Software Engineering

Session 2 – Main Theme
Software Development Lifecycles (SDLCs)

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Presentation material partially based on textbook slides
Software Engineering: A Practitioner's Approach (7/e)
by Roger S. Pressman

Agenda – Session Overview

1 Session Overview

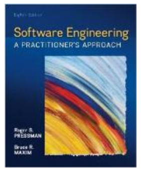
2 Software Engineering LifeCycles (SDLCs)

3 Summary and Conclusion

What is the class about?

- Course description and syllabus:
  - » http://www.nyu.edu/classes/jcf/g22.2440-001/
  - » http://www.cs.nyu.edu/courses/spring15/G22.2440-001/

- Textbooks:



  - » ***Software Engineering: A Practitioner's Approach***
    Roger S. Pressman
    McGraw-Hill Higher International
    ISBN-10: 0078022126, ISBN-13: 978-0078022128, 8th Edition (01/23/14)
  - » Recommended:
    - » Code Complete: A Practical Handbook of Software Construction, 2nd Edition
    - » The Mythical Man-Month: Essays on Software Engineering, 2nd Edition

Agenda

- Software Engineering Detailed
- Process Models
- Agile Development
- Software Engineering Knowledge
- Roles and Types of Standards
  - ISO 12207: Life Cycle Standard
  - IEEE Standards for Software Engineering Processes and Specifications
- Summary and Conclusion
  - Readings
  - Assignment #1
  - Course Project

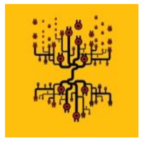
Icons / Metaphors

Information

Common Realization

Knowledge/Competency Pattern

Governance


Alignment

Solution Approach

Agenda

1 Session Overview

2 Software Engineering LifeCycles (SDLCs)

3 Summary and Conclusion

Agenda – Software Engineering LifeCycles (SDLCs)

What is a SDLC

System Development Life Cycle:

- It is about developing a software-driven solution to a business problem
- It concerns a process which takes from two months to two years
- This is called a System Development Life Cycle but it should really be called a (Business) Solution Development Life Cycle

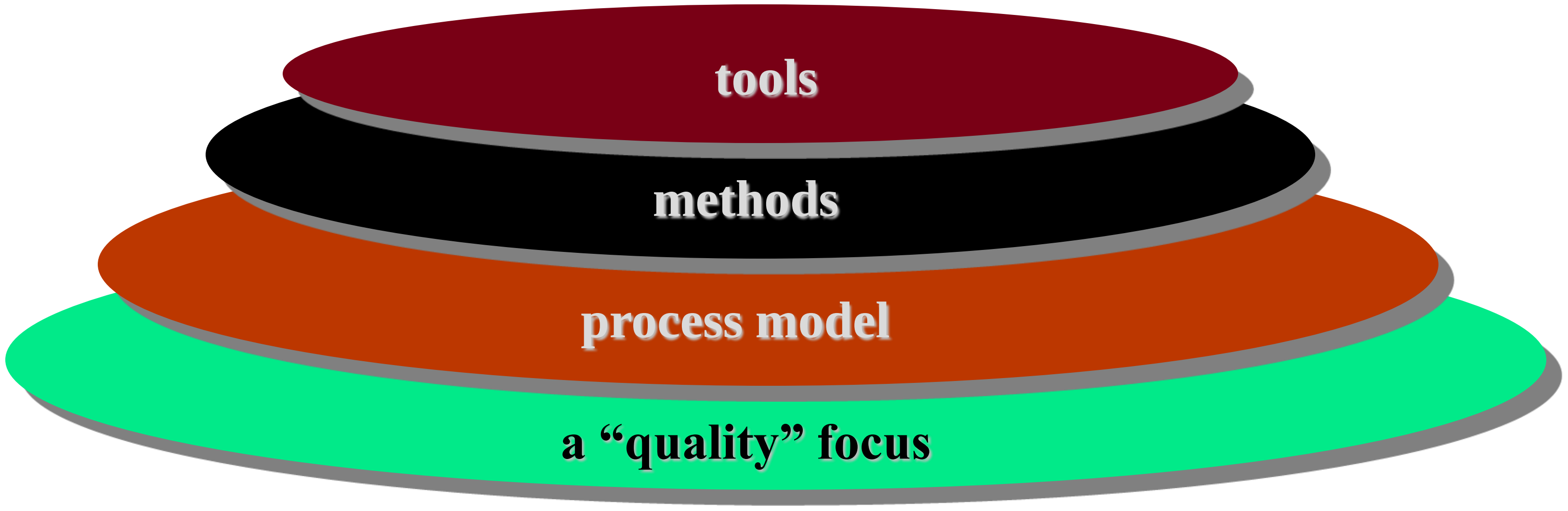
Software Engineering (1/2)

- Some realities
  - A concerted effort should be made to understand the problem before a software solution is developed
  - Design becomes a pivotal activity
  - Software should exhibit high quality
  - Software should be maintainable
- The seminal definition
  - Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines

Software Engineering (2/2)

- The IEEE definition
  - Software Engineering:
    - (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software
    - (2) The study of approaches as in (1)

A Layered Technology



Software Engineering

A Process Framework

Process framework

- **Framework activities**
  - work tasks
  - work products
  - milestones & deliverables
  - QA checkpoints
- **Umbrella Activities**

Framework Activities

- Communication
- Planning
- Modeling
  - Analysis of requirements
  - Design
- Construction
  - Code generation
  - Testing
- Deployment

Umbrella Activities

- Software project management
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Work product preparation and production
- Reusability management
- Measurement
- Risk management

Adapting a Process Model

- The overall flow of activities, actions, and tasks and the interdependencies among them
- The degree to which actions and tasks are defined within each framework activity
- The degree to which work products are identified and required
- The manner which quality assurance activities are applied
- The manner in which project tracking and control activities are applied
- The overall degree of detail and rigor with which the process is described
- The degree to which the customer and other stakeholders are involved with the project
- The level of autonomy given to the software team
- The degree to which team organization and roles are prescribed

The Essence of Practice

- Polya* suggests:
  1. Understand the problem (communication and analysis)
  2. Plan a solution (modeling and software design)
  3. Carry out the plan (code generation)
  4. Examine the result for accuracy (testing and quality assurance)

* http://www.stevemcconnell.com/rl-top10.htm

Understand the Problem

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the Solution

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry Out the Plan

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

Examine the Result

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

David Hooker's General Principles*

- 1: *The Reason It All Exists*
- 2: *KISS (Keep It Simple, Stupid!)*
- 3: *Maintain the Vision*
- 4: *What You Produce, Others Will Consume*
- 5: *Be Open to the Future*
- 6: *Plan Ahead for Reuse*
- 7: *Think!*

* http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment

Software Myths

- Affect managers, customers (and other non-technical stakeholders) and practitioners
- Are believable because they often have elements of truth,

but …

- Invariably lead to bad decisions,

therefore …

- Insist on reality as you navigate your way through software engineering

How It all Starts

- Every software project is precipitated by some business need
  - The need to correct a defect in an existing application;
  - The need to the need to adapt a ‘legacy system’ to a changing business environment;
  - The need to extend the functions and features of an existing application, or
  - The need to create a new product, service, or system.

SDLC Characteristics

- Whatever form it takes, it is always followed by a maintenance cycle:
  - Maintenance is the most expensive part
  - If all the steps are done carefully maintenance is reduced
  - For maintenance to be effective , documentation must exist

Business Solution Characteristics

- A software-driven solution consists of two parts:
  - Model
    - Prototypes
    - Diagrams and supporting Documents
  - System
    - Hardware
    - Software

Some Definitions (1/3)

- Prototype
  - An initial software-driven solution usually done with a rapid development tool
  - Usually has limited functionality
  - Users can see results very quickly
- Planning
  - The process of gathering what is needed to solve a business problem
  - Includes a feasibility study
  - Includes project steps

Some Definitions (2/3)

- Analysis
  - The process of determining detail requirements in the form of a model
- Design
  - The process of drawing blueprints for a new system at a high-level first then at a detailed level
- Construction
  - The actual coding of the model into a software package
  - Uses one or more programming languages
    - Java
    - C#
    - C++
    - etc.

Some Definitions (3/3)

- Implementation
  - Doing whatever is necessary to startup a system
  - Includes:
    - Database
    - Networks
    - Hardware configuration
- Maintenance
  - Doing whatever is necessary to keep a system running
  - Includes:
    - Repairs to correct errors
    - Enhancements to accommodate changes in requirements

Deliverables

- Deliverables consist mainly of diagrams and their supporting documentation
- For example:
  - Models that emphasize dynamics
  - Models that emphasize structure
  - Models can be used for specifying the outcome of analysis
  - Models can be used for specifying the outcome of design

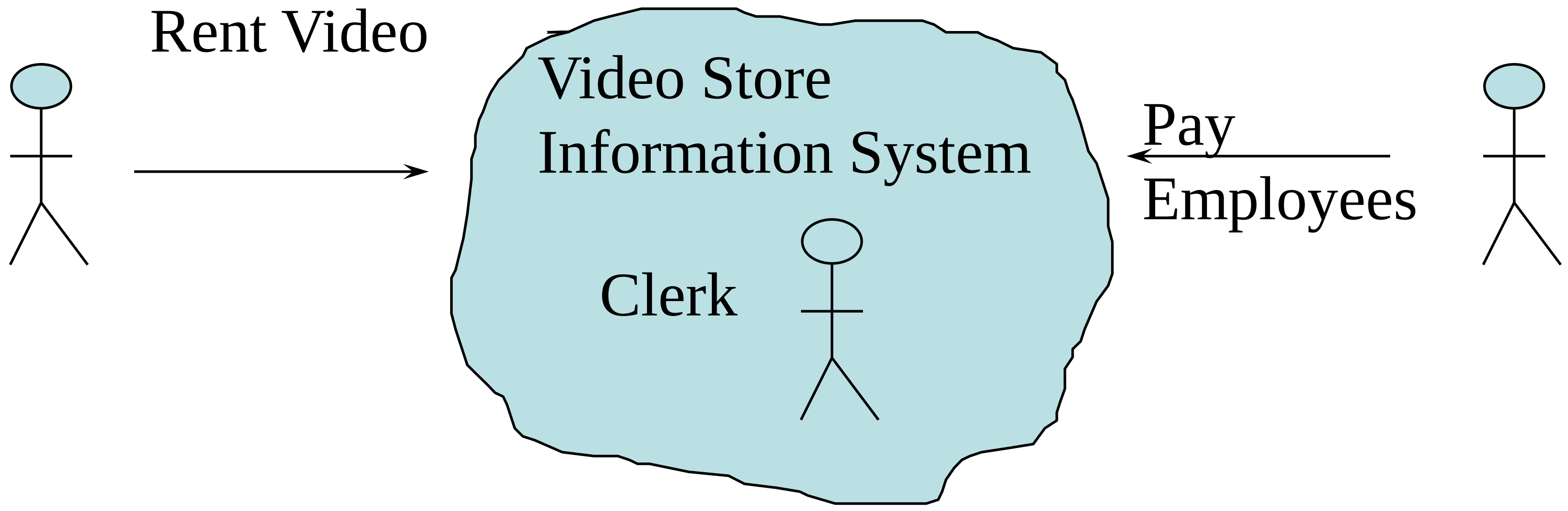
Sample Deliverables – Planning (1/3)

- Planning:
  - System Functions
    - A simple list of each requirement a system must do
    - For example:
      - record video rental
      - calculate fine
  - System Attributes
    - A simple property describing each requirement of a system
    - For example:
      - record video rental under 15 seconds
      - calculate fine and return response in 5 seconds

Sample Deliverables – Planning (2/3)

- Planning:

Environmental Diagram



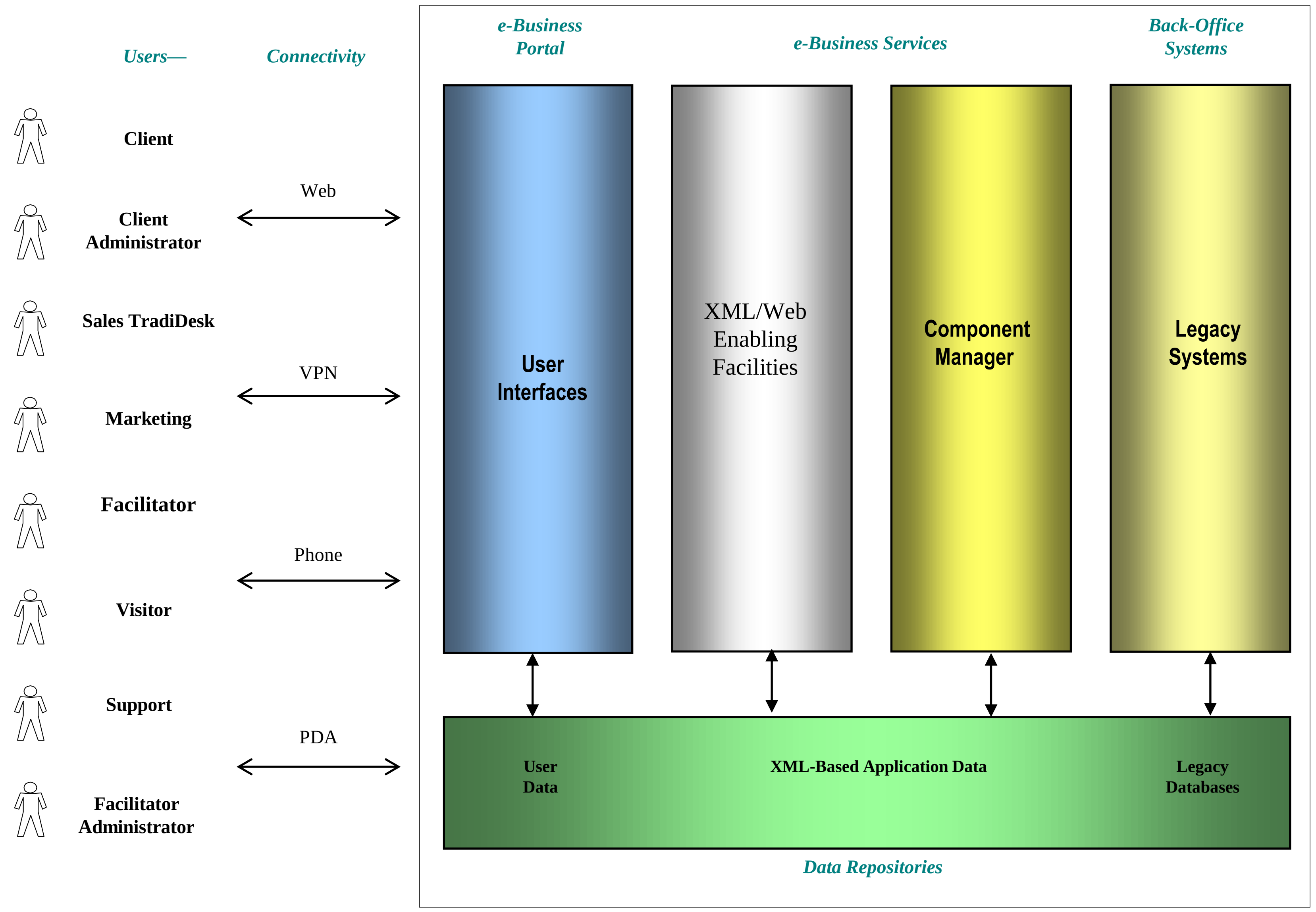
Sample Deliverables – Planning (3/3)

- Planning:
  - Prototype
    - Recall it is a first system usually done with a rapid development tool
    - Since users can see results very quickly they will pay attention
    - Final product is seldom created in same tool as the prototype

