- Extreme Programming (XP)
- Scrum
- Open Unified Process (OpenUP)
- Adaptive Software Development (ASD)
- Crystal Clear
- DSDM
- Feature Driven Development
- Lean software development
- Agile documentation
- Agile ICONIX

Microsoft Solutions Framework (MSF)

Agile Data

Agile Modeling

Agile Unified Process (AUP)

Essential Unified Process (EssUP)

Getting Real

…

see http://en.wikipedia.org/wiki/Agile_software_development

When to use in general?

- The culture of the organization is supportive of collaboration
- Team members are trusted (competence & confidence)
  - The organization is willing to live with the decisions developers make
- Fewer but competent team members
  - Ideally less than 10 co-located team members
  - Environment that facilitates rapid communication between team members
- Agile approaches are appropriate when requirements are difficult to predict, or change often
  - Situation where prototyping is required

When not to use?

- Agile approaches are not appropriate for every project
- Hard to use for
  - Large projects (>20 developers)
  - Distributed teams
  - Working environments not adapted to close collaboration
  - Critical applications (business-critical, safety-critical…)
  - Projects where a substantial specification is required prior to coding
  - When structure is rigid, e.g., military “Command-and-control” type projects

Extreme Programming (XP) (1)

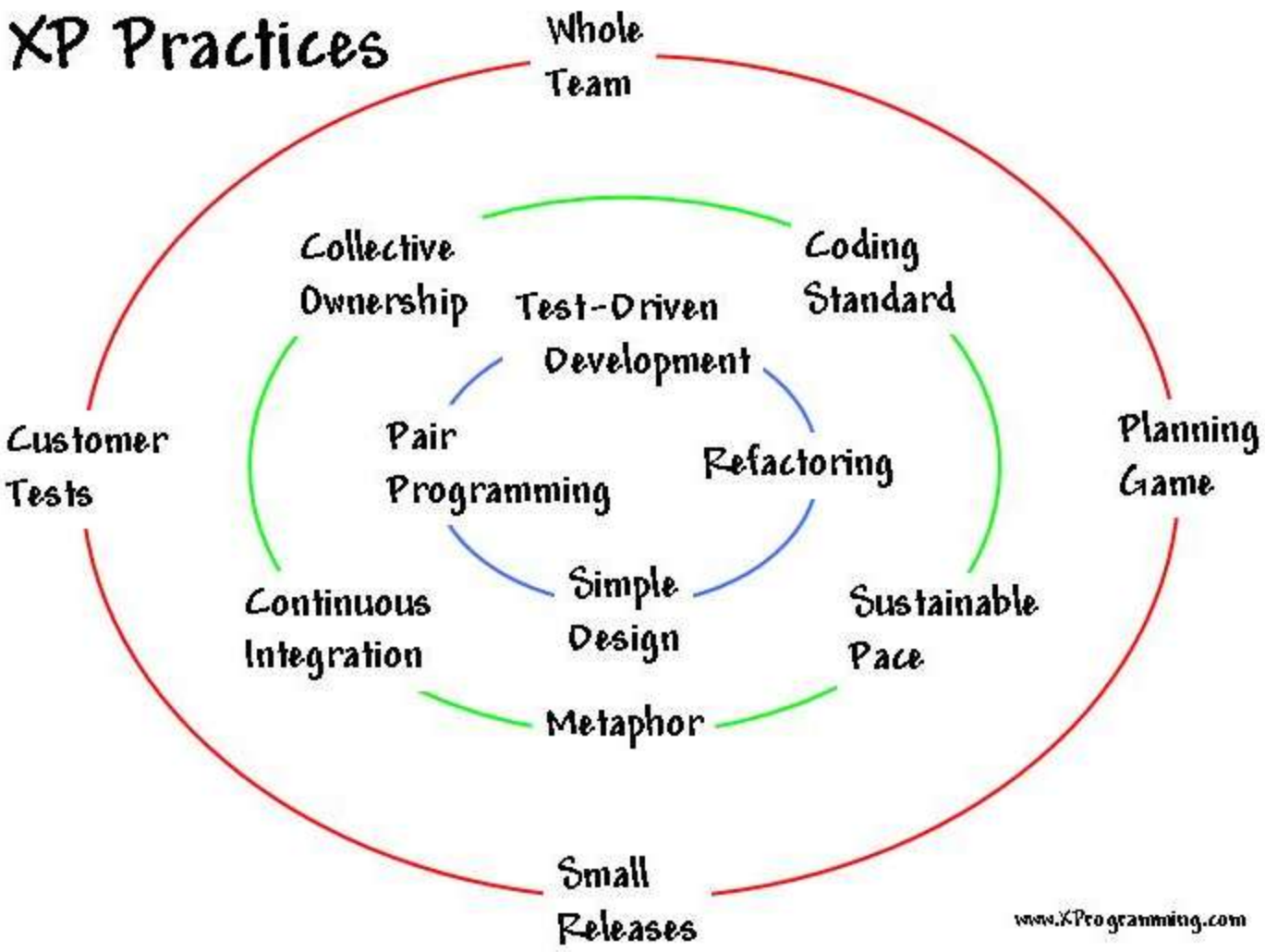
- Software is Listening, Testing, Coding, Designing. That's all there is to software. Anyone who tells you different is selling something.[1]
  - Listen to customers while gathering requirements, develop test cases (functional tests and unit tests), code the objects, design (refactor) as more objects are added to the system
- Listen – Test ~~Design~~ – Code – ~~Test~~ Design
- XP is a software development approach introduced by Kent Beck, Ward Cunningham, Ron Jeffries (~2000)
  - Lightweight process model for OO software development
  - Quick product delivery while retaining flexibility and maintaining quality
- Indented for small to medium size projects with a well integrated team
  - Small teams (up to 10-15 programmers)

