# Phát triển Hệ thống Thông tin
(systems development life cycle)

**GV**: Đỗ Oanh Cường

Bộ môn Hệ thống thông tin, Khoa CNTT.

Email: cuongdo@tlu.edu.vn

Website: www.cuongdo.info

Life Cycle:

Stages: Planning Analysis Design Implementation

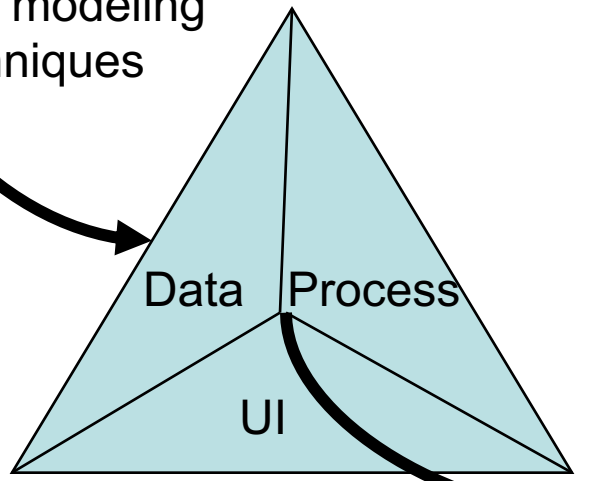
**What**
**Problems/Opportunities**
**Requirements**
**Soft/People Skills**

**How**
**Solutions**
**Specifications**
**Technical Skills**

Visibility: Deliverables/Documentation

Methodology*:
- Process (Life Cycle)
- Techniques (Modeling)