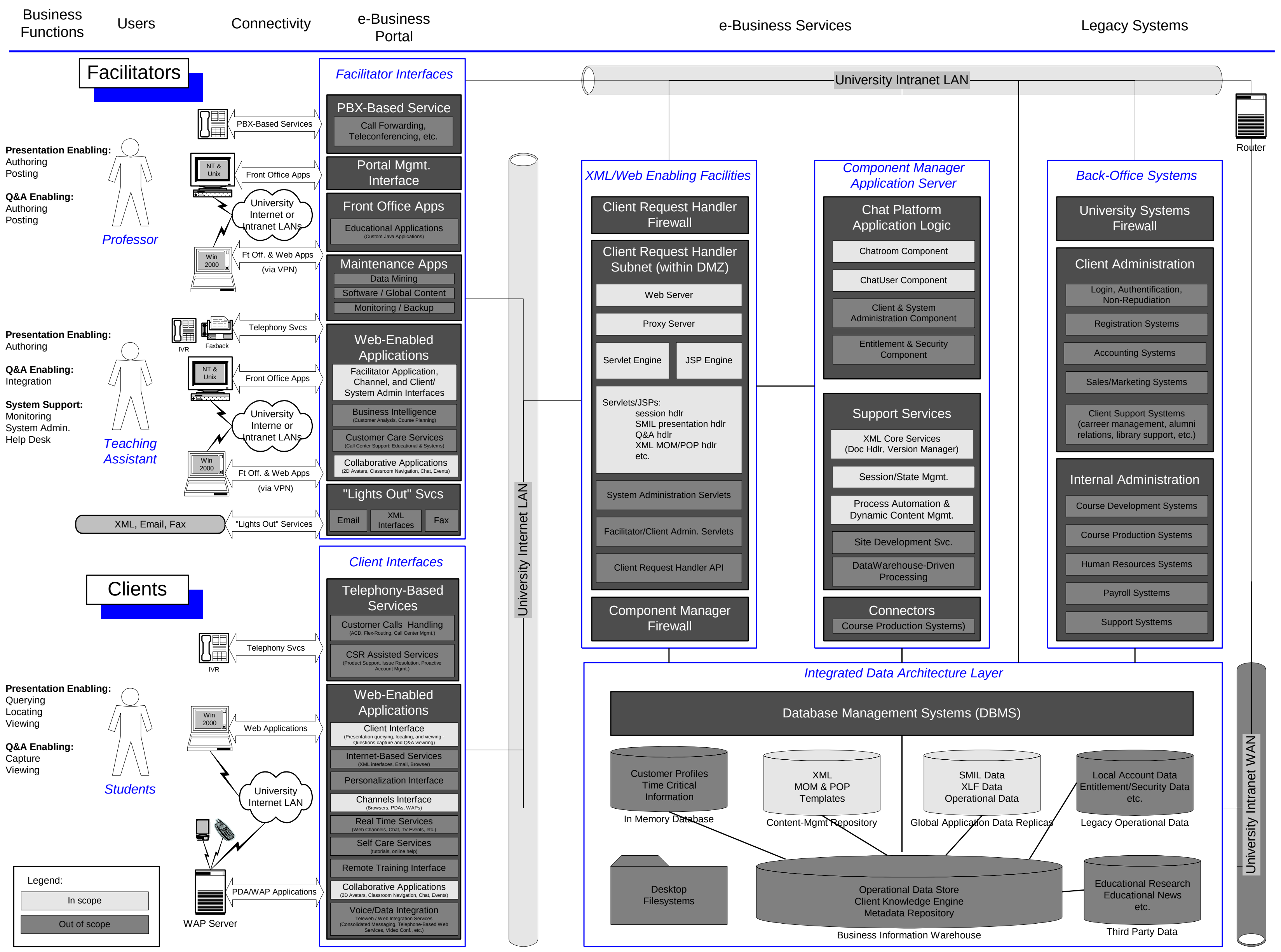
Sample Deliverables - Analysis

- Analysis:
  - Use case
    - Shows the dynamics between the users (actors) of the system and the system itself
    - This is a narrative representation
  - Conceptual Diagram
    - Shows the structure of the objects and their relationships
    - This is a graphical representation
  - System Sequence Diagram
    - Shows the dynamics between the users (actors) of the system and the system itself
    - This is a graphical representation
  - Contracts
    - Shows the state of each object before each action
    - This is a narrative representation

Sample High-Level Architecture Design Conceptual Blueprint

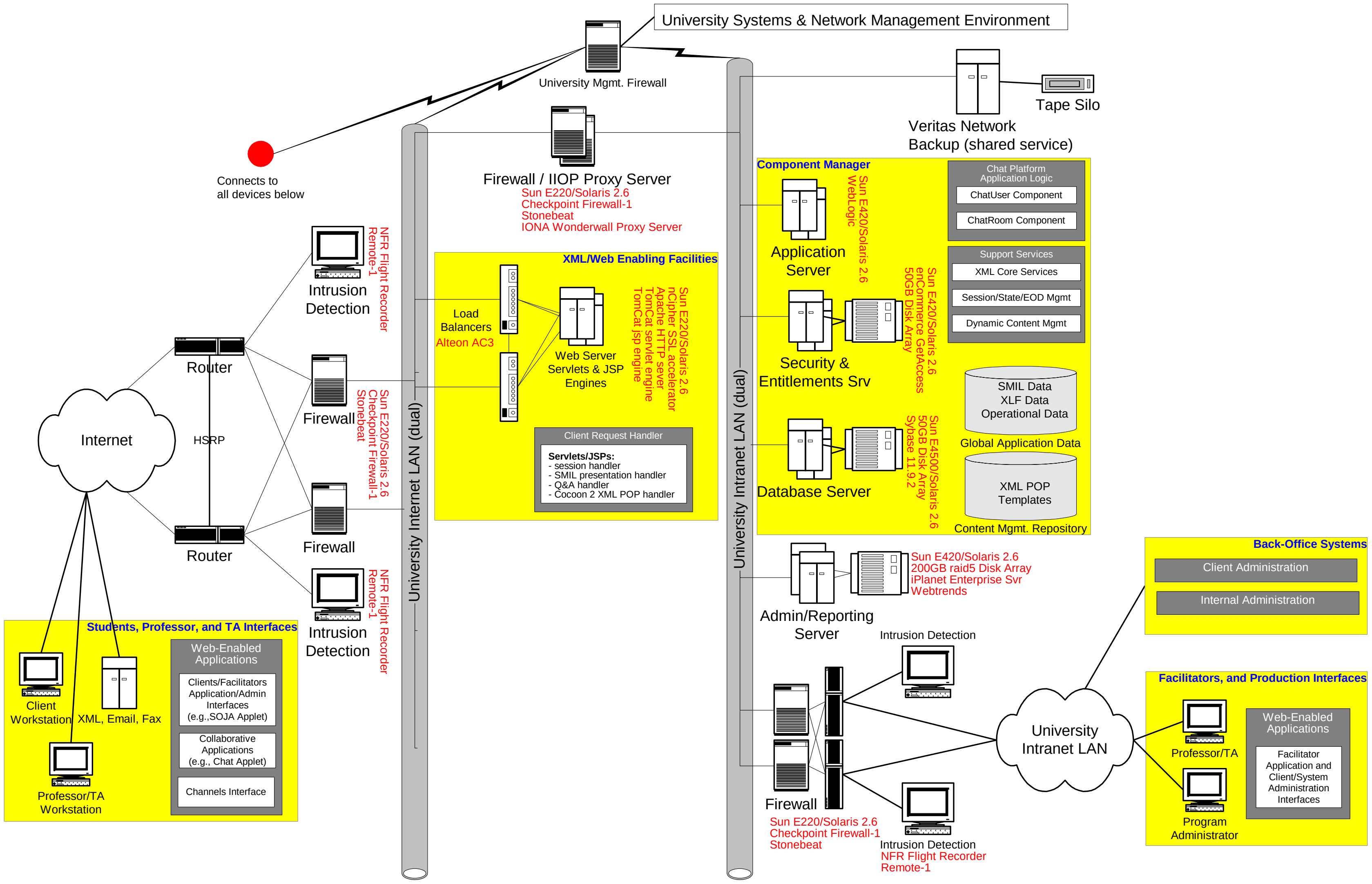


Sample High-Level Architecture Design Logical Blueprint



Business Functions
Users
Connectivity
e-Business Portal
e-Business Services
Legacy Systems
Facilitators
Presentation Enabling:
Authoring
Posting
Q&A Enabling:
Authoring
Posting
Professor
Presentation Enabling:
Authoring
Q&A Enabling:
Integration
System Support:
Monitoring
System Admin.
Help Desk
Teaching Assistant
XML, Email, Fax
Clients
Presentation Enabling:
Querying
Locating
Viewing
Q&A Enabling:
Capture
Viewing
Students
Legend:
In scope
Out of scope
PBX-Based Services
NT & Unix
Front Office Apps
University Internet or Intranet LANs
Win 2000
Ft Off. & Web Apps
(via VPN)
IVR
Faxback
Telephony Svcs
NT & Unix
Front Office Apps
University Interne or Intranet LANs
Win 2000
Ft Off. & Web Apps
(via VPN)
"Lights Out" Services
Telephony Svcs
IVR
Win 2000
Web Applications
University Internet LAN
PDA/WAP Applications
WAP Server
Facilitator Interfaces
PBX-Based Service
Call Forwarding, Teleconferencing, etc.
Portal Mgmt. Interface
Front Office Apps
Educational Applications
(Custom Java Applications)
Maintenance Apps
Data Mining
Software / Global Content
Monitoring / Backup
Web-Enabled Applications
Facilitator Application, Channel, and Client/ System Admin Interfaces
Business Intelligence
(Customer Analysis, Course Planning)
Customer Care Services
(Call Center Support: Educational & Systems)
Collaborative Applications
(2D Avatars, Classroom Navigation, Chat, Events)
"Lights Out" Svcs
Email
XML Interfaces
Fax
Client Interfaces
Telephony-Based Services
Customer Calls Handling
(ACD, Flex-Routing, Call Center Mgmt.)
CSR Assisted Services
(Product Support, Issue Resolution, Proactive Account Mgmt.)
Web-Enabled Applications
Client Interface
(Presentation querying, locating, and viewing - Questions capture and Q&A viewring)
Internet-Based Services
(XML interfaces, Email, Browser)
Personalization Interface
Channels Interface
(Browsers, PDAs, WAPs)
Real Time Services
(Web Channels, Chat, TV Events, etc.)
Self Care Services
(tutorials, online help)
Remote Training Interface
Collaborative Applications
(2D Avatars, Classroom Navigation, Chat, Events)
Voice/Data Integration
Teleweb / Web Integration Services
(Consolidated Messaging, Telephone-Based Web Services, Video Conf., etc.)
University Internet LAN
University Intranet LAN
Router
XML/Web Enabling Facilities
Client Request Handler Firewall
Client Request Handler Subnet (within DMZ)
Web Server
Proxy Server
Servlet Engine
JSP Engine
Servlets/JSPs:
session hdlr
SMIL presentation hdlr
Q&A hdlr
XML MOM/POP hdlr
etc.
System Administration Servlets
Facilitator/Client Admin. Servlets
Client Request Handler API
Component Manager Firewall
Component Manager Application Server
Chat Platform Application Logic
Chatroom Component
ChatUser Component
Client & System Administration Component
Entitlement & Security Component
Support Services
XML Core Services
(Doc Hdlr, Version Manager)
Session/State Mgmt.
Process Automation & Dynamic Content Mgmt.
Site Development Svc.
DataWarehouse-Driven Processing
Connectors
Course Production Systems)
Back-Office Systems
University Systems Firewall
Client Administration
Login, Authentification, Non-Repudiation
Registration Systems
Accounting Systems
Sales/Marketing Systems
Client Support Systems
(carreer management, alumni relations, library support, etc.)
Internal Administration
Course Development Systems
Course Production Systems
Human Resources Systems
Payroll Systtems
Support Systtems
University Intranet WAN
Integrated Data Architecture Layer
Database Management Systems (DBMS)
Customer Profiles Time Critical Information
In Memory Database
XML MOM & POP Templates
Content-Mgmt Repository
SMIL Data XLF Data Operational Data
Global Application Data Replicas
Local Account Data Entitlement/Security Data etc.
Legacy Operational Data
Desktop Filesystems
Operational Data Store Client Knowledge Engine Metadata Repository
Business Information Warehouse
Educational Research Educational News etc.
Third Party Data

Sample High-Level Architecture Design Physical Blueprint

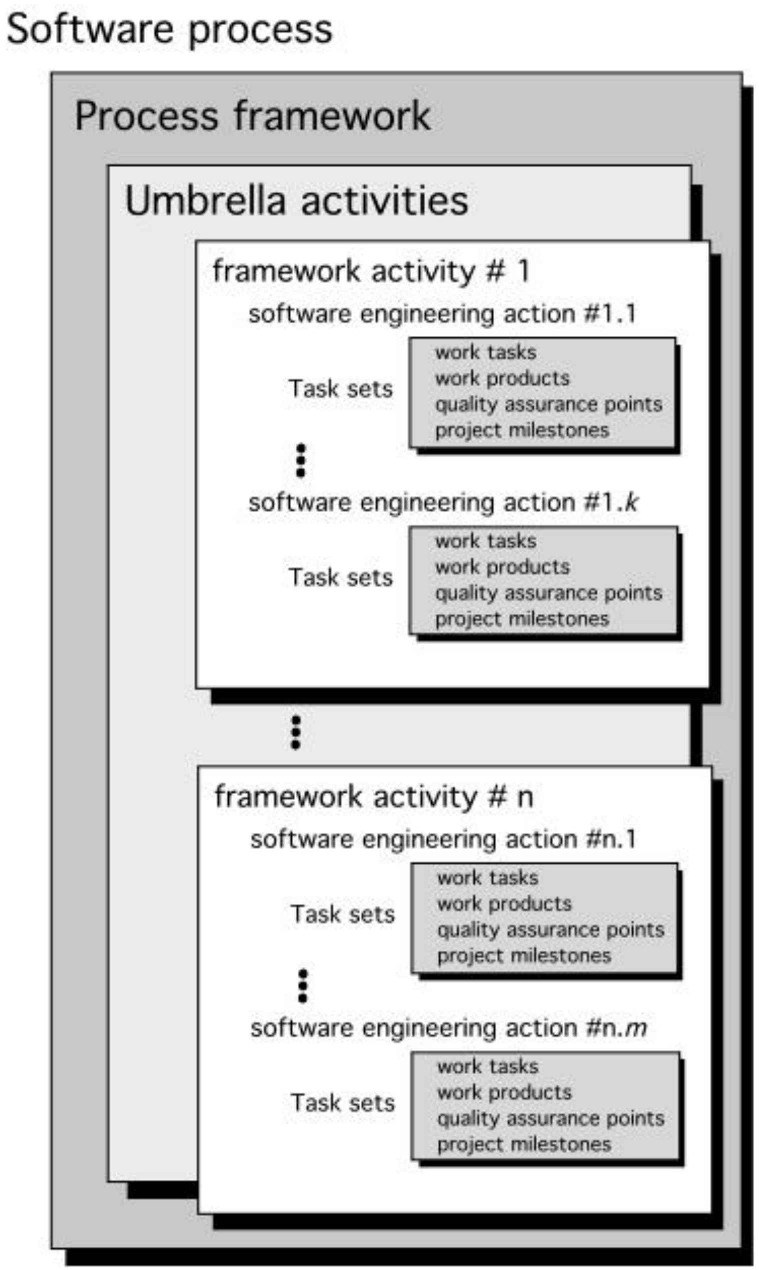


Sample Deliverables - Design

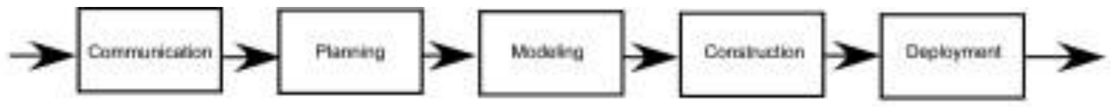
- Design:
  - Interaction Diagram
    - Shows the interaction between objects
    - This is a graphic representation
    - It is a dynamic blueprint
  - Class Diagram
    - Shows the structure between objects
    - Shows the structure inside objects
    - This is a graphic representation
    - It is a static blueprint