[1] Kent Beck, author of Extreme Programming Explained

Extreme Programming (XP) (2)

- Why extreme?
  - XP is based on the extreme application of 12 practices (guidelines or rules) that support each other
  - There is no real new practice, but the combination of practices and their extreme application is new
  - Code is at the centre of the process
  - All non-production related tasks are superfluous
  - Technical documentation = commented source code
  - Other documentation is a waste of time (e.g., UML diagrams)
- Elements of success
  - Common workplace and working hours
  - All tests must be automated and executed in short time
  - On-site customer
  - Developer and client must commit 100% to XP practices

12 XP Practices



Source: http://www.xprogramming.com/

Elements of Extreme Programming

Roles

Clients (and tester), developer, sponsors

Artefacts

Metaphors

User stories (prioritized)

Tasks, unit tests, functional tests, code

Activities

Planning Game

Writing User Stories

Frequent Releases

System Metaphor

Activities (cont'd)

- Spike Solutions
- Simple Design
- Writing and Running Tests
- Refactoring
- Pair Programming
- Collective Code Ownership
- Continuous Integration
- 40 Hour Week
- On-site Customer
- Coding Standards
- Stand Up Meeting
- Backlog

Planning Game (1)

- Pieces
  - User Stories
- Players
  - Customer & Developer
- Moves
  - User story writing
    - Requirements are written by the customer on small index cards (simple, effective… and disposable)
    - User stories are written in business language and describe things that the system needs to do
    - Each user story is assigned a business value
    - For a few months-long project, there may be 50-100 user stories

Story: Broken phone call

When a telephone call is interrupted because of network disconnection, pre-emption, or whatever, the system provides a "disconnected" tone in the user's ear for three seconds, then automatically hangs up and reestablishes a dial tone.

Planning Game (2)

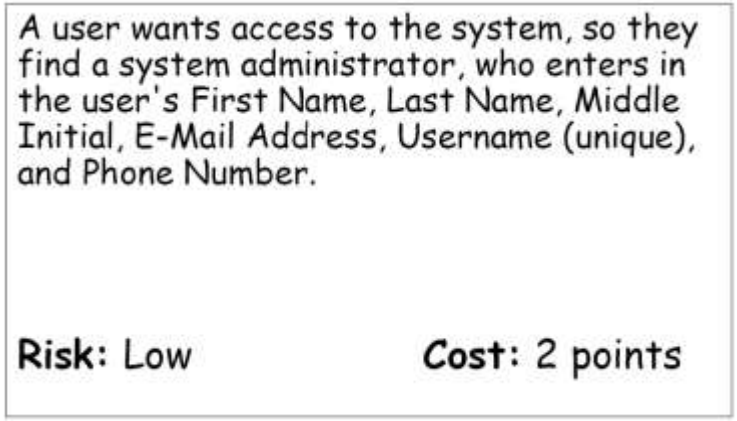
- Moves (cont'd)
  - Story estimation
    - Each user story is assigned a cost by the developer
    - Cost is measured in ideal development weeks (1-3 person weeks)
      - No interruptions, know exactly what to do
    - A story is split by the customer if it takes longer than 3 weeks to implement
    - If less than one week, it is too detailed and needs to be combined
    - Estimate risk (high, medium, low)
  - Commitment
    - Customer and developer decide which user stories constitute the next release
  - Value and risk first
    - Developer orders the user stories of the next release so that more valuable or riskier stories are moved earlier in the schedule

User Stories (1)

- A short description of the behavior of the system from the point of view of the customer/user
- Use the customer/user's terminology without technical jargon
- One for each major feature in the system
- Must be written by the customers/users
- Are used to create time estimates for release planning
- Replace a large requirements document