Use modeling techniques

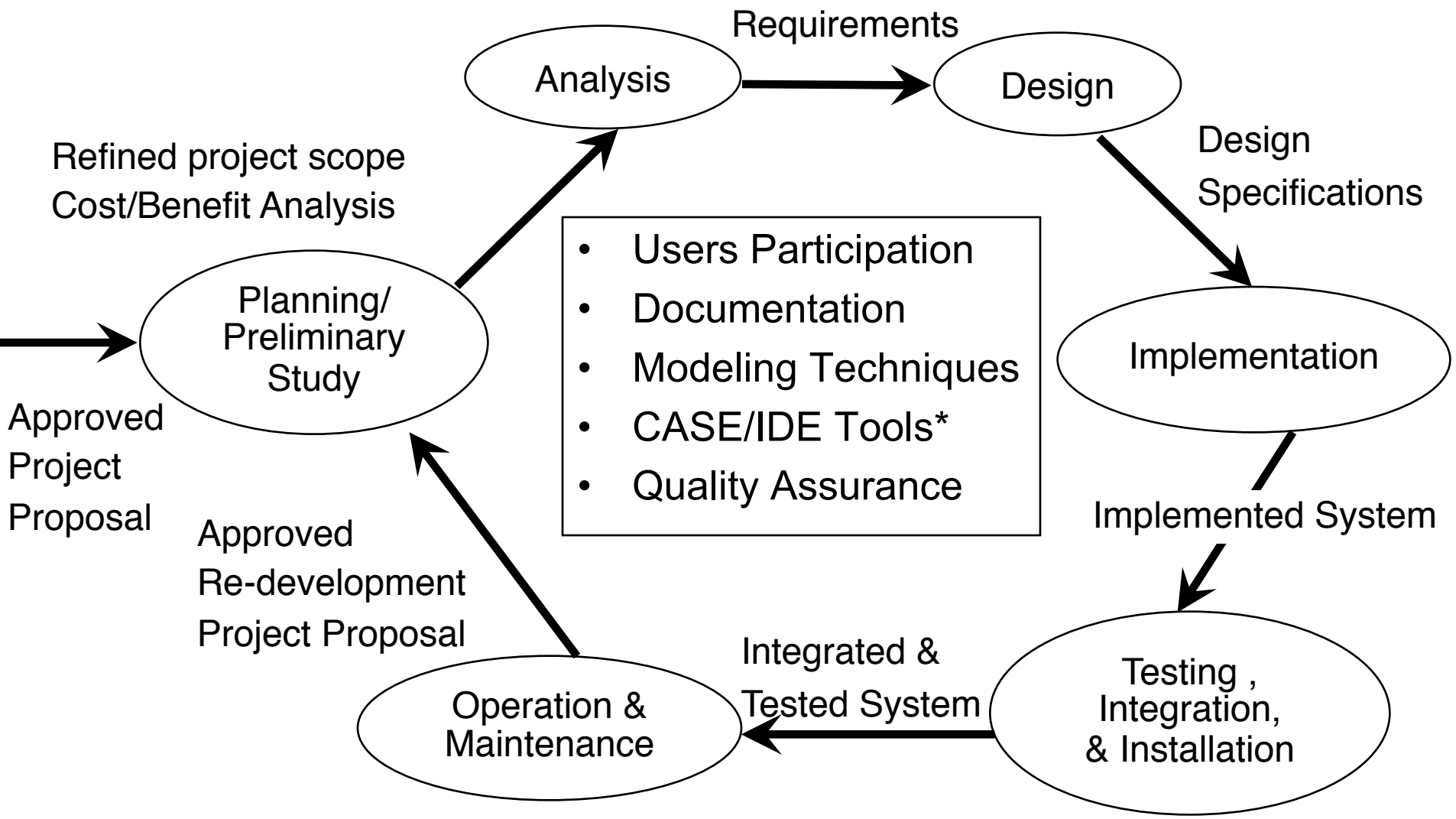



Prototyping
Coding
Programming
Implementation

UI: User Interface

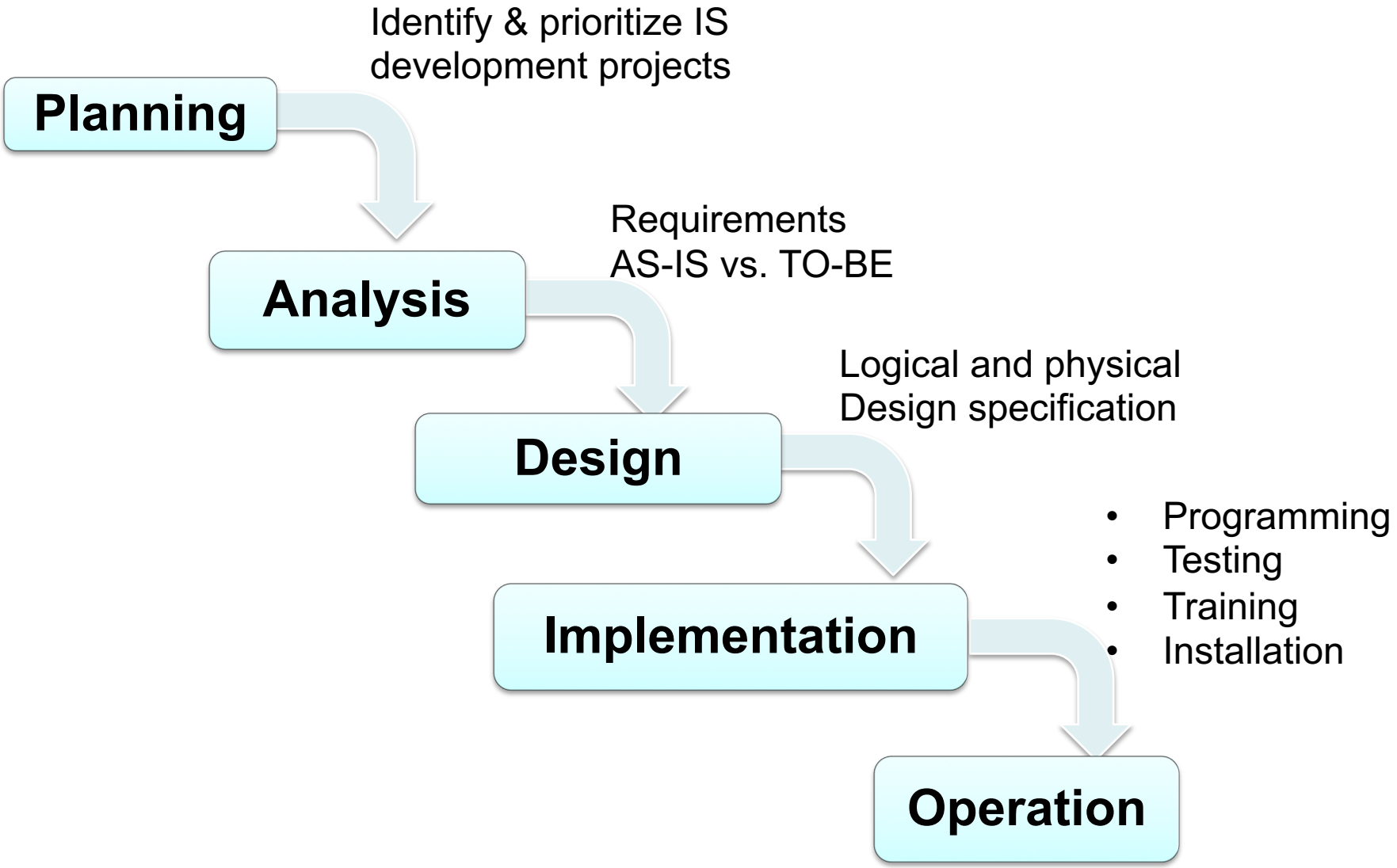
*A **system development methodology** is a framework that is used to structure, plan, and control the **process** of developing an information system.

# Structured Project SDLC




*CASE: Computer-Aided Software Engineering
IDE: Integrated Development Environment