Agenda – Software Engineering Lifecycles (SDLCs)

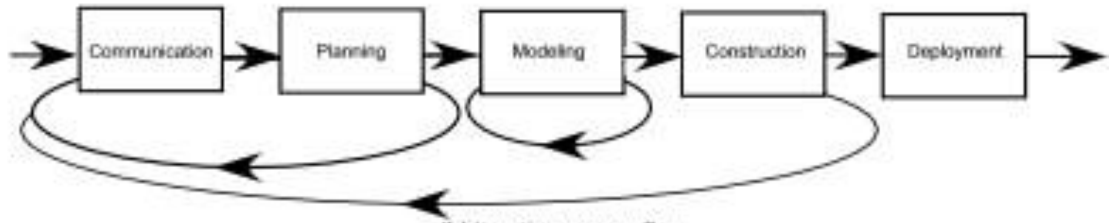
A Generic Process Model



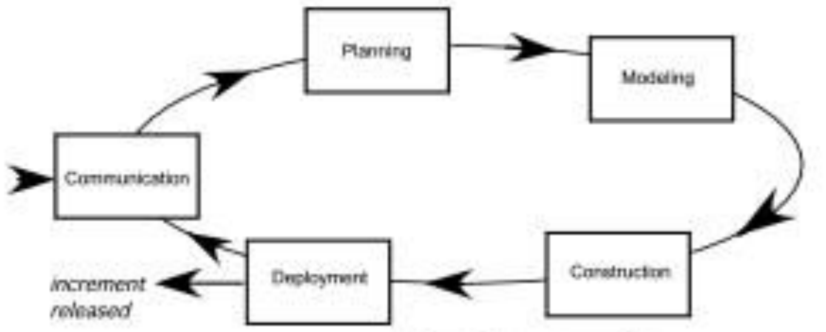
Process Flow



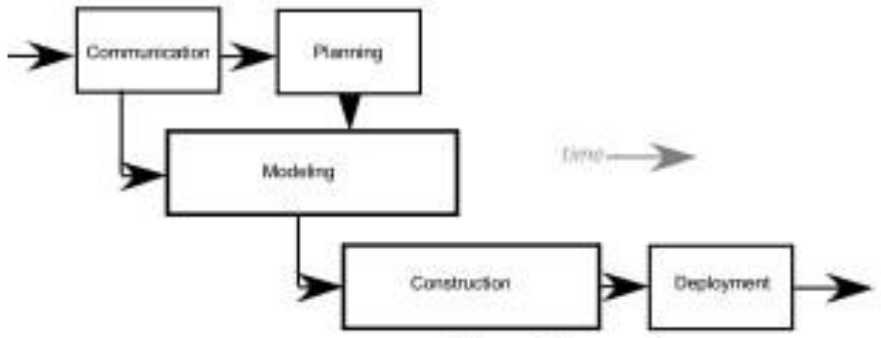
(a) linear process flow



(b) iterative process flow



(c) evolutionary process flow



(d) parallel process flow

Identifying a Task Set

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action
  - A list of the tasks to be accomplished
  - A list of the work products to be produced
  - A list of the quality assurance filters to be applied

Process Patterns

- A *process pattern*
  - describes a process-related problem that is encountered during software engineering work,
  - identifies the environment in which the problem has been encountered, and
  - suggests one or more proven solutions to the problem
- Stated in more general terms, a process pattern provides you with a *template* [Amb98] - a consistent method for describing problem solutions within the context of the software process

Process Pattern Types

- *Stage patterns* - defines a problem associated with a framework activity for the process
- *Task patterns* - defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice
- *Phase patterns* - define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature

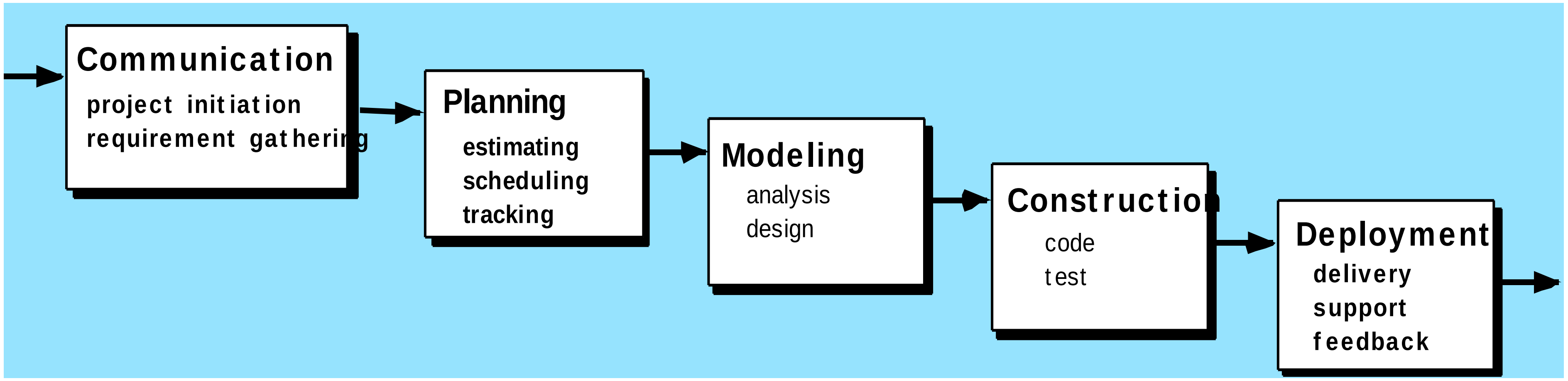
Prescriptive Models

- Prescriptive process models advocate an orderly approach to software engineering

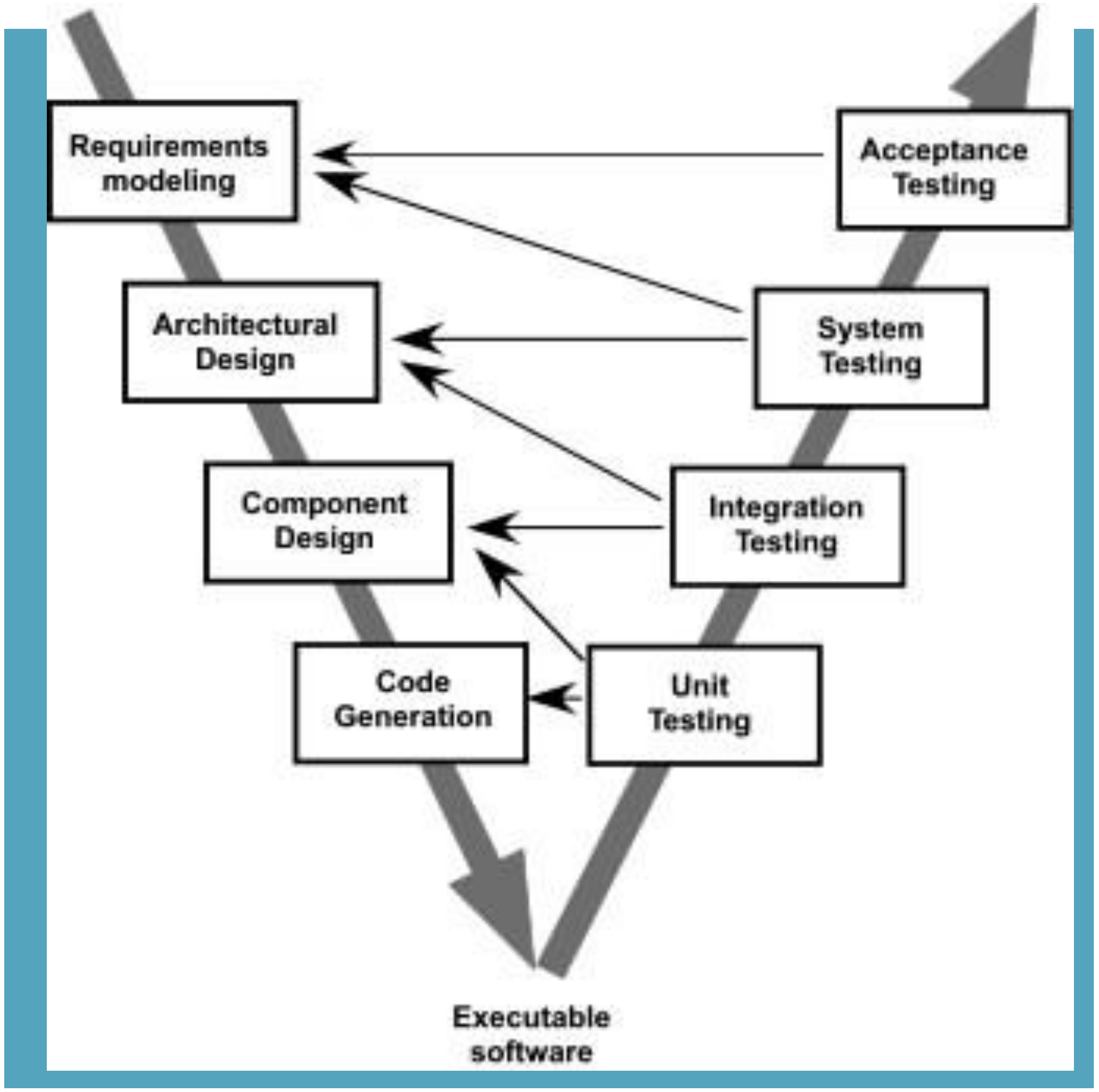
That leads to a few questions …

- If prescriptive process models strive for structure and order, are they inappropriate for a software world that thrives on change?
- Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

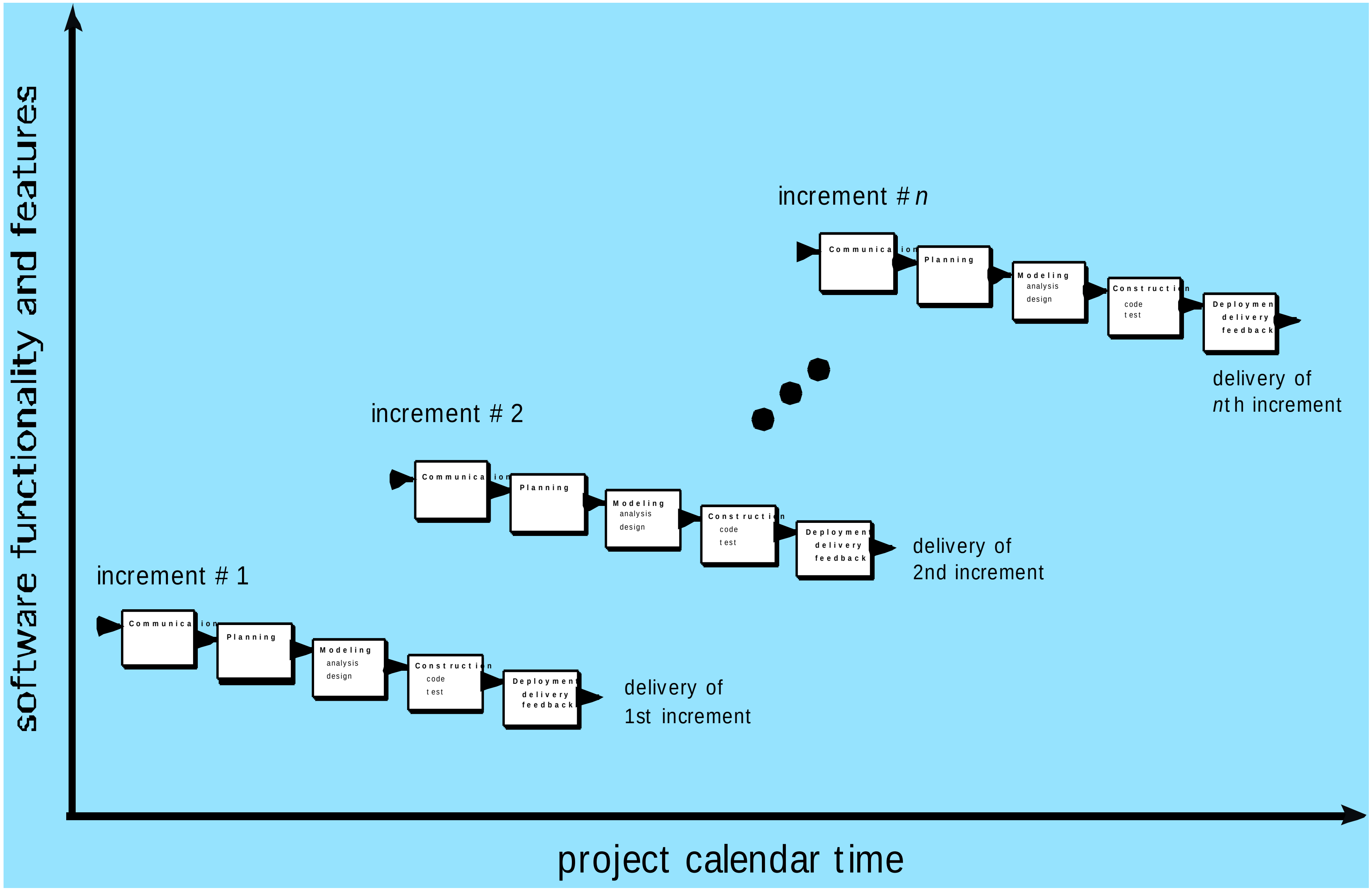
The Waterfall Model Revisited



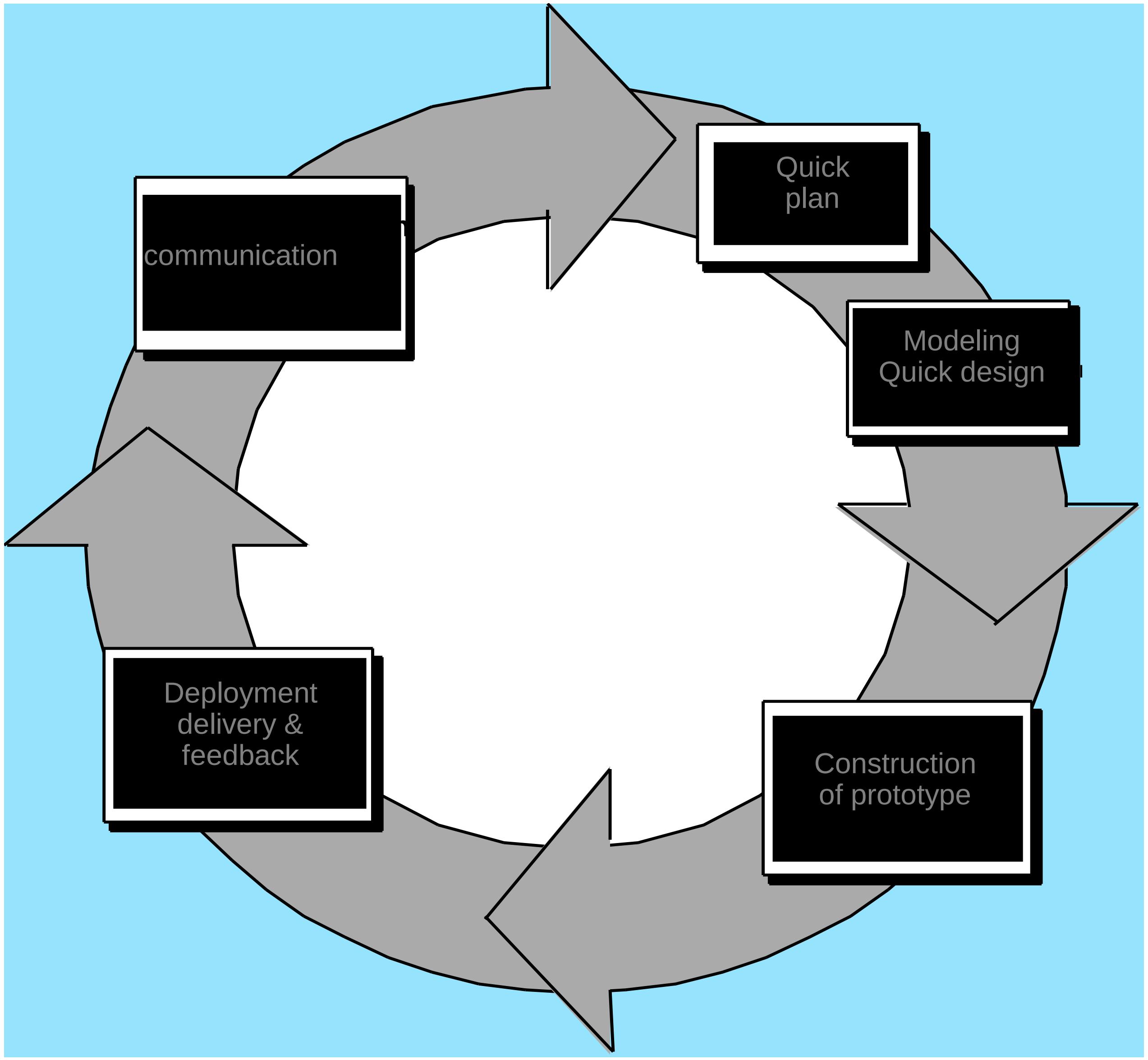
The V-Model


Requirements modeling
Architectural Design
Component Design
Code Generation
Acceptance Testing
System Testing
Integration Testing
Unit Testing
Executable software