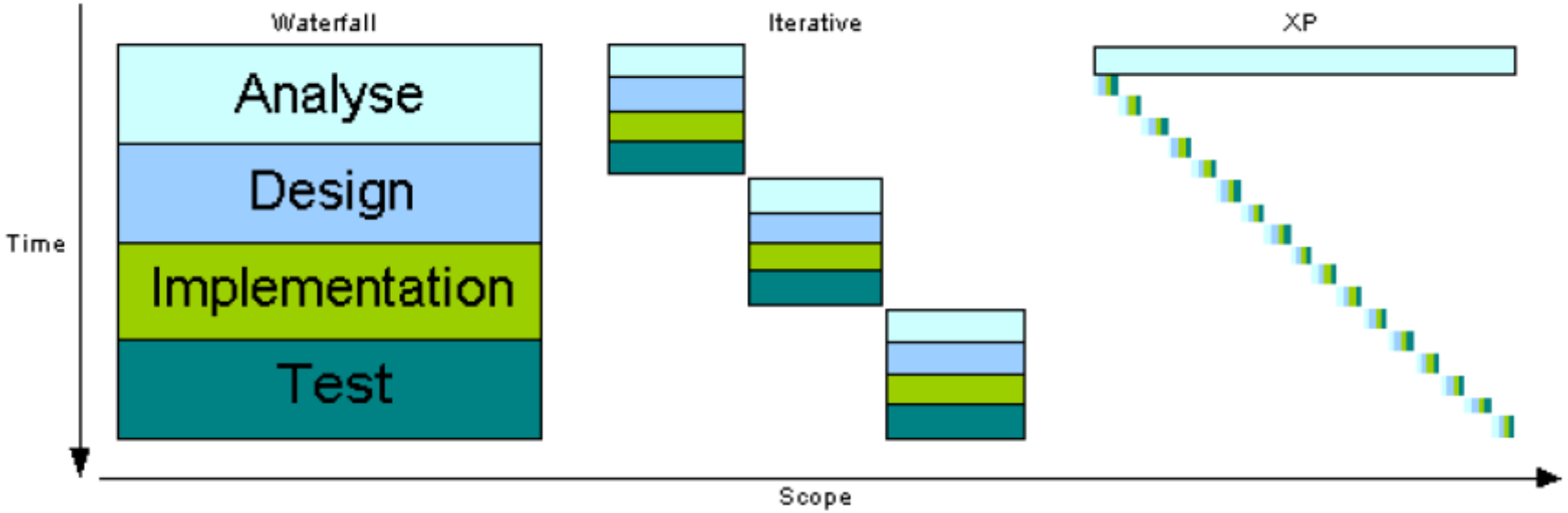
User Stories (2)

- Drive the creation of the acceptance tests
  - One or more tests to verify that a story has been properly implemented
- Different than requirements
  - Should only provide enough detail to make a reasonably low risk estimate of how long the story will take to implement
- Different than use cases
  - Written by customer, not programmers
- User stories have 3 crucial aspects
  - Card (= enough information to identify the story)
  - Conversation
    - Customer and programmers discuss the story to elaborate on the details
    - Verbal when possible, but documented when required
  - Confirmation (= acceptance tests)



Frequent Releases

- Highly iterative development process – release cycle of up to 3 months – iterations of up to 3 weeks
  - Short cycles during which the four phases take place in parallel
  - Functional system delivered at the end of each cycle
  - Frequent feedback from the customer
  - In each iteration the selected user stories are implemented
  - Each user story is split into programming tasks of 1-3 days



System Metaphor

- System metaphor provides a broad view of the project's goal
- Overall theme to which developers and clients can relate
- The system is built around one (or more) system metaphors
- If the metaphor is well chosen, it leads to design approaches, class names, better communication...
  - Chrysler is a manufacturing company; we make cars. Using a manufacturing metaphor to define the project was an important first step in getting the team (and management) on a level playing field. The concepts of lines, parts, bins, jobs, and stations are metaphors understood throughout the company. The team had the benefit of a very rich domain model developed by members of the team in the project's first iteration. It gave the members of the project an edge in understanding an extremely complex domain.[1]
  - The computer is like an office.

[1] Chrysler Comprehensive Compensation project, Garzaniti 1997

Spike Solution

- Need a mutual understanding of architecture
- XP does not do a big design up front
- No architecture phase

- Architectural Spike
  - Very simple program to test out solutions for tough technical or design problems – a throwaway prototype
  - May lead to a system metaphor

Simple Design

- Do the simplest thing that could possibly work
  - Create the best (simplest) design you can
  - Do not spend time implementing potential future functionality (requirements will change)
- Put in what you need when you need it
- A simple design ensures that there is less
  - to communicate
  - to test
  - to refactor

Tests

- Tests play the most important and central role in XP
- Tests are written before the code is developed
  - Forces concentration on the interface; accelerates development; safety net for coding and refactoring
- All tests are automated (test suites, testing framework)
- If user stories are considered the requirements, then tests may be considered as the specification of the system
- Two kinds of test
  - Acceptance tests (functional tests)
    - Clients provide test cases for their stories
    - Developers transform these into automatic tests
  - Unit tests
    - Developers write tests for their classes (before implementing the classes)
    - All unit tests must run 100% successfully all the time