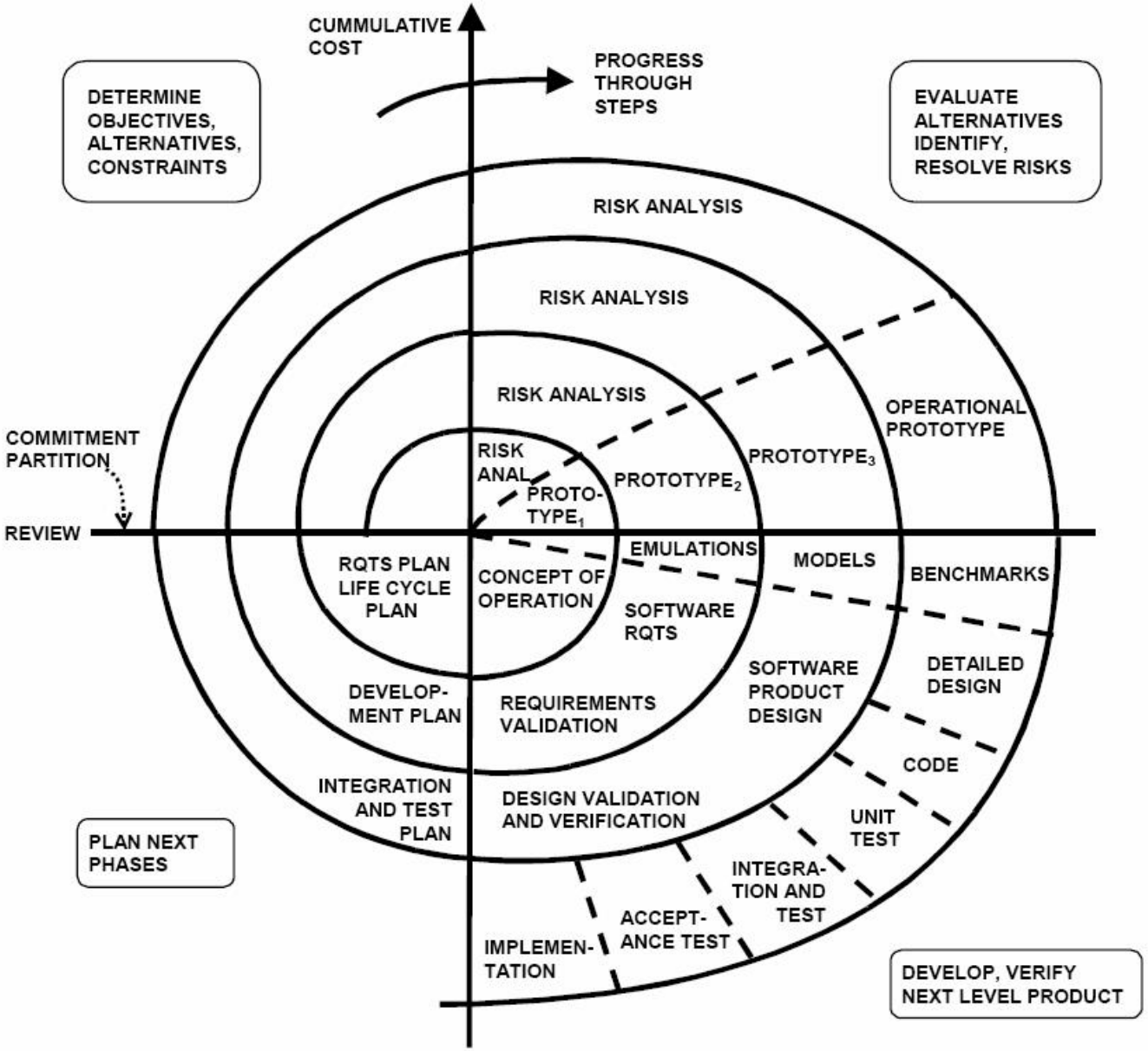
# SDLC Waterfall Model



# Deliverables/Documentations of SDLC Stages/Phases

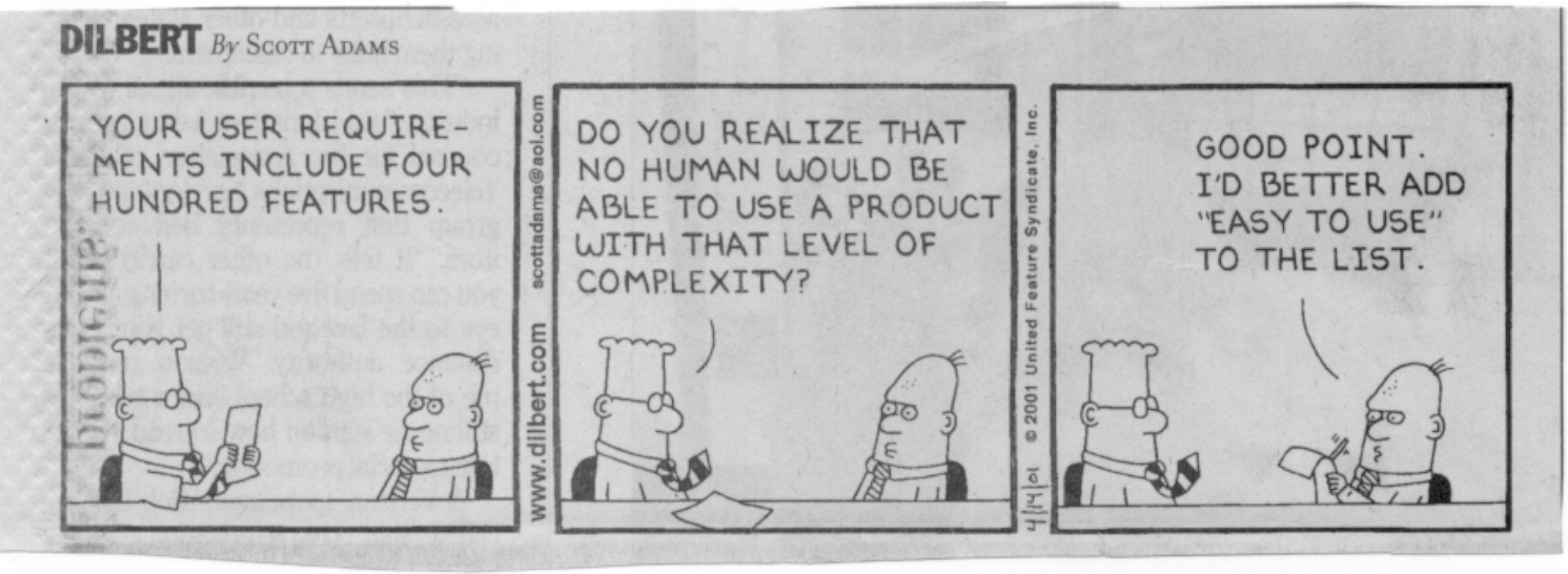
| *Phase* | *Products, Outputs, or Deliverables* |
|---|---|
| Planning | Priorities for systems and projects; an architecture for data, networks, and selection hardware, and IS management are the result of associated systems; |
| | Detailed steps, or work plan, for project; |
| | Specification of system scope and planning and high-level system requirements or features; |
| | Assignment of team members and other resources; |
| | System justification or business case |
| Analysis | Description of current system and where problems or opportunities are with a general recommendation on how to fix, enhance, or replace current system; |
| | Explanation of alternative systems and justification for chosen alternative |
| Design | Functional, detailed specifications of all system elements (data, processes, inputs, and outputs); |
| | Technical, detailed specifications of all system elements (programs, files, network, system software, etc.); |
| | Acquisition plan for new technology |
| Implementation | Code, documentation, training procedures, and support capabilities |
| Maintenance | New versions or releases of software with associated updates to documentation, training, and support |