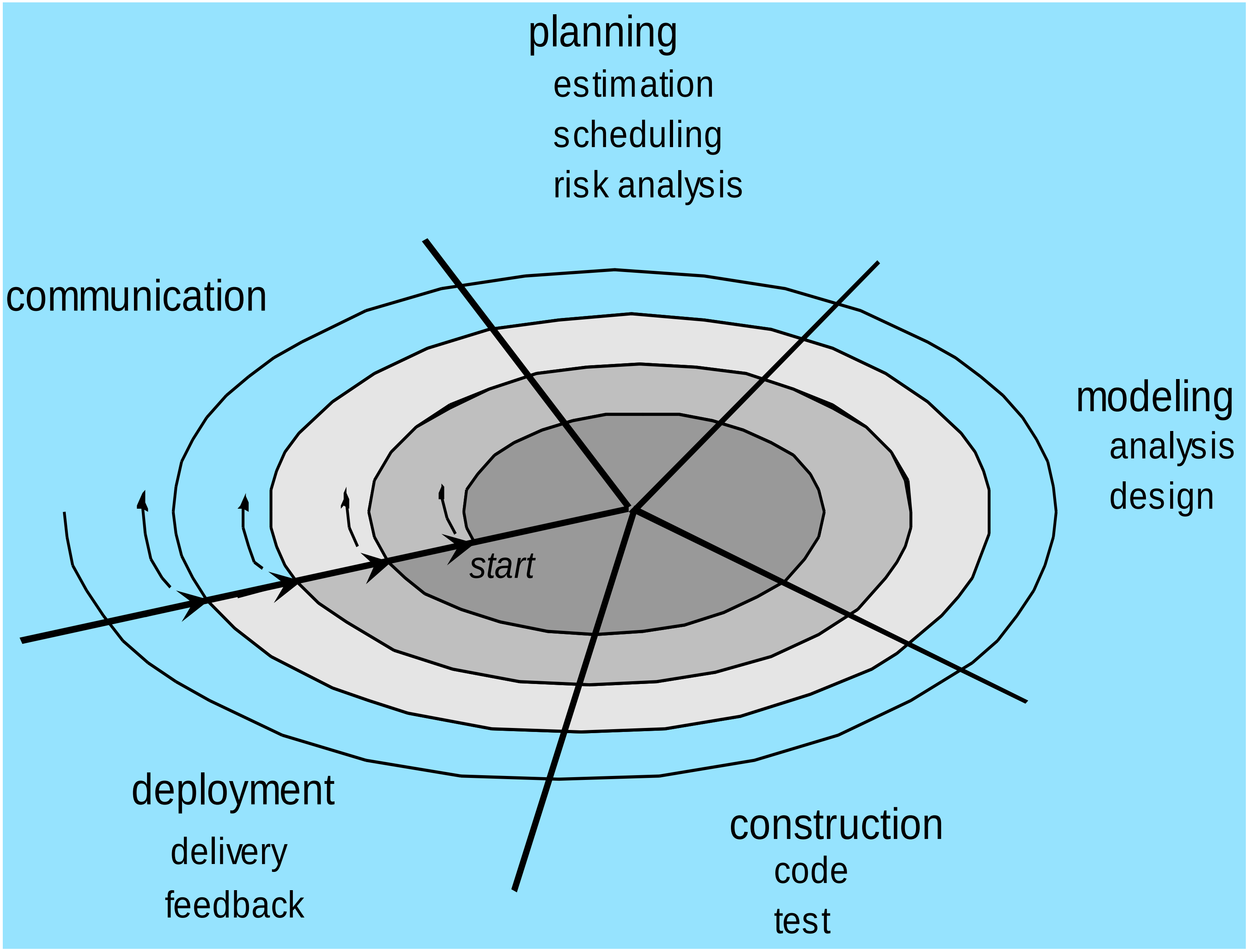
The Incremental Model



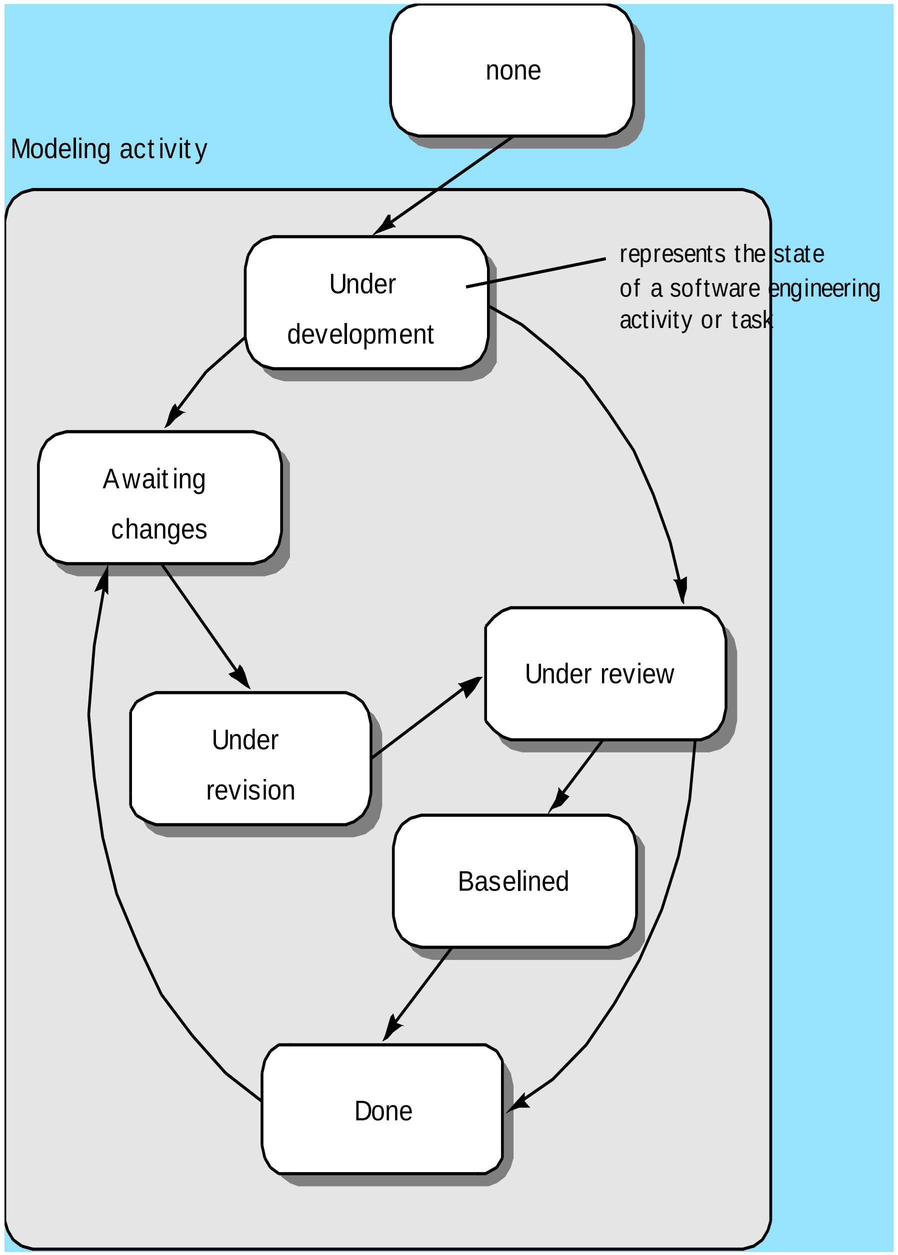
Evolutionary Models: Prototyping



Evolutionary Models: The Spiral Model Revisited



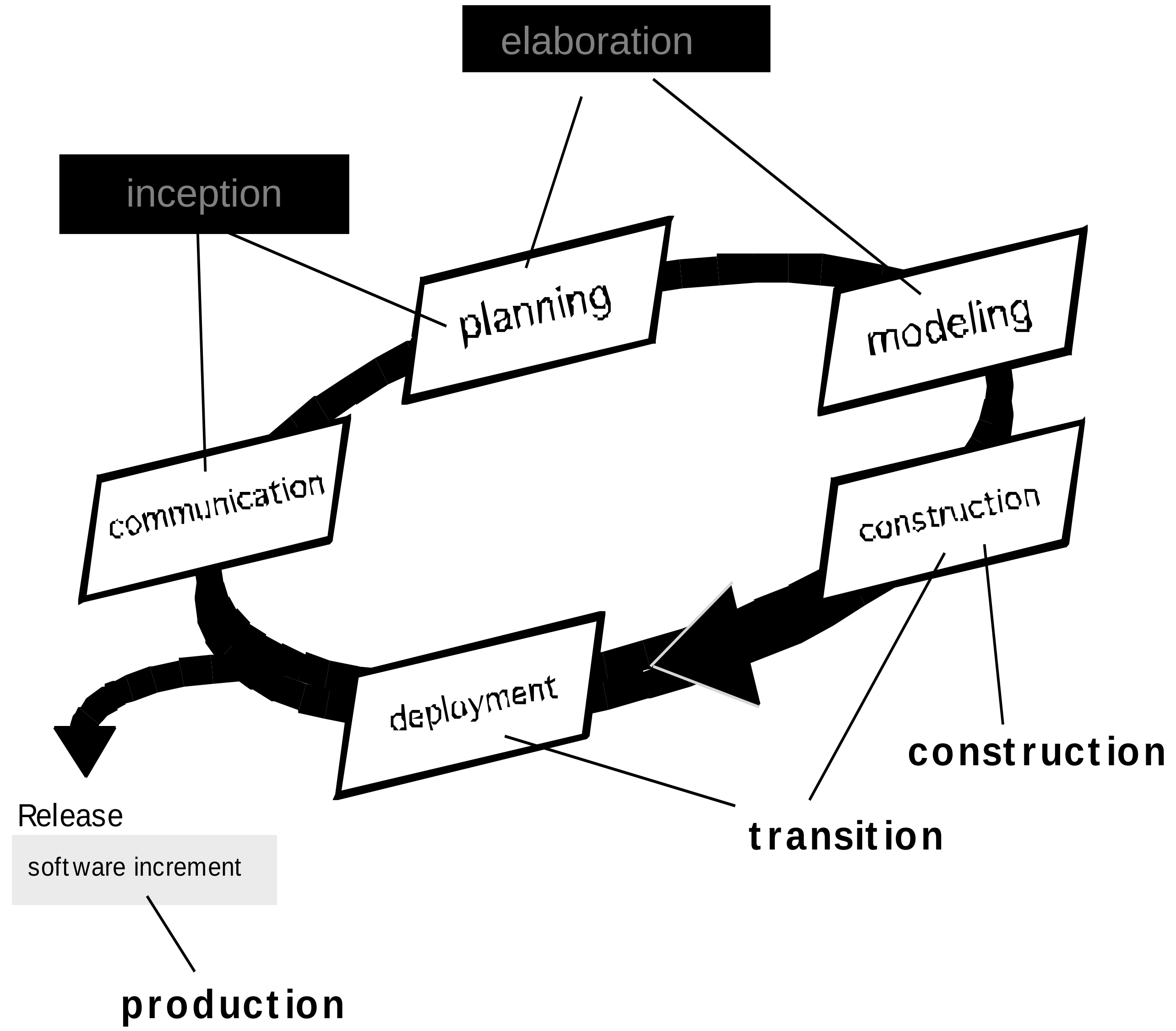
Evolutionary Models: Concurrent



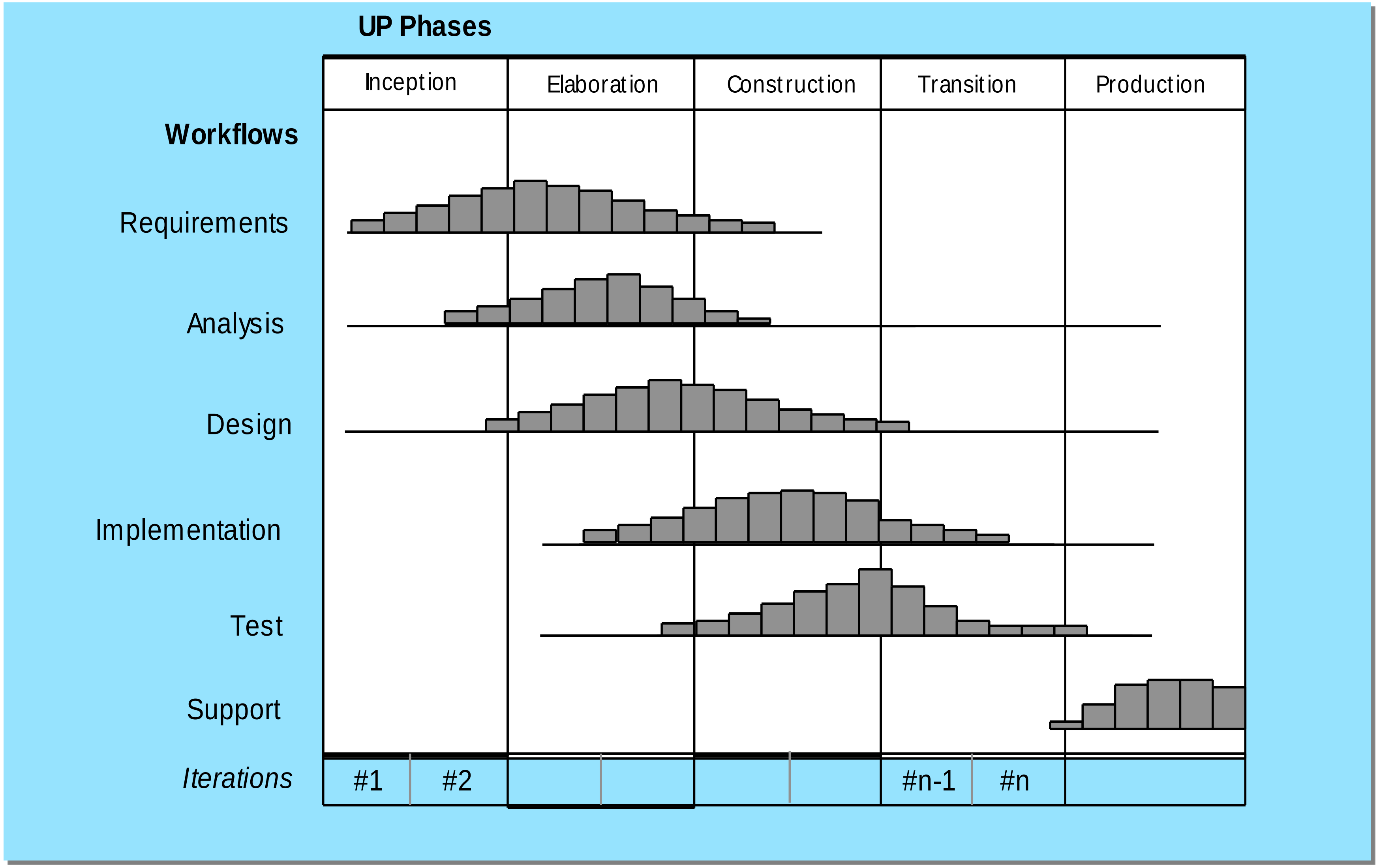
Still Other Process Models

- Component based development—the process to apply when reuse is a development objective
- Formal methods—emphasizes the mathematical specification of requirements
- AOSD—provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*
- Unified Process—a “use-case driven, architecture-centric, iterative and incremental” software process closely aligned with the Unified Modeling Language (UML)