Refactoring

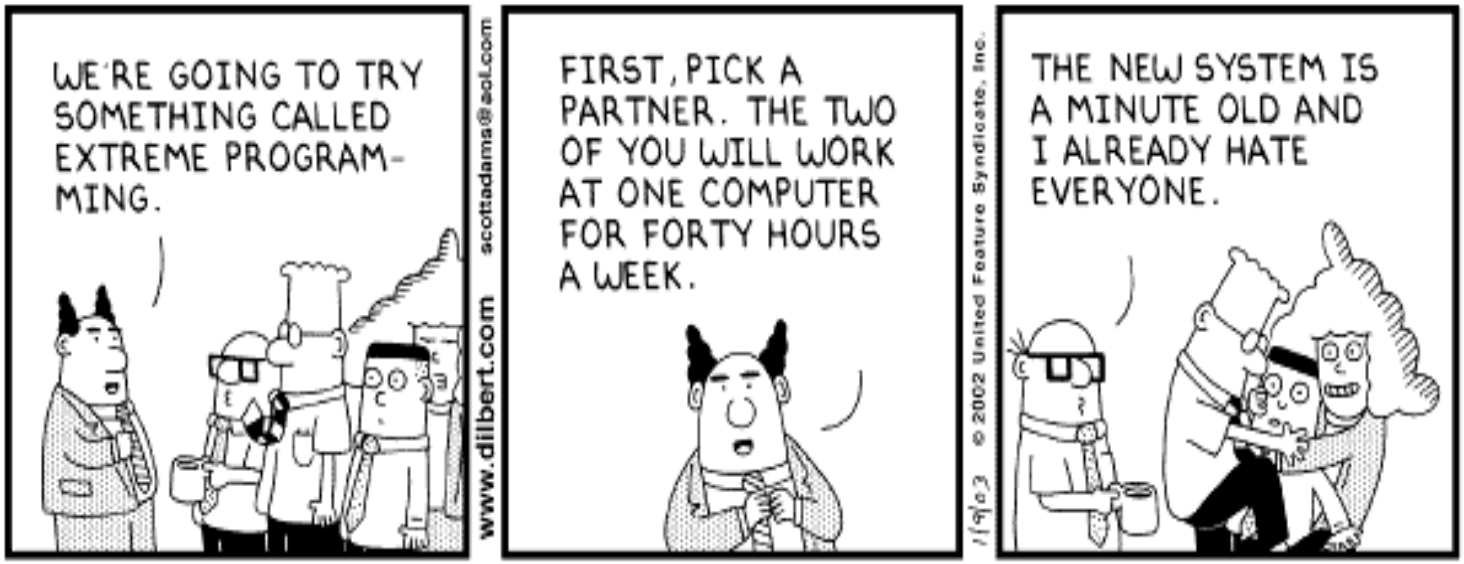
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.[1]

- The aim of refactoring is to
  - Make the design simpler
  - Make the code more understandable
  - Improve the tolerance of code to change
- Useful names should be used (system metaphor)
- Refactoring is continuous design
- Remove duplicate code
- Tests guarantee that refactoring did not break anything that worked!

[1] Martin Fowler

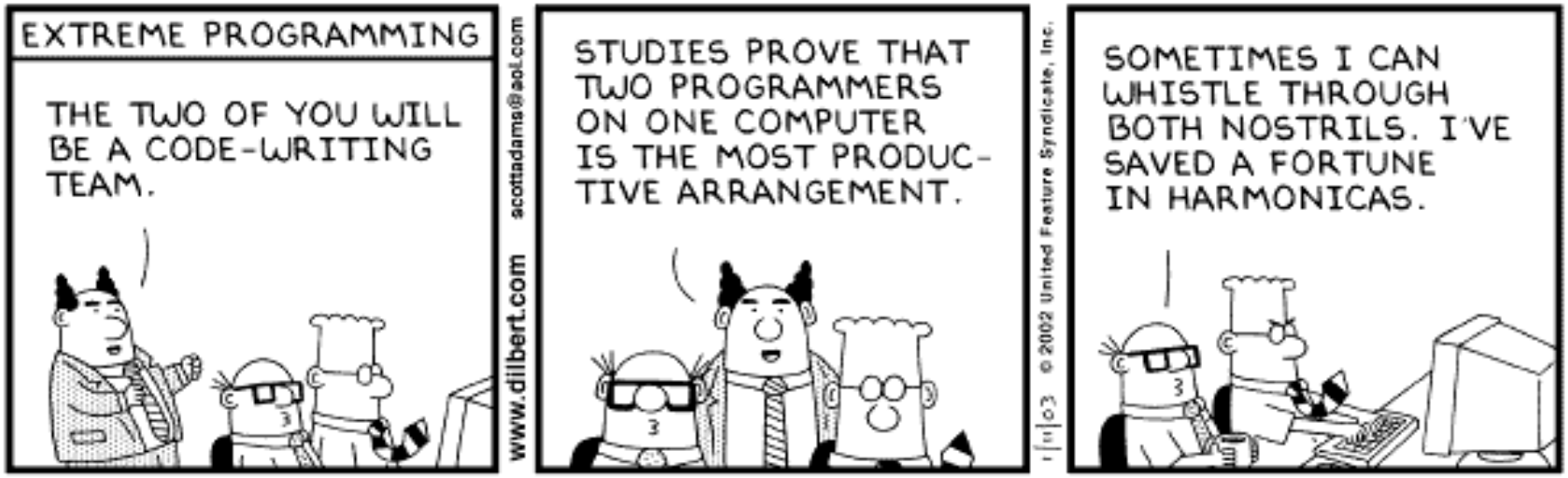
Pair Programming (1)

- Pairs change continuously (few times during a day)
  - Every programmer knows all aspects of the system
  - A programmer can be easily replaced in the middle of the project



Pair Programming (2)

- Costs 10-15% more than standalone programming
- Code is simpler (fewer LOC) with less defects (15%)
- Ensures continuous code inspection



Collective Code Ownership

- The code does not belong to any programmer but to the team
- Any programmer can (actually should) change any of the code at any time in order to
  - Make it simpler
  - Make it better
- Encourages the entire team to work more closely together
- Everybody tries to produce a high-quality system
  - Code gets cleaner
  - System gets better all the time
  - Everybody is familiar with most of the system