# Spiral Model and Prototyping

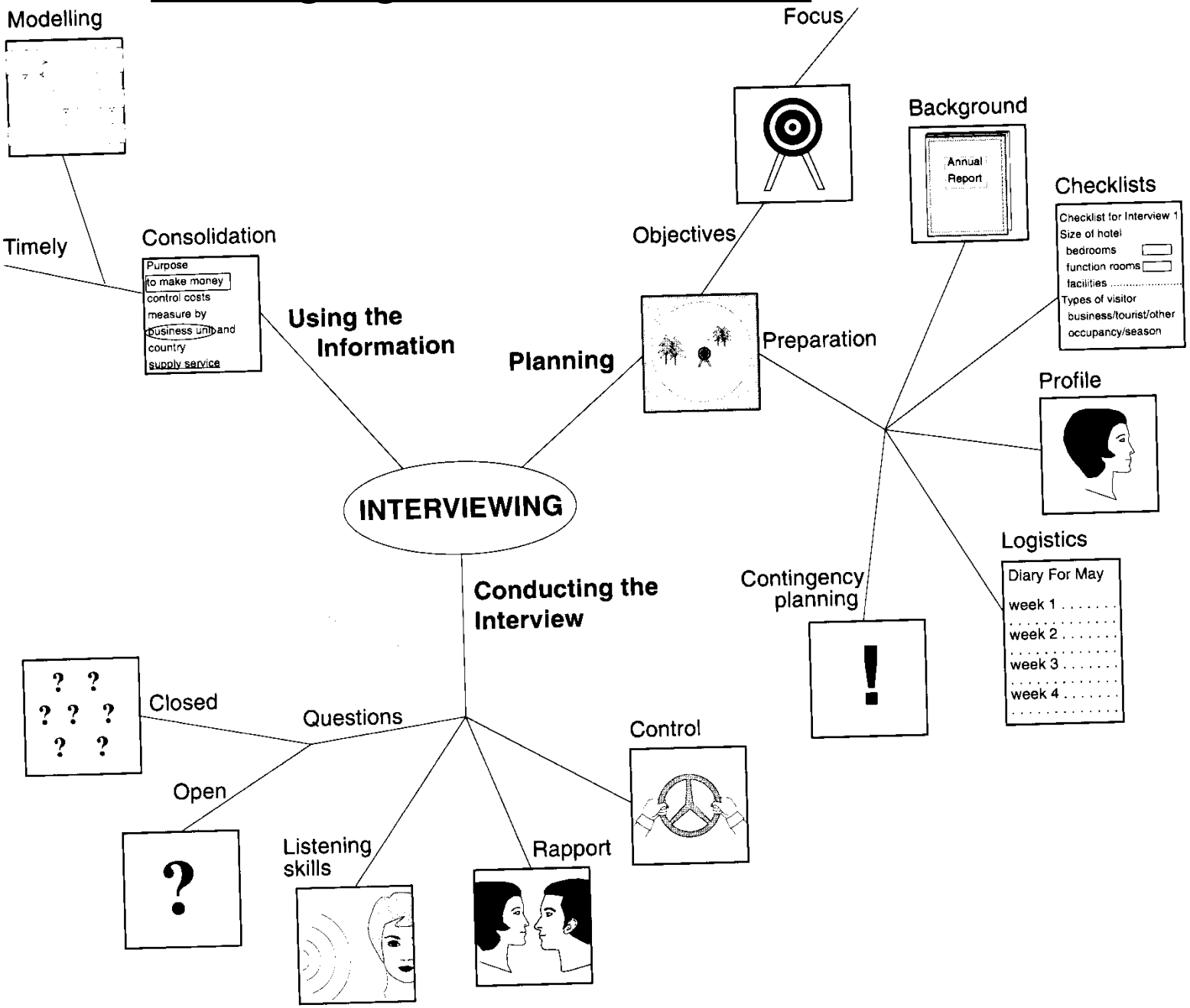



http://en.wikipedia.org/wiki/Spiral_model
http://www.reliablesoftware.com/weblog/uploaded_images/spiral-712085.bmp