The Unified Process (UP)



UP Phases


UP Phases
Inception
Elaboration
Construction
Transition
Production
Workflows
Requirements
Analysis
Design
Implementation
Test
Support
Iterations
#1
#2
#n-1
#n

UP Work Products

Inception phase

Vision document
Initial use-case model
Initial project glossary
Initial business case
Initial risk assessment.
Project plan,
 phases and iterations.
Business model,
 if necessary.
One or more prototypes

Elaboration phase

Use-case model
Supplementary requirements
 including non-functional
Analysis model
Software architecture
 Description.
Executable architectural
 prototype.
Preliminary design model
Revised risk list
Project plan including
 iteration plan
 adapted workflows
 milestones
 technical work products
Preliminary user manual

Construction phase

Design model
Software components
Integrated software
 increment
Test plan and procedure
Test cases
Support documentation
 user manuals
 installation manuals
 description of current
  increment

Transition phase

Delivered software increment
Beta test reports
General user feedback

Personal Software Process (PSP)

- **Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.
- **High-level design.** External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.
- **High-level design review.** Formal verification methods (Chapter 21 of the course textbook) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.
- **Development.** The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.
- **Postmortem.** Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

Team Software Process (TSP)

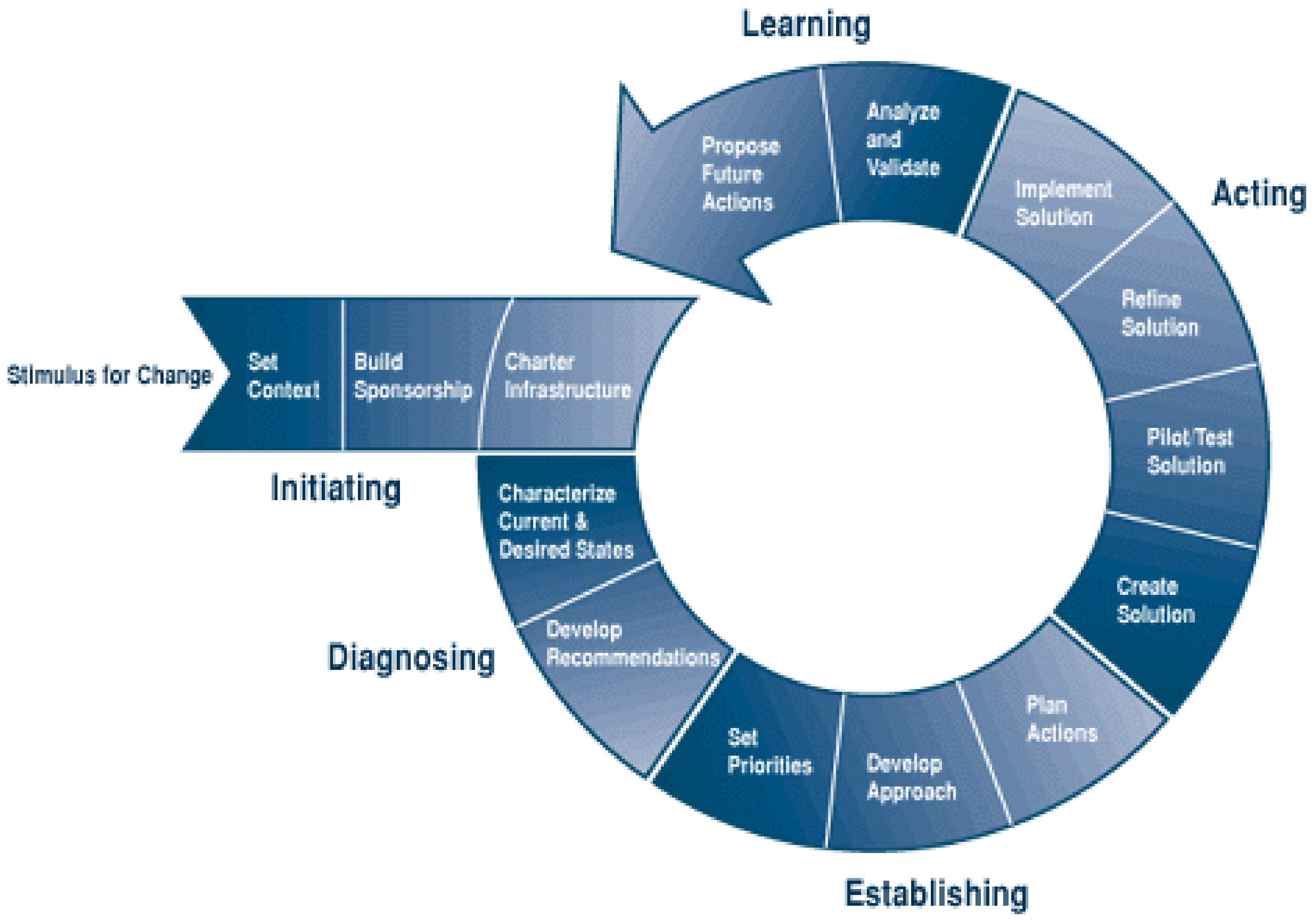
- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPT) of three to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
  - The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 30 of the course textbook.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

CMM & PSP/TSP (http://www.sei.cmu.edu)

- The Capability Maturity Model for Software (SW-CMM) is used by organizations to guide their software process improvement efforts
- Personal Software Process
  - http://www.sei.cmu.edu/tsp/psp.html
- The Team Software Process (TSP) was designed to implement effective, high-maturity processes for project teams
- If all projects in an organization are using the TSP, does the organization exhibit the characteristics of high process maturity, as described in the SW-CMM?
  - http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr008.pdf

SEI's IDEALModel

- IDEAL is an organizational improvement model



Team Exercise #1 – Groups will present in class

- Research and put together a comparative write-up or traditional Process Models:
  - Waterfall
  - V
  - Phased
  - Evolutionary
  - Spiral
  - CBSE
  - RUP
  - PSP/TSP

Agenda – Software Engineering Lifecycles (SDLCs)

The Manifesto for Agile Software Development

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- ***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."

Kent Beck et al

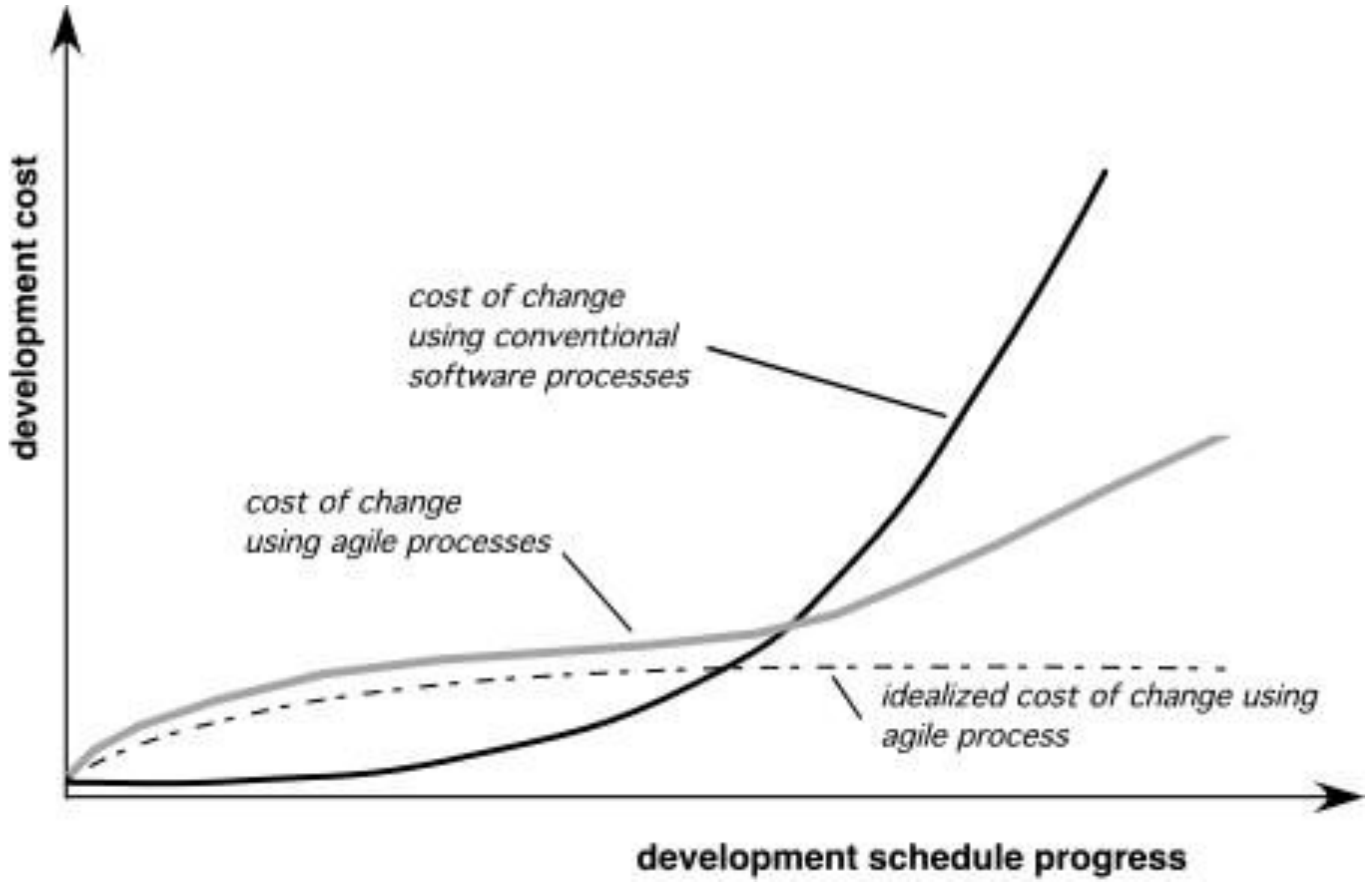
What is “Agility”?

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed

Yielding …

- Rapid, incremental delivery of software

Agility and the Cost of Change



An Agile Process

- Is driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple ‘software increments’
- Adapts as changes occur

Agility Principles (1/2)

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 timescale.
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.

Agility Principles (2/2)

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.

Human Factors Revisited

- *the process molds to the needs of the people and team*, not the other way around
- key traits must exist among the people on an agile team and the team itself:
  - » **Competence.**
  - » **Common focus.**
  - » **Collaboration.**
  - » **Decision-making ability.**
  - » **Fuzzy problem-solving ability.**
  - » **Mutual trust and respect.**
  - » **Self-organization.**

eXtreme Programming (XP) - http://www.extremeprogramming.org/

- A lightweight software methodology
  - Few rules and practices or ones which are easy to follow
  - Emphasizes customer involvement and promotes team work
  - See XP’s rules and practices at http://www.extremeprogramming.org/rules.html

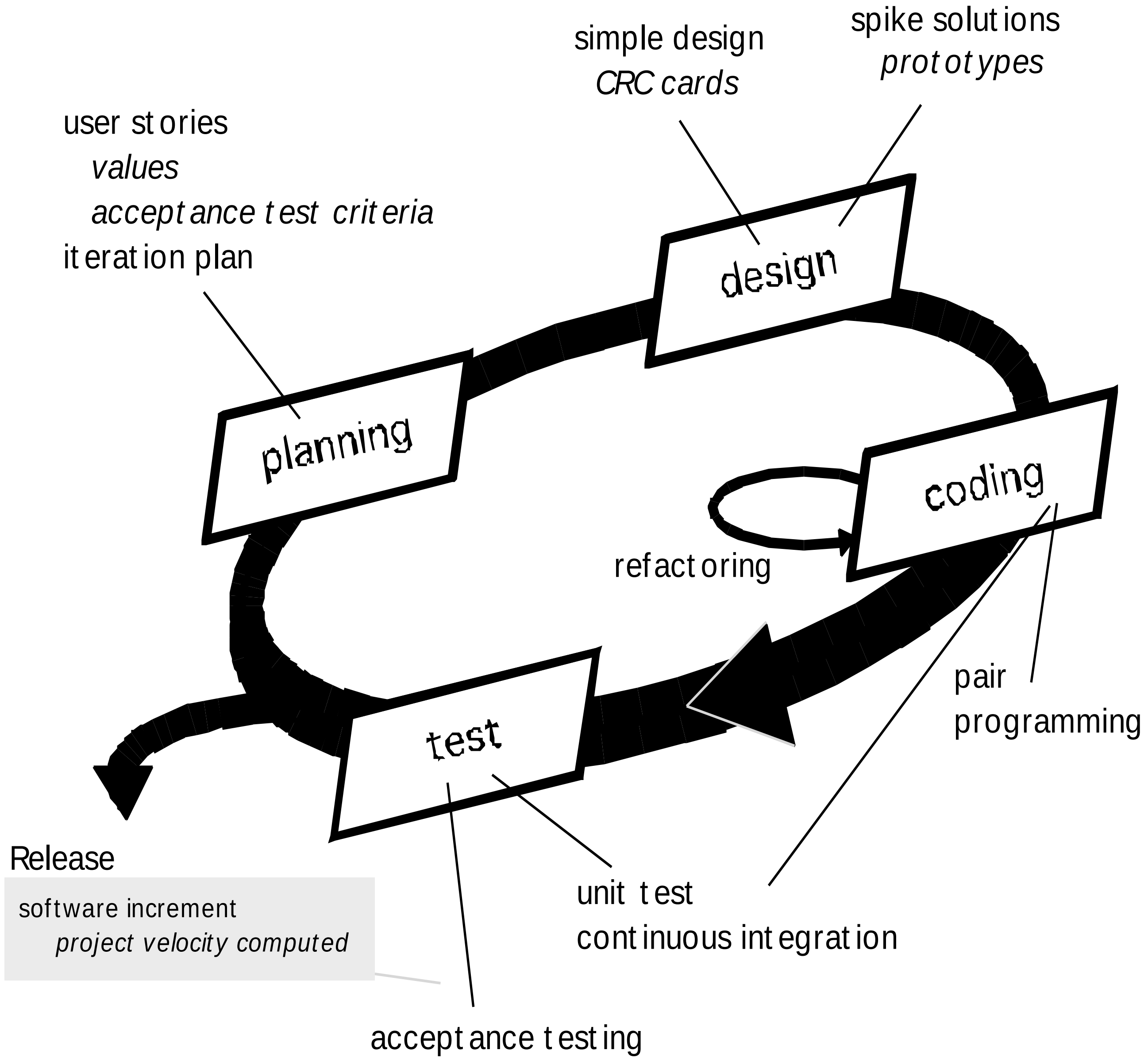
Extreme Programming (XP) (1/3)