Continuous Integration

- Daily integration at least
- The whole system is built (integrated) every couple of hours
- A working, tested system is always available

- XP feedback cycle
  - Develop unit test
  - Code
  - Integrate
  - Run all units tests (100%)
  - Release

40 Hour Week

- Overtime is defined as time in the office when you do not want to be there.[1]

- If there is overtime one week, the next week should not include more overtime
- If more is needed then something is wrong with the schedule

- Keep people happy and balanced
- Rested programmers are more likely to refactor effectively, think of valuable tests, and handle the strong team interaction

[1] Ron Jeffries

On-site Customer

- The customer must always be available
  - To resolve ambiguities
  - Set priorities
  - Provide test cases
- User stories are not detailed, so there are always questions to ask the customer
- Customer is considered part of the team

Coding Standards

- Coding standards make pair programming and collective code ownership easier

- Common scheme for choosing names
- Common code formatting

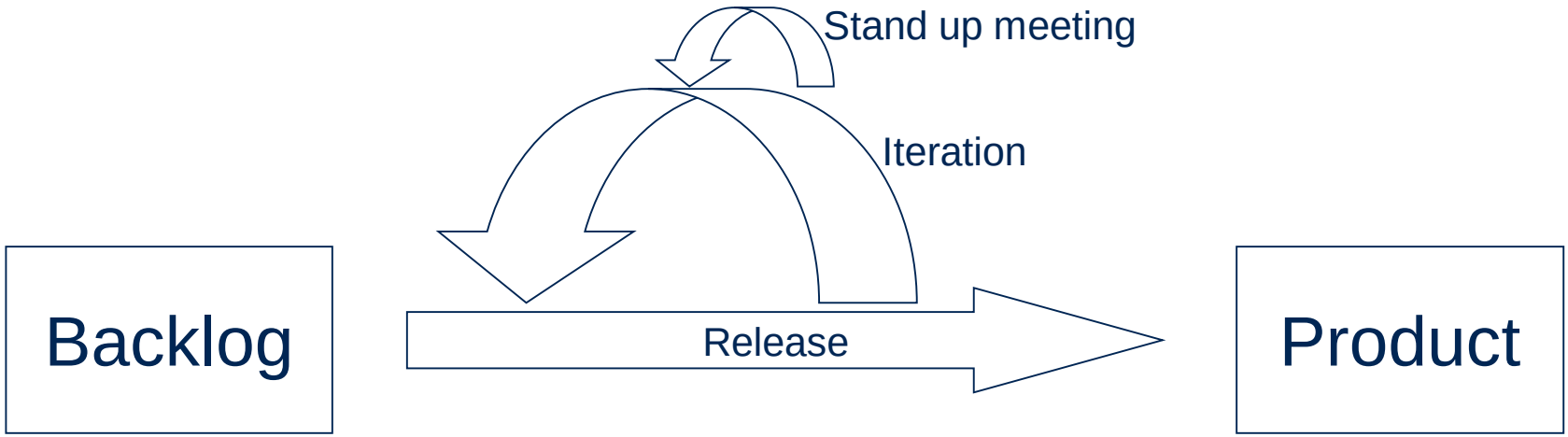
Stand Up Meeting

- Daily at the beginning of the day, 15min long, standing up
- Not for problem solving
  - Anyone may attend but only team members/sponsor/manager talk
- Helps avoid other unproductive meetings
- Make commitments in front of your peers
  - What did you do yesterday?
  - What will you do today?
  - What, if anything, is in your way?

Backlog

- Keep track of what needs to be done
  - Prioritized – value and risk
  - May contain user stories