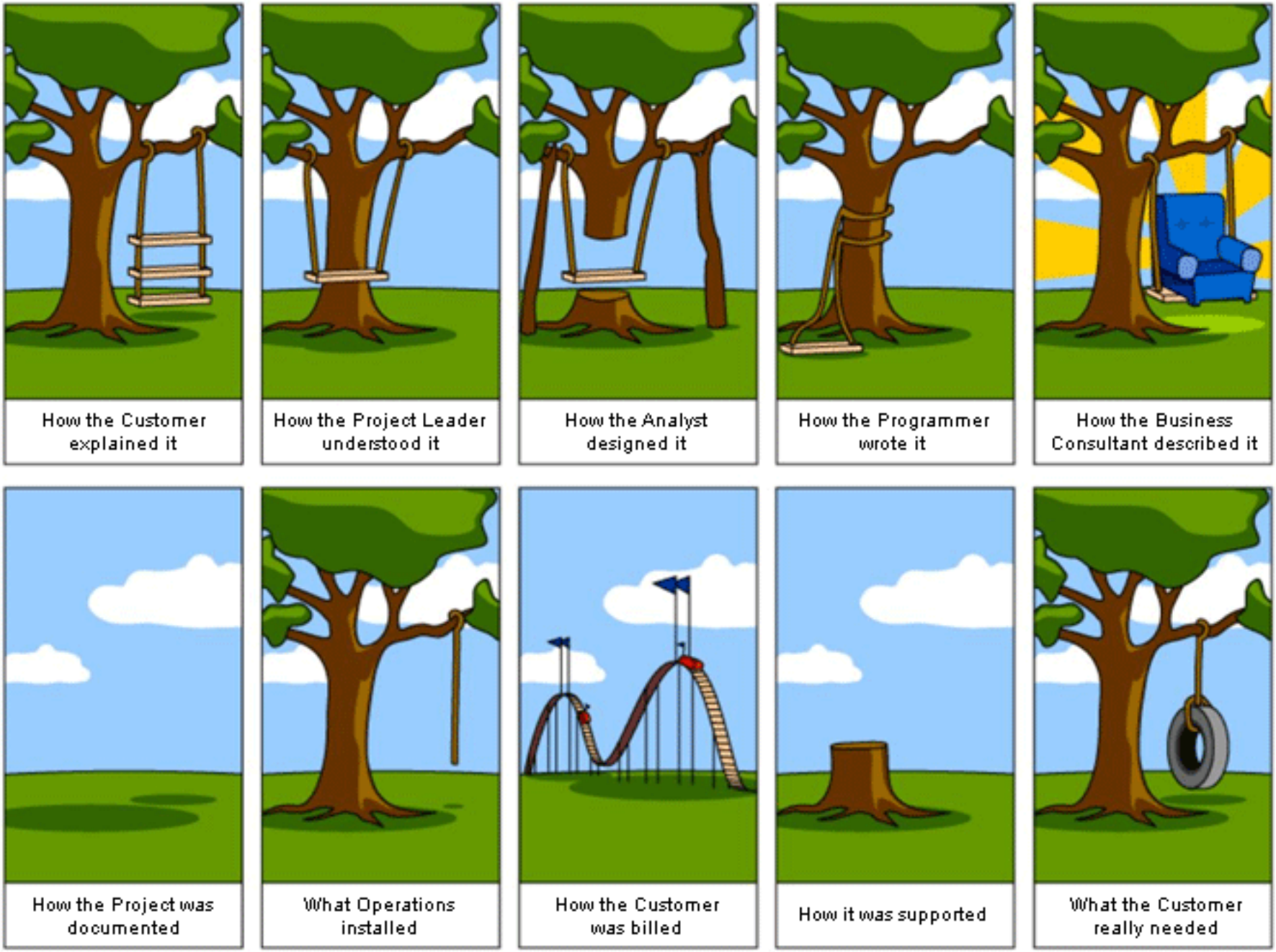
# Requirements Elicitation



# Managing User Interviews

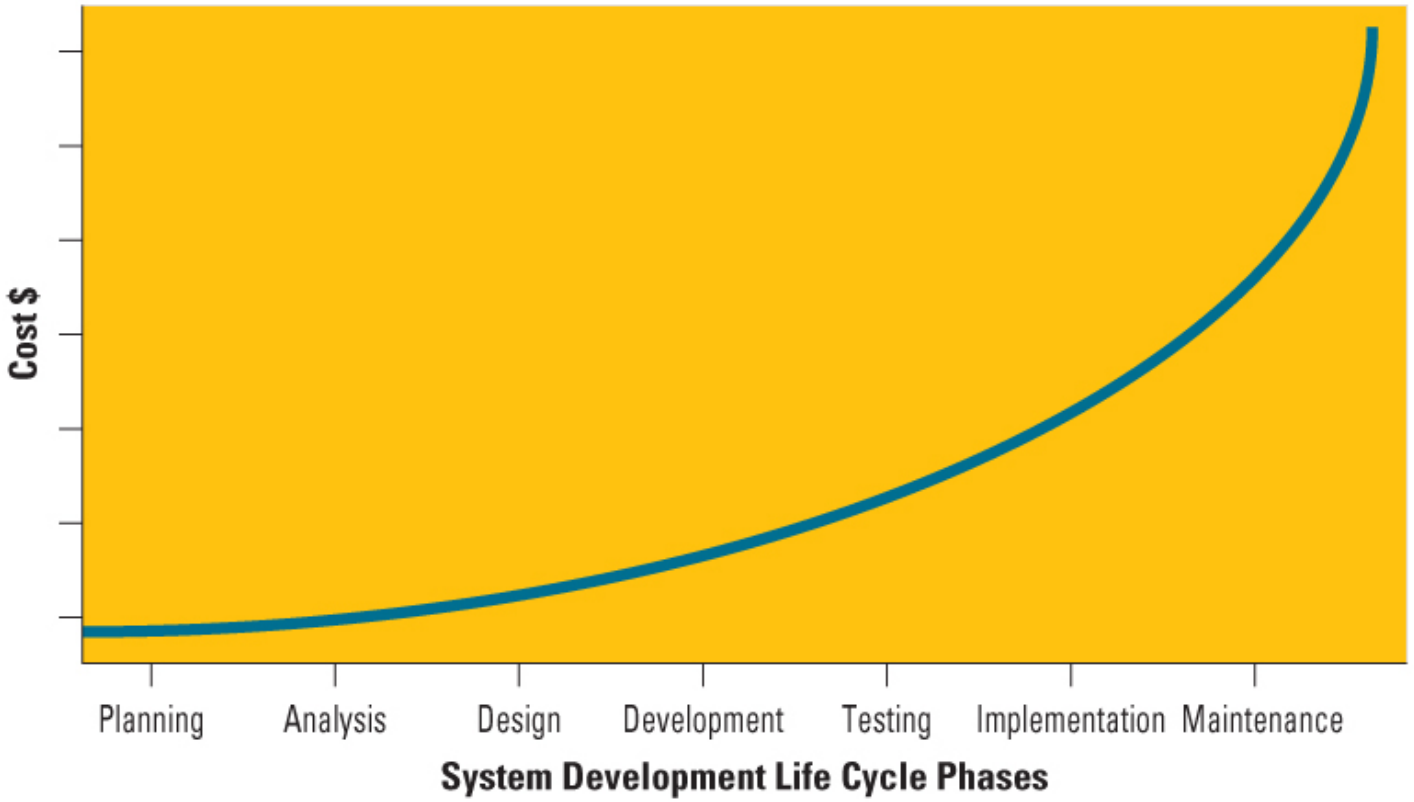


# Stakeholder Perspectives



# SOFTWARE PROBLEMS ARE BUSINESS PROBLEMS

- Find errors early: the later in the SDLC an error is found - the more expensive it is to fix



# Balancing The Triple Constraints in Projects

# The Mythical Man-Month

Men and months are interchangeable commodities only when a task can be partitioned among many workers *with no communication among them* (Fig. 2.1). This is true of reaping wheat or picking cotton; it is not even approximately true of systems programming.

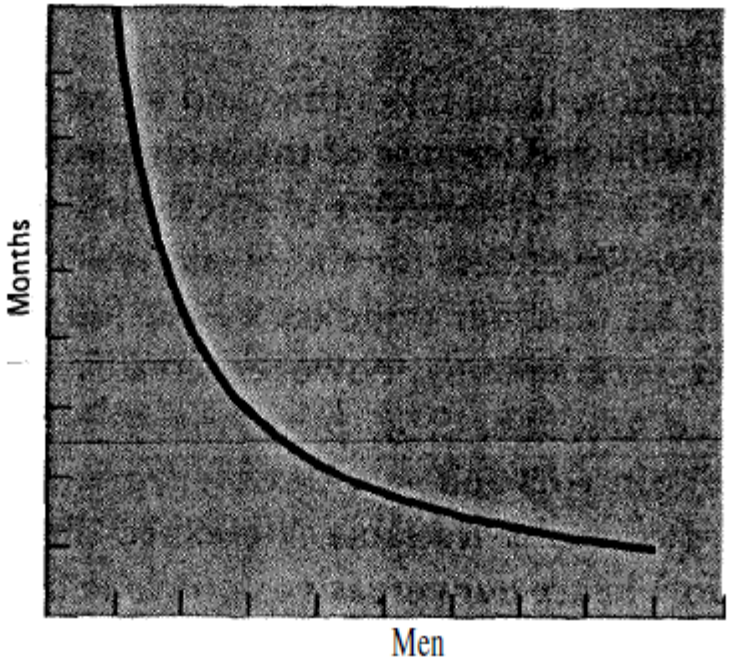
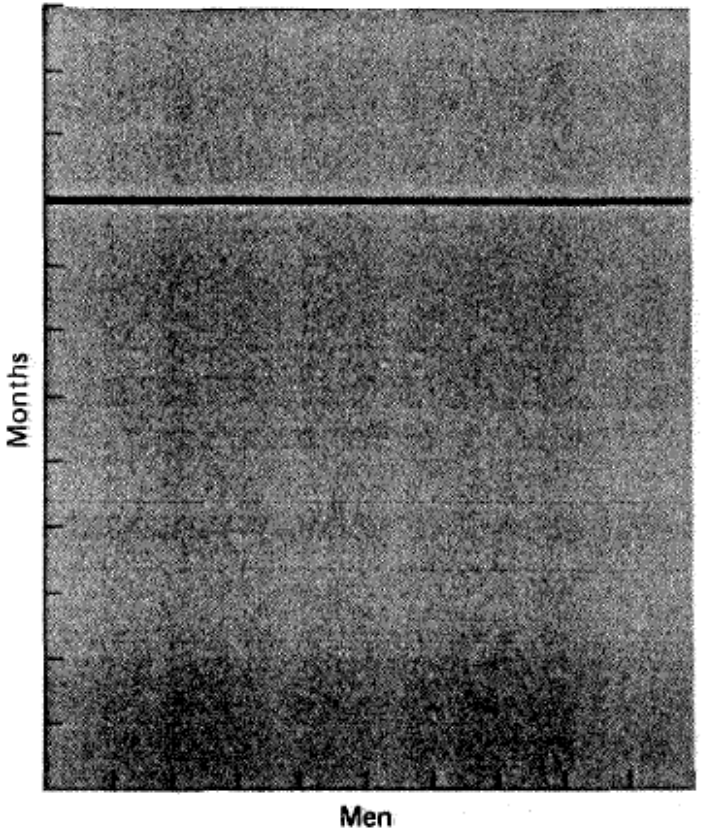