- The most widely used agile process, originally proposed by Kent Beck
- XP Planning
  - Begins with the creation of “user stories”
  - Agile team assesses each story and assigns a cost
  - Stories are grouped to for a deliverable increment
  - A commitment is made on delivery date
  - After the first increment “project velocity” is used to help define subsequent delivery dates for other increments

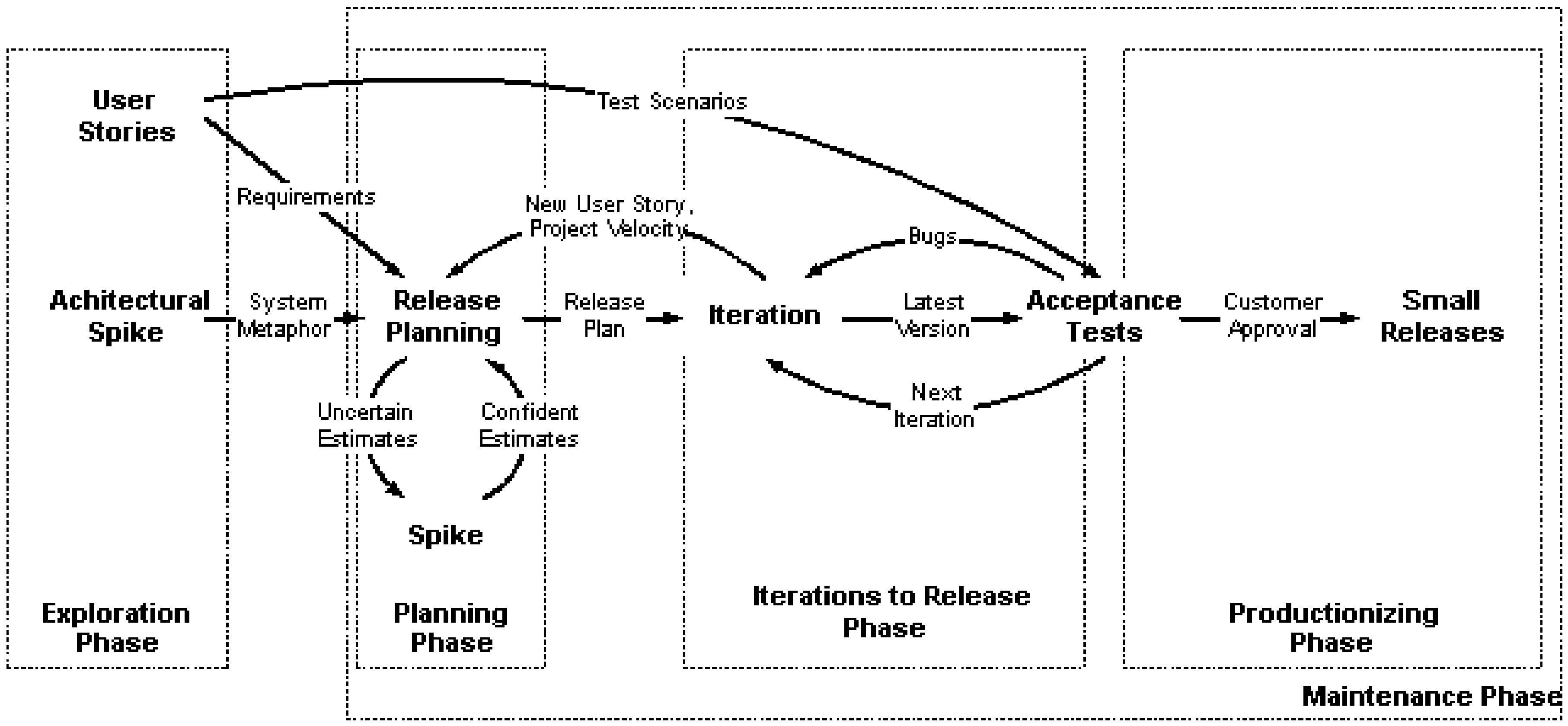
Extreme Programming (XP) (2/3)

- XP Design
  - Follows the KISS principle
  - Encourage the use of CRC cards (see textbook ch 8)
  - For difficult design problems, suggests the creation of "spike solutions"—a design prototype
  - Encourages "refactoring"—an iterative refinement of the internal program design
- XP Coding
  - Recommends the construction of a unit test for a store *before* coding commences
  - Encourages "pair programming"
- XP Testing
  - All unit tests are executed daily
  - "Acceptance tests" are defined by the customer and excuted to assess customer visible functionality

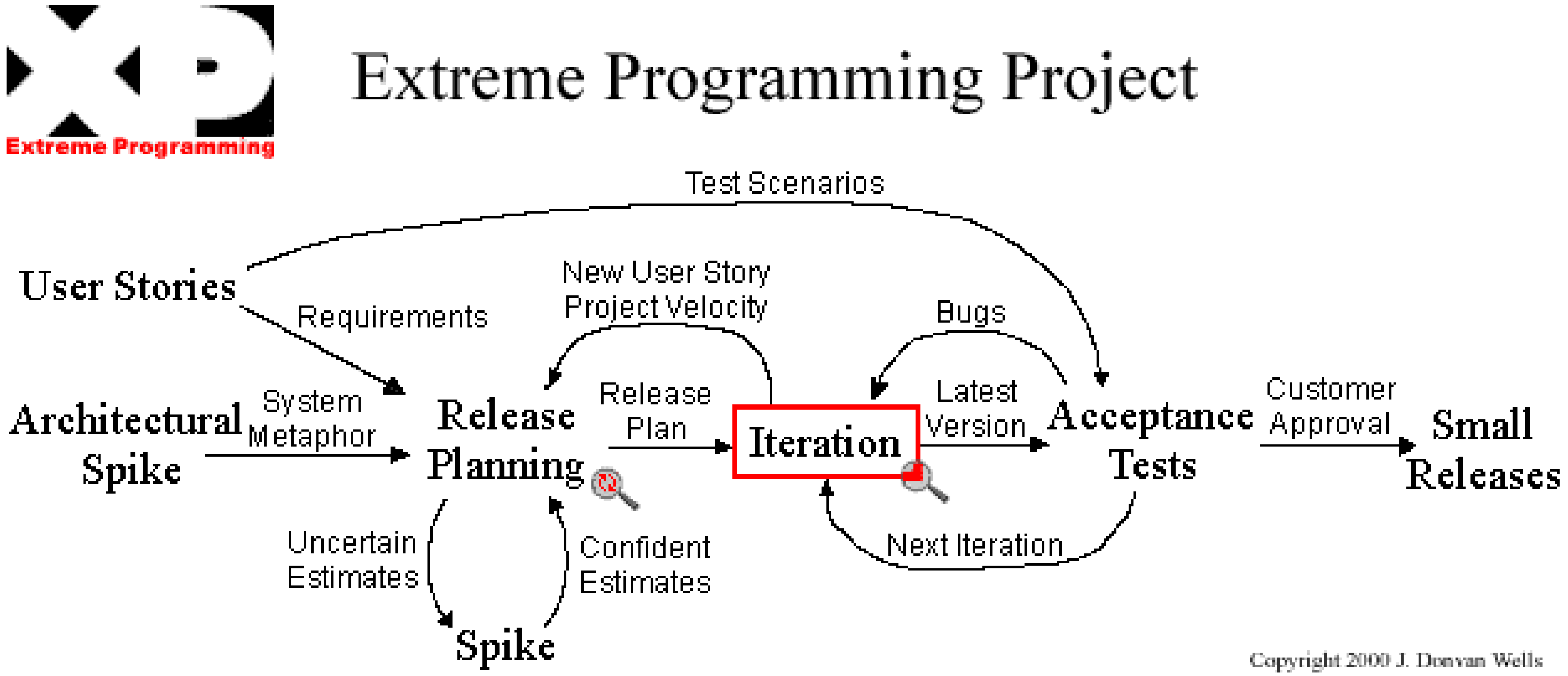
Extreme Programming (XP) (3/3)



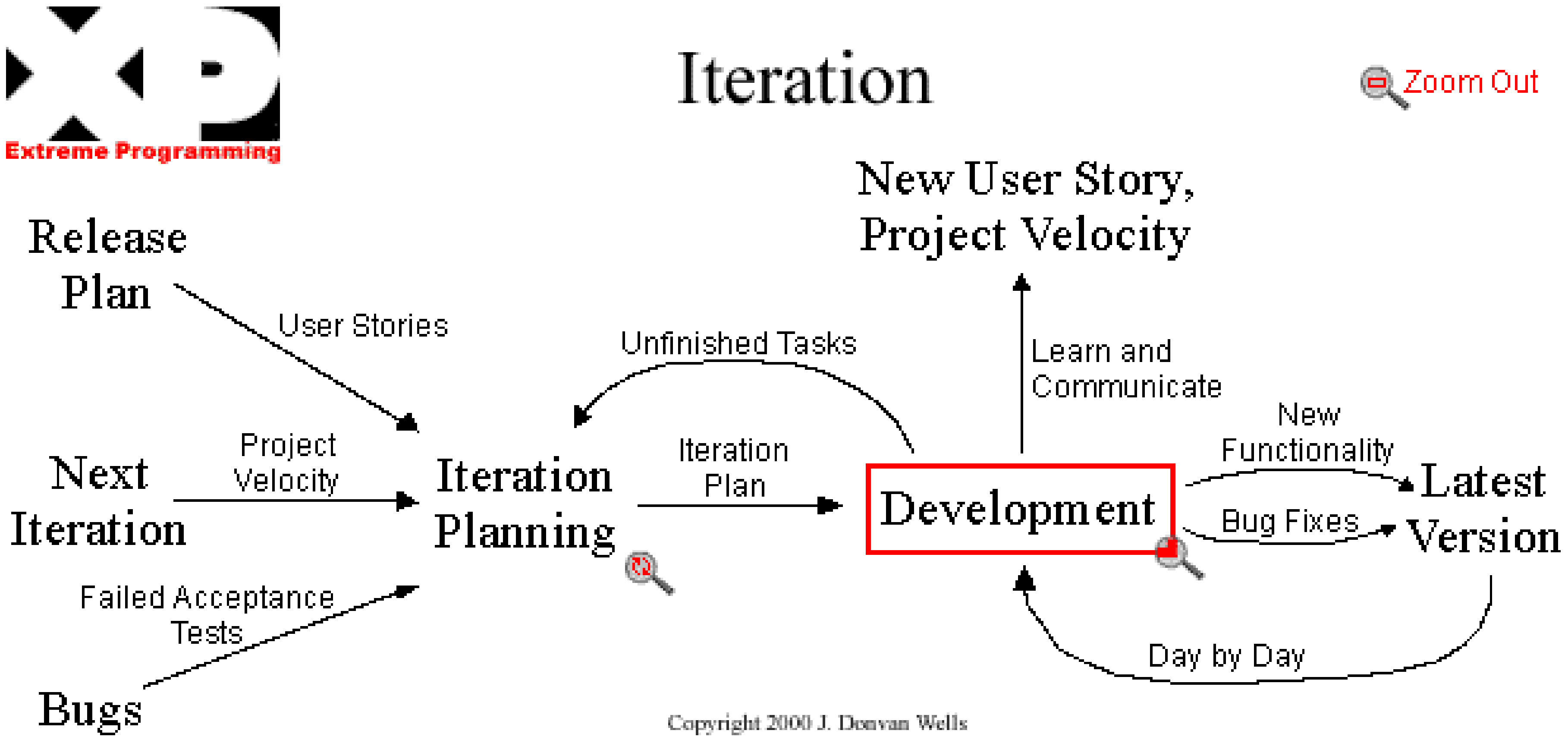
XP Project Lifecycle



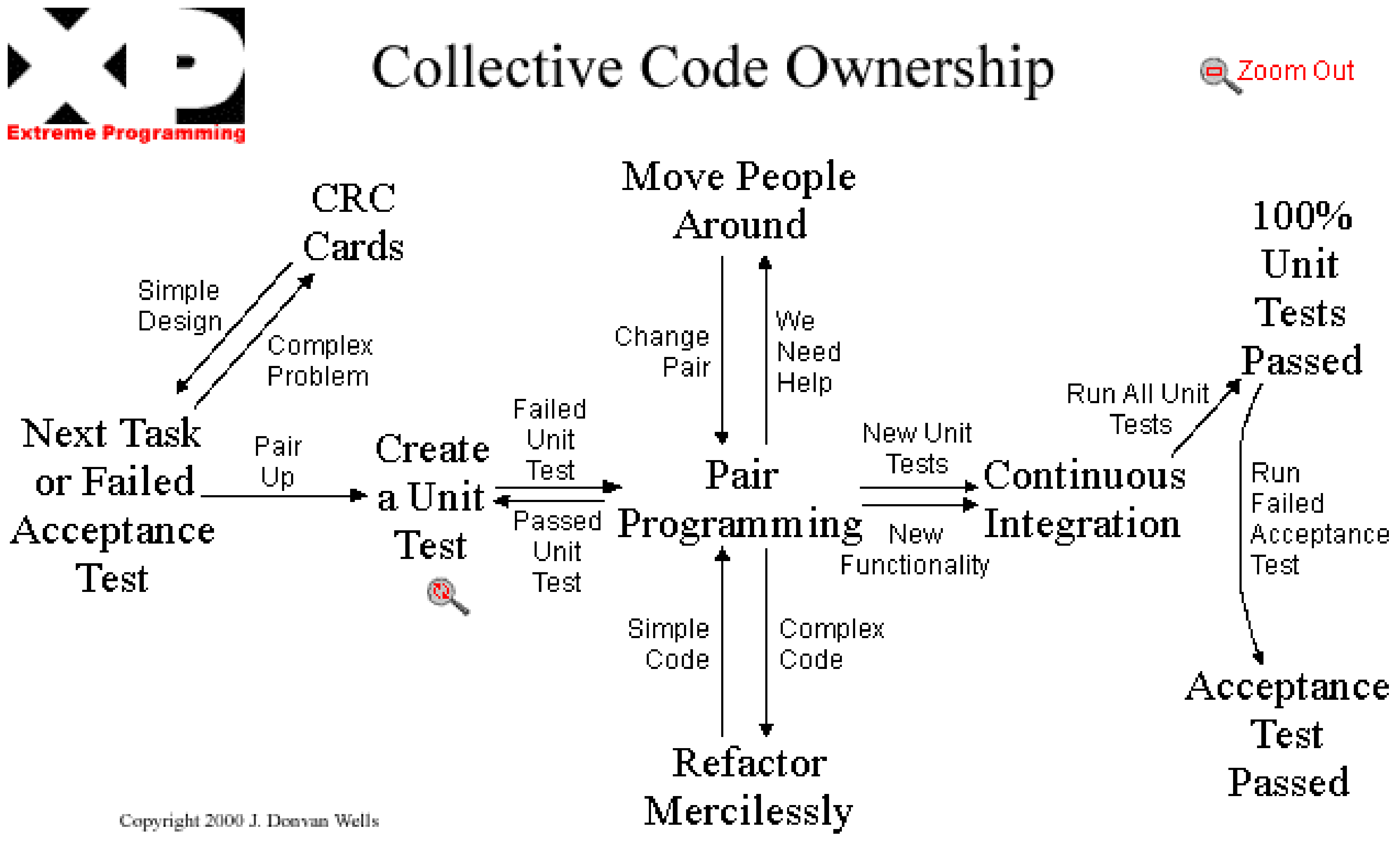
eXtreme Programming (XP) Map



XP Iteration Planning



XP Unit Testing



Agile Modeling & XP Summarized

- Practices-based software process whose scope is to describe how to model and document in an effective and "agile" manner
- One goal is to address the issue of how to apply modeling techniques on software projects taking an agile approach such as:
  - eXtreme Programming (XP)
  - Dynamic Systems Development Method (DSDM)
  - SCRUM
  - etc.
- Using modeling throughout the XP lifecycle
  - http://www.agilemodeling.com/essays/agileModelingXPLifecycle.htm
- Additional information
  - http://www.agilemodeling.com/
  - http://www.agilemodeling.com/resources.htm