- Requirements



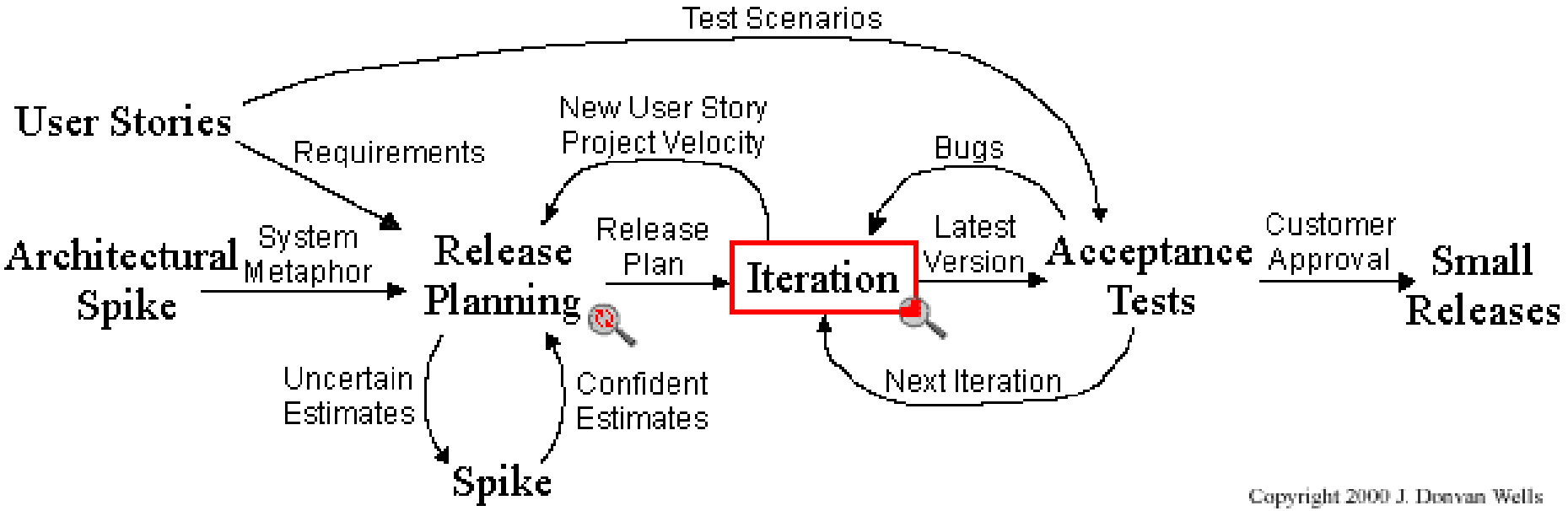
12 XP Practices Revisited

- XP practices are not new
- They are supposed to be used all together to get the full value of the approach
- The practices work together to create synergy

- The full value of XP will not come until all the practices are in place. Many of the practices can be adopted piecemeal, but their effects will be multiplied when they are in place together.[1]