**Fig. 2.1** Time versus number of workers—perfectly partitionable task

http://www.cs.drexel.edu/~yfcai/CS451/RequiredReadings/MythicalManMonth.pdf

# The Mythical Man-Month




Time versus number of workers—unpartitionable task

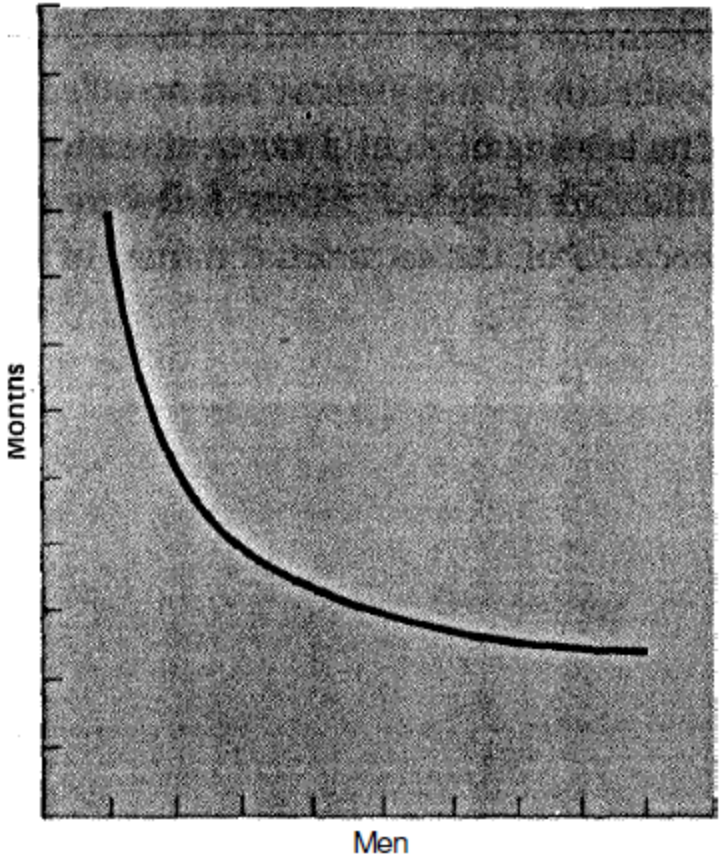
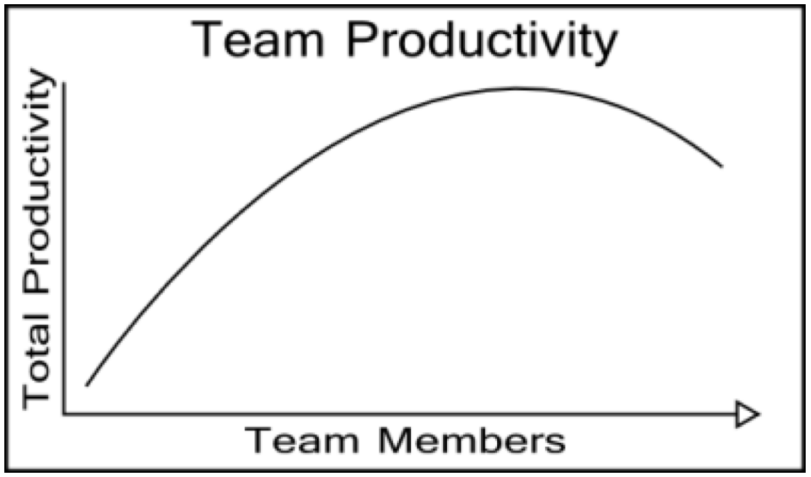
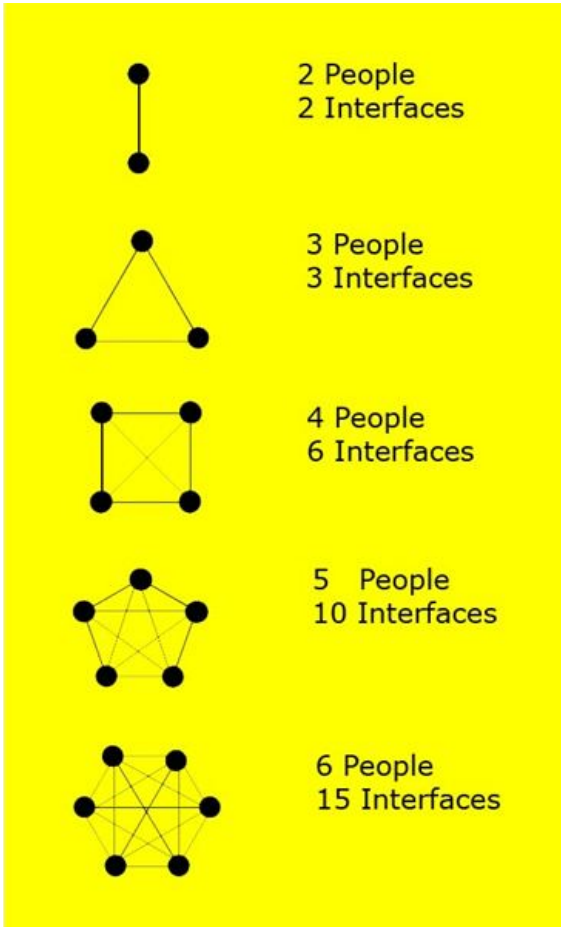