Adaptive Software Development (1/2)

- Originally proposed by Jim Highsmith
- ASD — distinguishing features
  - Mission-driven planning
  - Component-based focus
  - Uses “time-boxing” (See textbook Chapter 24)
  - Explicit consideration of risks
  - Emphasizes collaboration for requirements gathering
  - Emphasizes “learning” throughout the process

Adaptive Software Development (2/2)

adaptive cycle planning

uses mission statement

project constraints

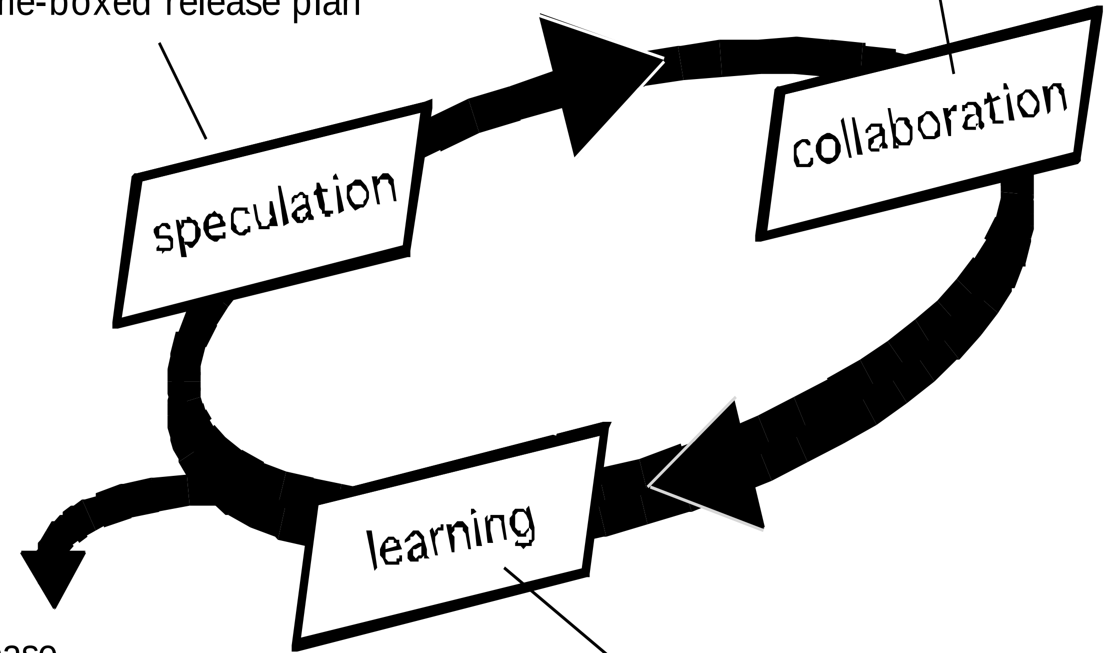
basic requirements

time-boxed release plan

Requirements gathering

JAD

mini-specs



Release

software increment

adjustments for subsequent cycles

components implemented/tested

focus groups for feedback

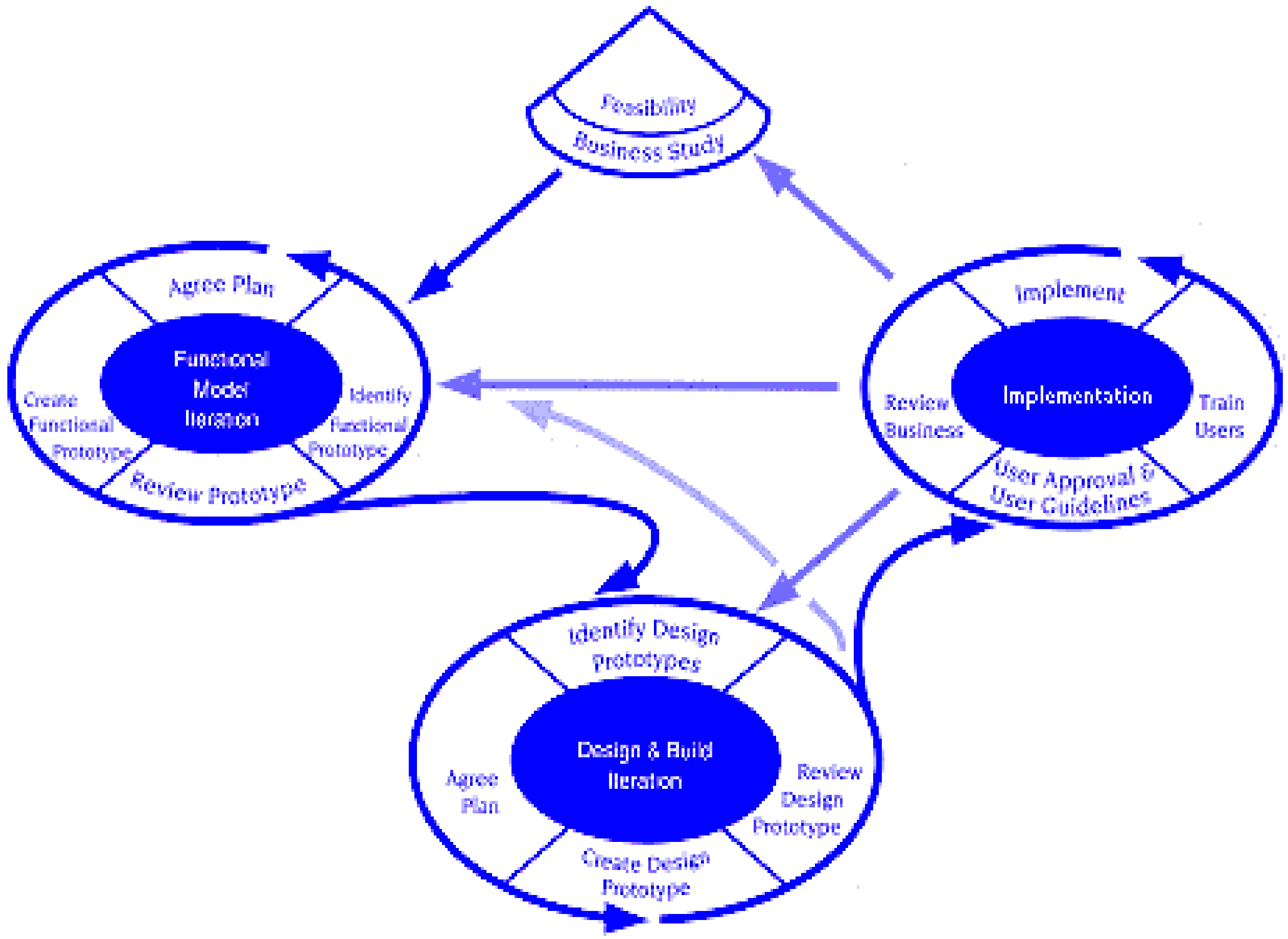
formal technical reviews

postmortems

Dynamic Systems Development Method (1/2)

- Promoted by the DSDM Consortium (www.dsdm.org)
- DSDM—distinguishing features
  - Similar in most respects to XP and/or ASD
  - Nine guiding principles
    - Active user involvement is imperative.
    - DSDM teams must be empowered to make decisions.
    - The focus is on frequent delivery of products.
    - Fitness for business purpose is the essential criterion for acceptance of deliverables.
    - Iterative and incremental development is necessary to converge on an accurate business solution.
    - All changes during development are reversible.
    - Requirements are baselined at a high level
    - Testing is integrated throughout the life-cycle.

Dynamic Systems Development Method (2/2)

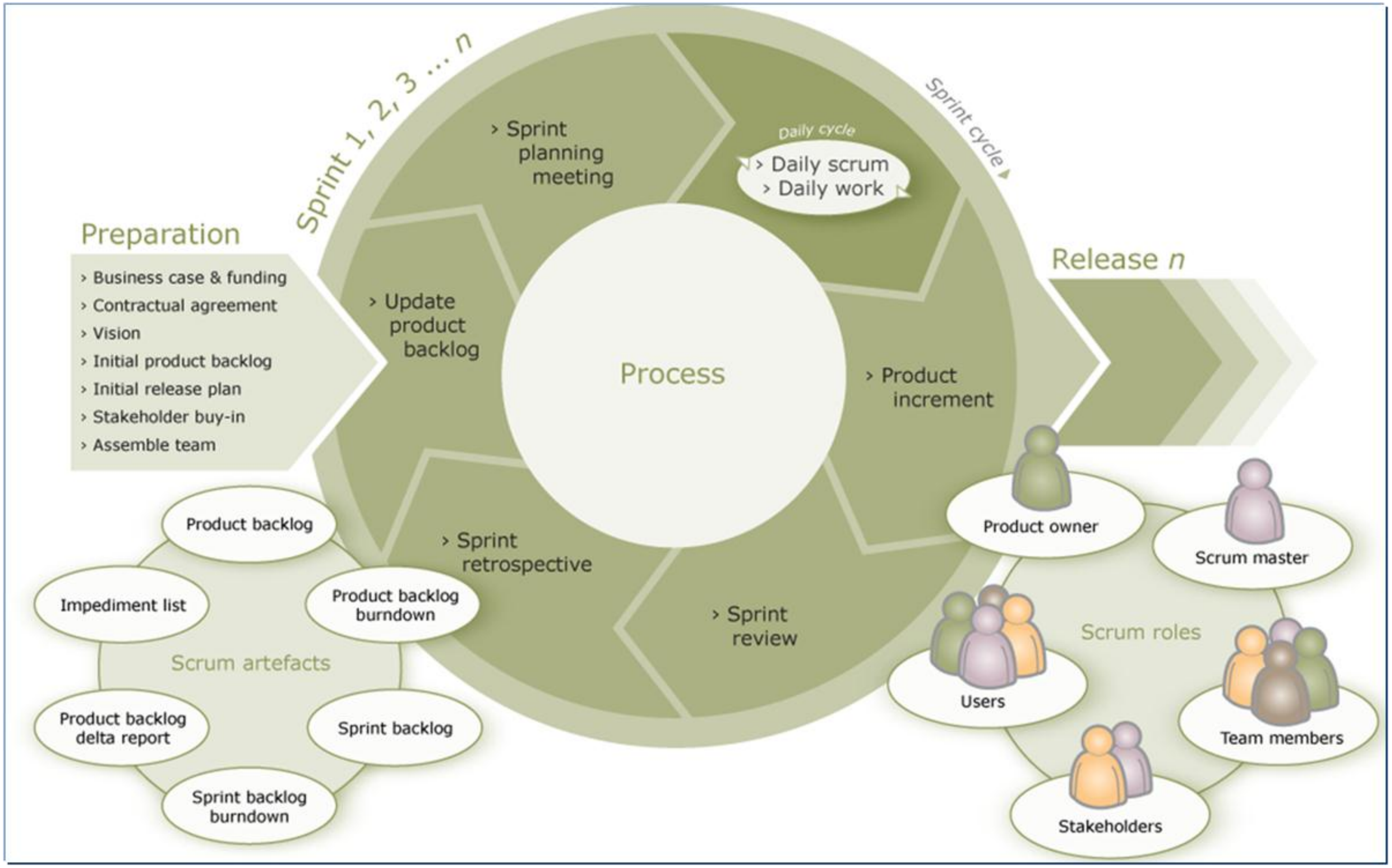


DSDM Life Cycle (with permission of the DSDM consortium)

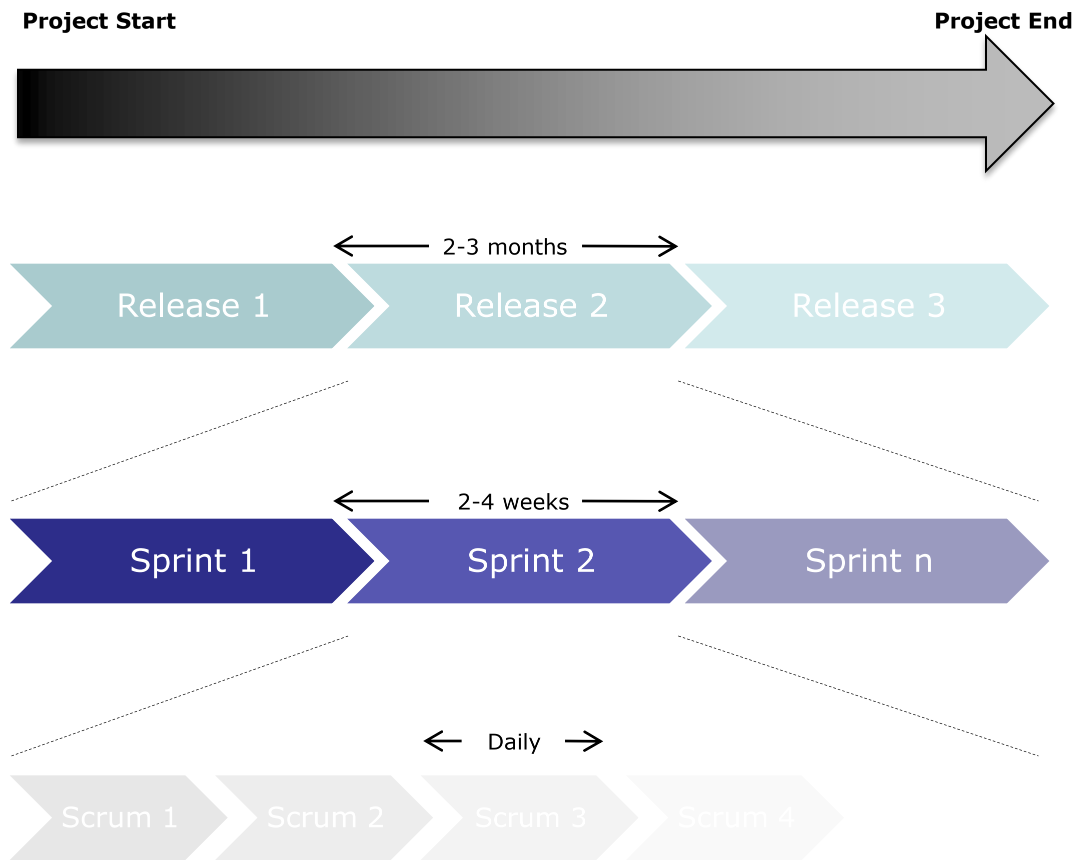
Scrum

- Originally proposed by Schwaber and Beedle
- Scrum—distinguishing features
  - Development work is partitioned into "packets"
  - Testing and documentation are on-going as the product is constructed
  - Work occurs in "sprints" and is derived from a "backlog" of existing requirements
  - Meetings are very short and sometimes conducted without chairs
  - "demos" are delivered to the customer with the time-box allocated