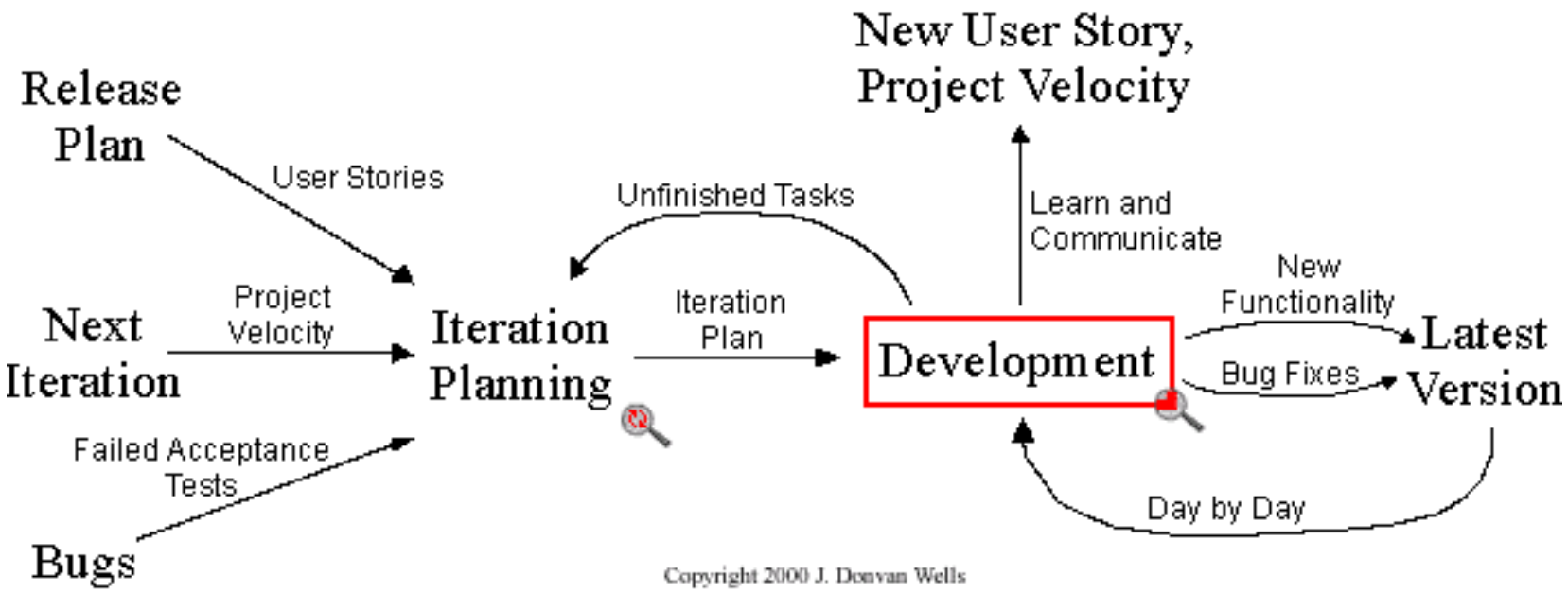
[1] Ken Beck

XP Process – Overview



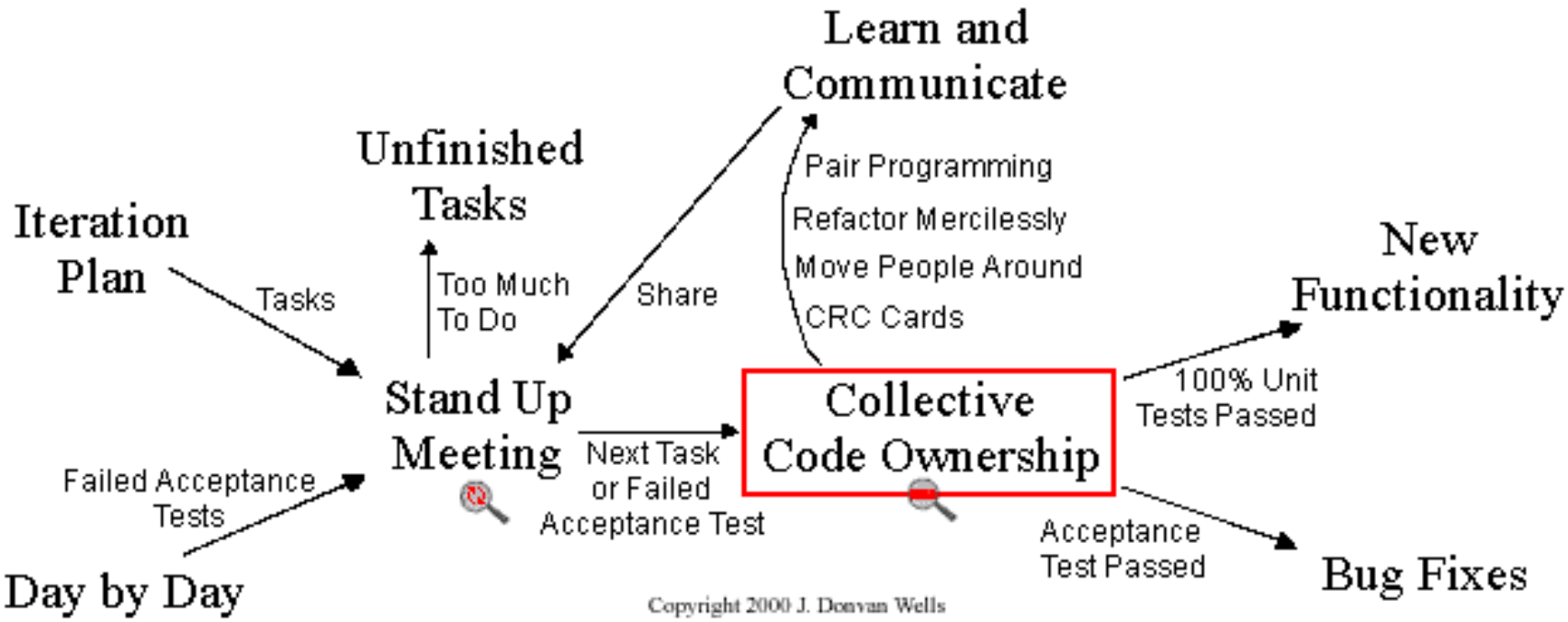
Source: http://www.extremeprogramming.org/

XP Process – Details of one "Iteration"



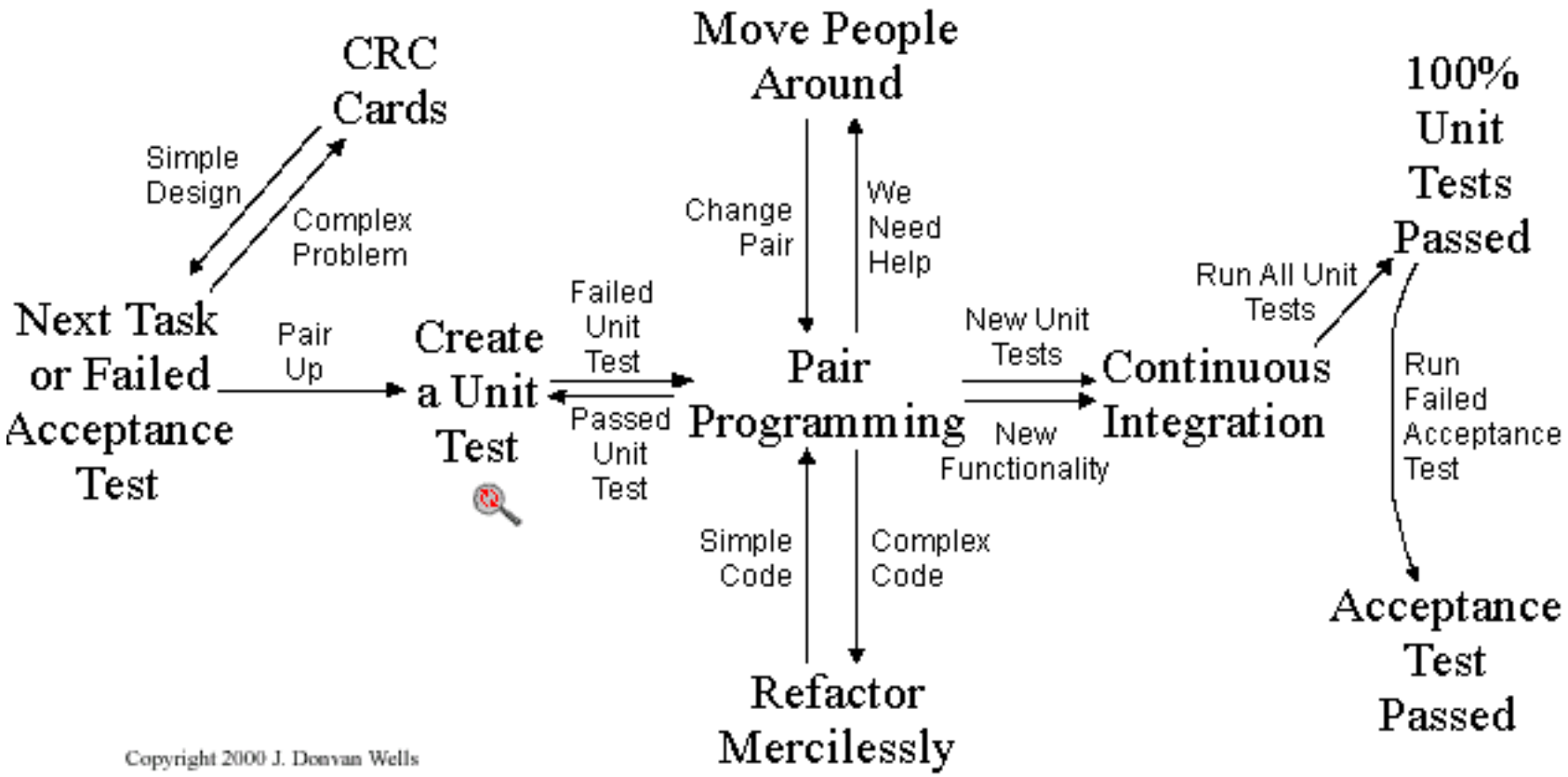
Source: http://www.extremeprogramming.org/