Fig. 2.3 Time versus number of workers—partitionable task requiring communication




# Team Productivity



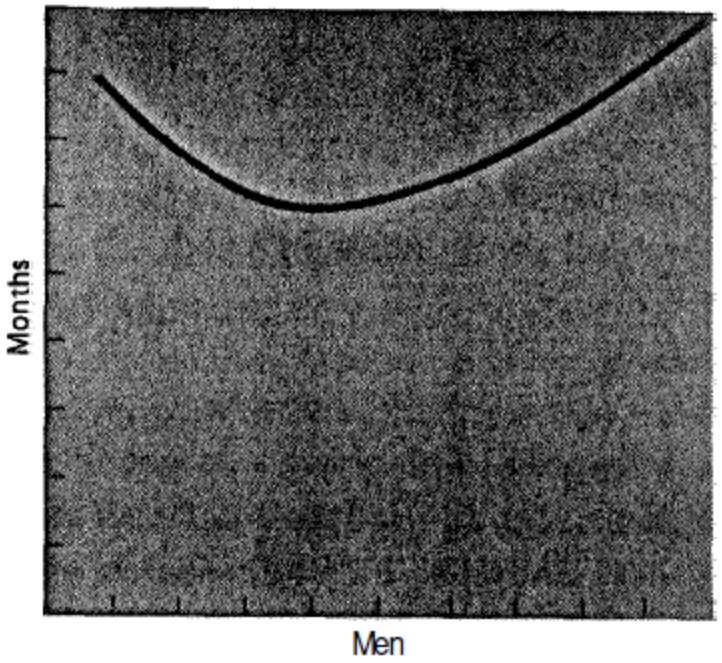



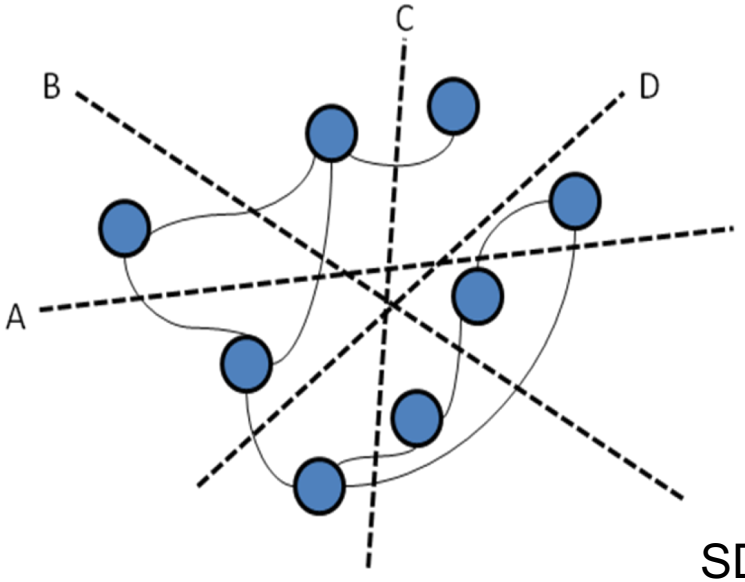

**Fig.** 2.4 Time versus number of workers—task with complex interrelationships

# Adding More People

- **Brook's Law:**
- Adding developers to a late project will make it later.

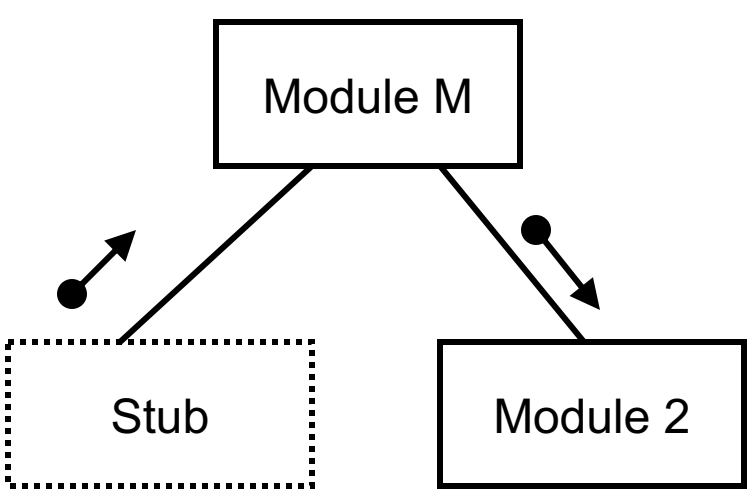
# Design: Cohesion and Coupling

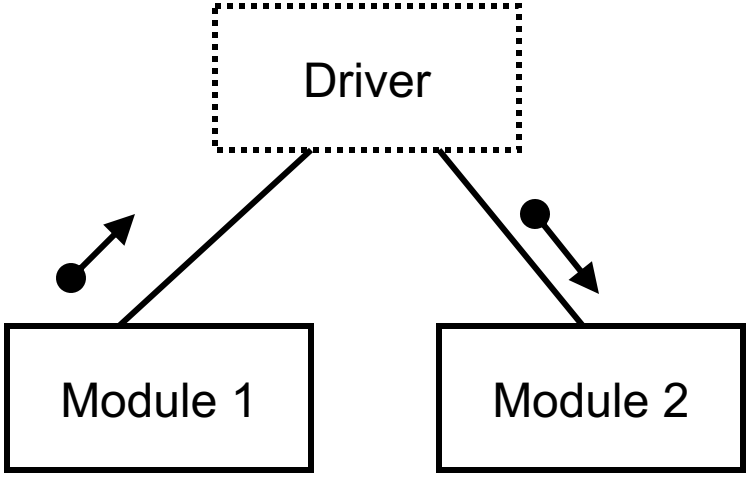
- **Divide and Conquer for effective teamwork**
- Software Design Criteria
- **Modularization**: Simple, stable, and clearly defined interface for each module, no need to understand the internal structure or design of the module to use it.
- Good design is a system that has **low coupling** between modules and **high cohesion** within modules



# Stubs and Drivers

The most common build problem occurs when one component tries to use another component that has not yet been written. This occurs with modular design because the components are often created out of sequence.