Scrum Process Overview



Scrum Timeline Breakdown



- Agile based projects are broken down into Releases
- At the end of each Release, a working version of the product is delivered to the Business Users
- Each Release is made up of a number of sprints
- At the end of each Sprint, a working version of the product is deployed with incremental functionalities added over the previous sprint
- During each Sprint, daily scrum calls are held where status update is provided by the team

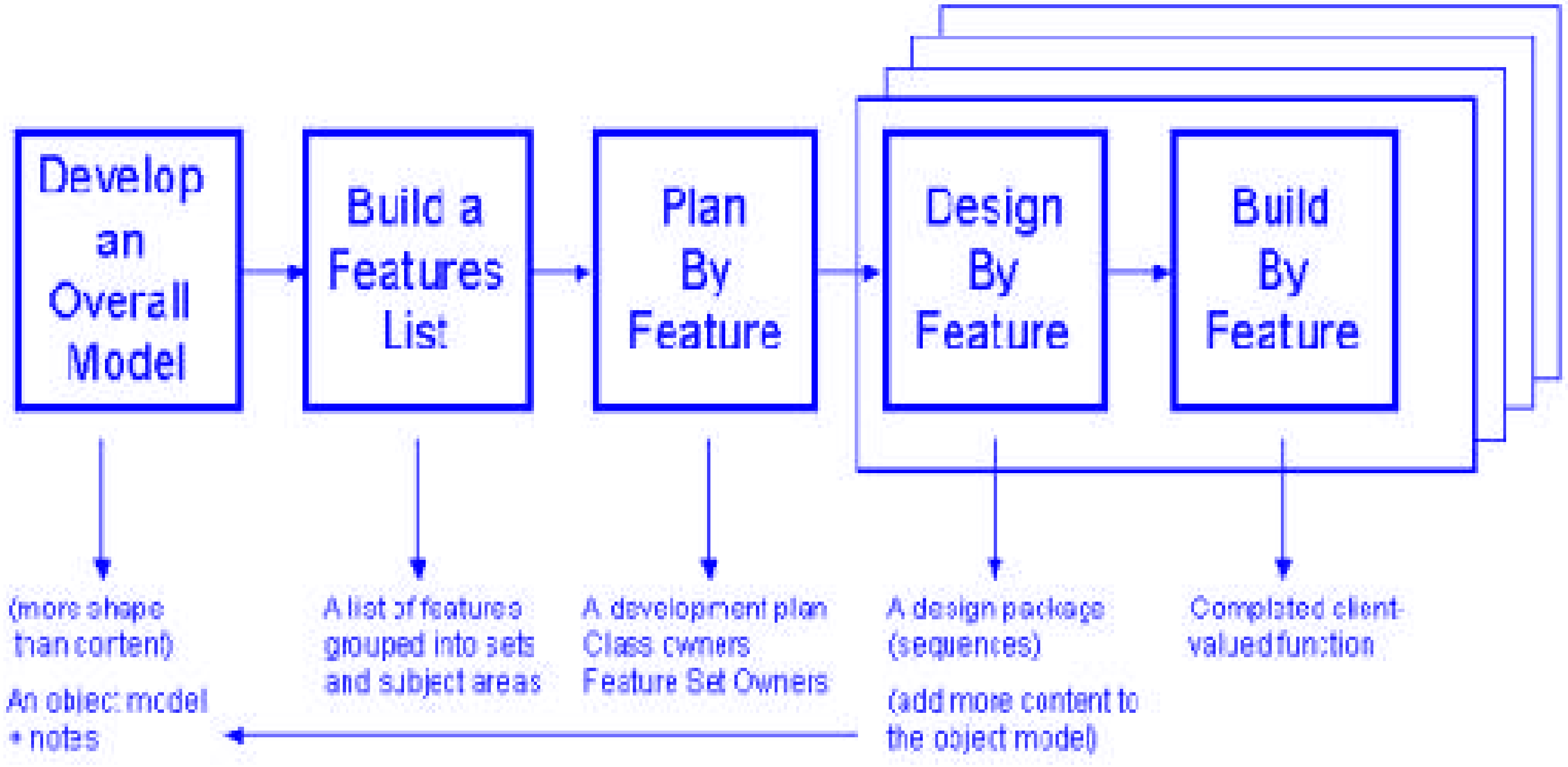
Crystal

- Proposed by Cockburn and Highsmith
- Crystal—distinguishing features
  - Actually a family of process models that allow “maneuverability” based on problem characteristics
  - Face-to-face communication is emphasized
  - Suggests the use of “reflection workshops” to review the work habits of the team

Feature Driven Development (1/2)

- Originally proposed by Peter Coad et al
- FDD—distinguishing features
  - Emphasis is on defining “features”
    - a *feature* “is a client-valued function that can be implemented in two weeks or less.”
  - Uses a feature template
    - <action> the <result> <by | for | of | to> a(n) <object>
  - A features list is created and “plan by feature” is conducted
  - Design and construction merge in FDD

Feature Driven Development (2/2)



Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
An object model + notes
A list of features grouped into sets and subject areas
A development plan
Class owners
Feature Set Owners
A design package (sequences)
(add more content to the object model)
Completed client-valued function

Reprinted with permission of Peter Coad

Agile Modeling

- Originally proposed by Scott Ambler
- Suggests a set of agile modeling principles
  - Model with a purpose
  - Use multiple models
  - Travel light
  - Content is more important than representation
  - Know the models and the tools you use to create them
  - Adapt locally

Agenda – Software Engineering Lifecycles (SDLCs)

Software Engineering Knowledge

- *You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge—a kind that I think of as "software engineering principles"—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.*

Steve McConnell

Principles that Guide Process - I

- **Principle #1. *Be agile.*** Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach.
- **Principle #2. *Focus on quality at every step.*** The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.
- **Principle #3. *Be ready to adapt.*** Process is not a religious experience and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.
- **Principle #4. *Build an effective team.*** Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.

Principles that Guide Process - II

- **Principle #5. Establish mechanisms for communication and coordination.** Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product.
- **Principle #6. Manage change.** The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved and implemented.
- **Principle #7. Assess risk.** Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans.
- **Principle #8. Create work products that provide value for others.** Create only those work products that provide value for other process activities, actions or tasks.

Principles that Guide Practice

- **Principle #1. *Divide and conquer.*** Stated in a more technical manner, analysis and design should always emphasize *separation of concerns* (SoC).
- **Principle #2. *Understand the use of abstraction.*** At it core, an abstraction is a simplification of some complex element of a system used to communication meaning in a single phrase.
- **Principle #3. Strive for consistency.** A familiar context makes software easier to use.
- **Principle #4. *Focus on the transfer of information.*** Pay special attention to the analysis, design, construction, and testing of interfaces.

Principles that Guide Practice

- **Principle #5. *Build software that exhibits effective modularity.*** Separation of concerns (Principle #1) establishes a philosophy for software. *Modularity* provides a mechanism for realizing the philosophy.
- **Principle #6. *Look for patterns.*** Brad Appleton [App00] suggests that: “The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development.
- **Principle #7. *When possible, represent the problem and its solution from a number of different perspectives.***
- **Principle #8. *Remember that someone will maintain the software.***

Communication Principles

- **Principle #1.** ***Listen.*** Try to focus on the speaker's words, rather than formulating your response to those words.
- **Principle # 2.** ***Prepare before you communicate.*** Spend the time to understand the problem before you meet with others.
- **Principle # 3.** ***Someone should facilitate the activity.*** Every communication meeting should have a leader (a facilitator) (1) to keep the conversation moving in a productive direction; (2) to mediate any conflict that does occur, and (3) to ensure than other principles are followed.
- **Principle #4.** ***Face-to-face communication is best.*** But it usually works better when some other representation of the relevant information is present.

Communication Principles

- **Principle # 5.** ***Take notes and document decisions.*** Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.
- **Principle # 6.** ***Strive for collaboration.*** Collaboration and consensus occur when the collective knowledge of members of the team is combined …
- **Principle # 7.** ***Stay focused, modularize your discussion.*** The more people involved in any communication, the more likely that discussion will bounce from one topic to the next.
- **Principle # 8.** ***If something is unclear, draw a picture.***
- **Principle # 9.** ***(a) Once you agree to something, move on; (b) If you can't agree to something, move on; (c) If a feature or function is unclear and cannot be clarified at the moment, move on.***
- **Principle # 10.** ***Negotiation is not a contest or a game. It works best when both parties win.***

Planning Principles

- **Principle #1.** ***Understand the scope of the project.*** It's impossible to use a roadmap if you don't know where you're going. Scope provides the software team with a destination.
- **Principle #2.** ***Involve the customer in the planning activity.*** The customer defines priorities and establishes project constraints.
- **Principle #3.** ***Recognize that planning is iterative.*** A project plan is never engraved in stone. As work begins, it very likely that things will change.
- **Principle #4.** ***Estimate based on what you know.*** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.

Planning Principles

- **Principle #5.** ***Consider risk as you define the plan.*** If you have identified risks that have high impact and high probability, contingency planning is necessary.
- **Principle #6.** ***Be realistic.*** People don't work 100 percent of every day.
- **Principle #7.** ***Adjust granularity as you define the plan.*** *Granularity* refers to the level of detail that is introduced as a project plan is developed.
- **Principle #8.** ***Define how you intend to ensure quality.*** The plan should identify how the software team intends to ensure quality.
- **Principle #9.** ***Describe how you intend to accommodate change.*** Even the best planning can be obviated by uncontrolled change.
- **Principle #10.** ***Track the plan frequently and make adjustments as required.*** Software projects fall behind schedule one day at a time.

Modeling Principles

- In software engineering work, two classes of models can be created:
  - ***Requirements models* (also called *analysis models*)** represent the customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain.
  - ***Design models*** represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Requirements Modeling Principles

- **Principle #1.** ***The information domain of a problem must be represented and understood.***
- **Principle #2.** ***The functions that the software performs must be defined.***
- **Principle #3.** ***The behavior of the software (as a consequence of external events) must be represented.***
- **Principle #4.** ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.***
- **Principle #5.** ***The analysis task should move from essential information toward implementation detail.***

Design Modeling Principles

- **Principle #1.** ***Design should be traceable to the requirements model.***
- **Principle #2.** ***Always consider the architecture of the system to be built.***
- **Principle #3.** ***Design of data is as important as design of processing functions.***
- **Principle #5.** ***User interface design should be tuned to the needs of the end-user. However, in every case, it should stress ease of use.***
- **Principle #6.** ***Component-level design should be functionally independent.***
- **Principle #7.** ***Components should be loosely coupled to one another and to the external environment.***
- **Principle #8.** ***Design representations (models) should be easily understandable.***
- **Principle #9.** ***The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity.***

Agile Modeling Principles

- **Principle #1.** ***The primary goal of the software team is to build software, not create models.***
- **Principle #2.** ***Travel light—don’t create more models than you need.***
- **Principle #3.** ***Strive to produce the simplest model that will describe the problem or the software.***
- **Principle #4.** ***Build models in a way that makes them amenable to change.***
- **Principle #5.** ***Be able to state an explicit purpose for each model that is created.***
- **Principle #6.** ***Adapt the models you develop to the system at hand.***
- **Principle #7.** ***Try to build useful models, but forget about building perfect models.***
- **Principle #8.** ***Don’t become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.***
- **Principle #9.** ***If your instincts tell you a model isn’t right even though it seems okay on paper, you probably have reason to be concerned.***
- **Principle #10.** ***Get feedback as soon as you can.***

Construction Principles

- The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end-user.
- Coding principles and concepts are closely aligned programming style, programming languages, and programming methods.
- Testing principles and concepts lead to the design of tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

Preparation Principles

- **Before you write one line of code, be sure you:**
  - Understand of the problem you’re trying to solve.
  - Understand basic design principles and concepts.
  - Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
  - Select a programming environment that provides tools that will make your work easier.
  - Create a set of unit tests that will be applied once the component you code is completed.

Coding Principles

- **As you begin writing code, be sure you:**
  - Constrain your algorithms by following structured programming [Boh00] practice.
  - Consider the use of pair programming
  - Select data structures that will meet the needs of the design.
  - Understand the software architecture and create interfaces that are consistent with it.
  - Keep conditional logic as simple as possible.
  - Create nested loops in a way that makes them easily testable.
  - Select meaningful variable names and follow other local coding standards.
  - Write code that is self-documenting.
  - Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Validation Principles

- **After you’ve completed your first coding pass, be sure you:**
  - Conduct a code walkthrough when appropriate.
  - Perform unit tests and correct errors you’ve uncovered.
  - Refactor the code.

Testing Principles

- Al Davis suggests the following:
  - **Principle #1.** ***All tests should be traceable to customer requirements.***
  - **Principle #2.** ***Tests should be planned long before testing begins.***
  - **Principle #3.** ***The Pareto principle applies to software testing.***
  - **Principle #4.** ***Testing should begin "in the small" and progress toward testing "in the large."***
  - **Principle #5.** ***Exhaustive testing is not possible.***

Deployment Principles

- **Principle #1.** ***Customer expectations for the software must be managed.*** Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately.
- **Principle #2.** ***A complete delivery package should be assembled and tested.***
- **Principle #3.** ***A support regime must be established before the software is delivered.*** An end-user expects responsiveness and accurate information when a question or problem arises.
- **Principle #4.** ***Appropriate instructional materials must be provided to end-users.***
- **Principle #5.** ***Buggy software should be fixed first, delivered later.***

Agenda – Software Engineering Lifecycles (SDLCs)

Process Assessment and Improvement

- **Standard CMMI Assessment Method for Process Improvement (SCAMPI) —** provides a five step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting and learning.
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)—**provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01]
- **SPICE—The SPICE (ISO/IEC15504)** standard defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process. [ISO08]

- **ISO 9001:2000 for Software—**a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies. [Ant06]

Other Standards

- ISO 12207
  - http://www.acm.org/tsc/lifecycle.html
  - http://www.12207.com/
- IEEE Standards for Software Engineering Processes and Specifications
  - http://standards.ieee.org/catalog/olis/se.html
  - http://members.aol.com/kaizensepg/standard.htm

Agenda – Summary and Conclusion

1 **Session Overview**

2 **Software Engineering LifeCycles (SDLCs)**

3 **Summary and Conclusion**

Course Assignments

- Individual Assignments
  - Reports based on case studies / class presentations
- Project-Related Assignments
  - All assignments (other than the individual assessments) will correspond to milestones in the team project.
  - As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consist of inter-related deliverables which are due on a (bi-) weekly basis.
  - There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
  - A sample project description and additional details will be available under handouts on the course Web site

Team Project

- Project Logistics
  - Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
  - Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may <u>not</u> be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

Team Project Approach - Overall

- Document Transformation methodology driven approach
  - Strategy Alignment Elicitation
    - Equivalent to strategic planning
      - i.e., planning at the level of a project set
  - Strategy Alignment Execution
    - Equivalent to project planning + SDLC
      - i.e., planning a the level of individual projects + project implementation
- Build a methodology Wiki & partially implement the enablers
- Apply transformation methodology approach to a sample problem domain for which a business solution must be found
- Final product is a wiki/report that focuses on
  - Methodology / methodology implementation / sample business-driven problem solution

Team Project Approach – Initial Step

- Document sample problem domain and business-driven problem of interest
  - Problem description
  - High-level specification details
  - High-level implementation details
  - Proposed high-level timeline

Assignments & Readings

- Readings
  - » Slides and Handouts posted on the course web site
  - » Textbook: Part One-Chapters 3-4
- Team Project
  - » Team Project proposal (format TBD in class)
- Team Exercise #1
  - » Presentation topic proposal (format TBD in class)
- Project Frameworks Setup (ongoing)
  - » As per reference provided on the course Web site

Next Session: Planning and Managing Requirements

Any Questions?