XP Process – Details of one "Development"



Source: http://www.extremeprogramming.org/

XP Process – Details of one "Collective Code Ownership"



Source: http://www.extremeprogramming.org/

Comparison – Requirements

- Requirements in RUP
  - Use cases
  - Requirements documents (including non-functional requirements)
- Requirements in XP
  - User stories on cards
  - On-site customers with strong involvement

Comparison – Risks (1)

General risks: Wrong/bad requirements, changing requirements, schedule and costs overruns

- RUP: Risks
  - Do the use cases adequately describe all requirements ?
- RUP: Risk management
  - Iterations, architecture for known risks, prioritization of use cases
- XP: Risks
  - Difficulty in having an on-site customer with a complete view of the client side and the ability to write stories & tests
  - NFRs not covered
  - Traceability?
- XP: Risk management
  - Iterations (very short), extreme simplicity, user stories selected by customer, estimates by developer
  - Stories that are difficult to estimate are considered higher risk

Comparison – Risks (2)

- Risk: later iterations may require more sophisticated design than the earlier iterations
  - RUP is predictive: elaboration phase considers all use cases and focuses on the ones with the highest impact on the architecture
    - Time wasted if some use cases end up being dropped
    - First delivery delayed
  - XP is adaptive: keep design simple and refactor when needed (extreme refactoring)
    - Can end up being very difficult if later iterations introduce major changes
- Risk: developers leave and new developers join
  - RUP: documentation of the architecture and key scenarios
  - XP: use of pair programming to teach new developers and to ensure knowledge of the system is well distributed

Suggested Improvements to XP

- eXtreme Requirements (Leite)
  - http://www-di.inf.puc-rio.br/~julio/Slct-pub/XR.pdf
- eXtreme Requirements improved (Leite and Leonardi)
  - http://www.mm.di.uoa.gr/~rouvas/ssi/caise2002/23480420.pdf
- EasyWinWin (Grünbacher and Hofer)
  - http://www.agilealliance.com/show/909
- XP modified (Nawrocki et al.)
  - http://www.cs.put.poznan.pl/jnawrocki/publica/re02-essen.doc

eXtreme Requirements (XR)

- Five improvements (or processes)
  - Notion of scenario and process
  - Non-functional requirements (constraints)
  - Traceability (lexicon)
  - Derivation of situations from scenarios
  - Formal expression of scenarios
- XR improved
  - Business rule concept
  - Interviews, workshops

EasyWinWin

- Complements XP with requirements negotiation
- Defines 7 activities that guide stakeholders through a negotiation process
  - Review and expand negotiation topics (features, services, interfaces, system properties, cost, schedule…)
  - Brainstorm stakeholder interest
  - Converge on win-win conditions
  - Capture a glossary of terms
  - Prioritize win-win conditions (polling used (a) to determine priorities of win-win conditions, and (b) to reveal conflicts and different perceptions among stakeholders)
  - Reveal issues and constraints (examine the results of the prioritization poll to analyze patterns of agreement and disagreement)
  - Identify issues, options, and agreements

XP modified

- Three improvements to XP
  - Written requirements documentation
  - Several customer representatives
  - Requirements engineering phase

Is agile better? The debate is still on! (1)

- Daniel Berry (on XP and RE)
  - However, all the data I have seen say that full RE is the most cost-effective way to produce quality software, and that it will beat out any agile method any time in the cost and quality attributes.
  - When all the details are hammered out during an RE process that lasts until it is done (and not until a predetermined date), the development proceeds so much quicker and with so few bugs!
- Chrysler Comprehensive Compensation (C3) project – Success or Failure?
  - http://calla.ics.uci.edu/histories/ccc/
  - http://c2.com/cgi/wiki?CthreeProjectTerminated

Is agile better? The debate is still on! (2)

- After seeing heavy and light processes, the decision is not always binary
- It really depends on your context and your project
  - What are your needs?
  - What are the appropriate artifacts (documents, models…)?
  - What are the tasks and roles?
  - What are the appropriate tools?
  - What process elements are adequate and how to assemble them?
- Based on an analysis of your context and project, you should be able to make the appropriate decisions
- See also mandatory supplementary material by Alan M. Davies posted on the course website