- *Stubs* are non-functional components that provide the class, property, or method definition used by the other component. Stubs are a kind of outline of the code you will create later.
- To test two components that need to work together through a third component that has not been written yet, you create a driver. *Drivers* are simply test components that make sure two or more components work together. Later in the project, testing performed by the driver can be performed by the actual component.

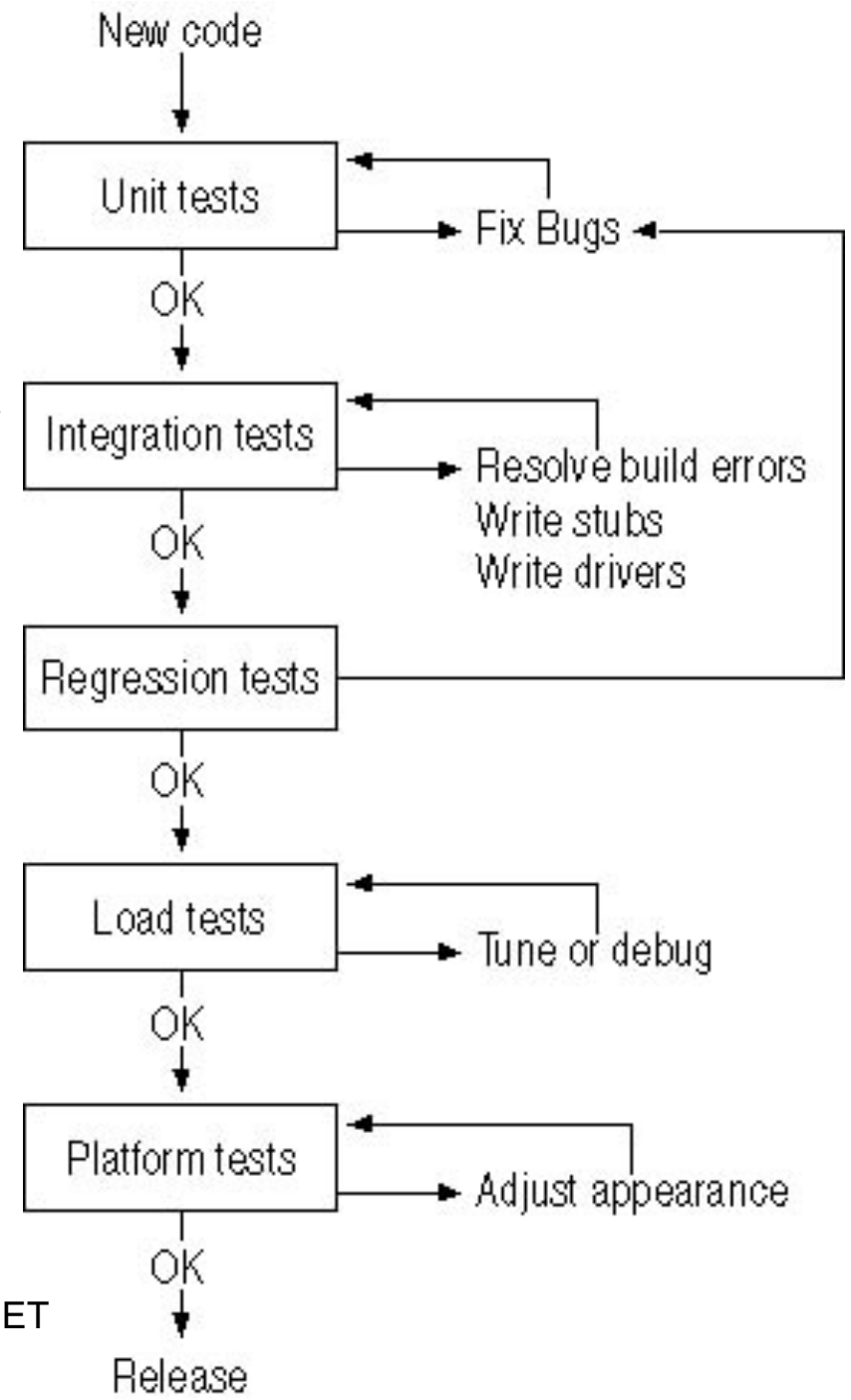
# General Systems Theory: Abstract Thinking

"(General Systems Theory) does not seek, of course, to establish a single, self contained 'general theory' of practically everything which will replace the theories of particular disciplines. Such a theory would be almost without content, for we always pay for generality by sacrificing content, and we can say about practically everything is almost nothing. Somewhere however between the specific that has no meaning and the general that has no content there must be, for each purpose and at each level of abstraction, an optimum degree of generality. It is the contention of the General Systems Theorists that this optimum degree of generality is not always reached by the particular sciences."

Source: http://cimru.nuigalway.ie/david/pdf/SE/Slides/Theory.PDF

# Testing

- Test plan objectives
  - Is thoroughly tested
  - Meets requirements
  - Does not contain defects
- Test plan covers
  - Tools
  - Who
  - Schedule
  - Test result analysis
  - What is being tested?
- Test cases
- Automated testing
  - Reproducible
  - Measurable




Source:
Developing Web Applications with Microsoft Visual Basic .NET and Microsoft Visual C# .NET

# Types of Tests

| *Test type* | *Objectives* |
|---|---|
| **Unit test** | **Each independent piece of code works correctly** |
| **Integration test** | **All units work together without errors** |
| **Regression test** | **Newly added features do not introduce errors to other features that are already working** |
| **Load test (also called stress test)** | **The product continues to work under extreme usage** |
| **Platform test** | **The product works on all of the target hardware and software platforms** |

# Regression and Regression Test

- Regression testing is the process of validating modified parts of the software and ensuring that no new errors are introduced into previously tested code.
- Unit and integration tests form the basis of regression testing. As each test is written and passed, it gets checked into the test library for a regularly scheduled testing run. If a new component or a change to an existing component breaks one of the existing unit or integration tests, *the error is called a regression.*

# Reasons for Project Failures

Primary reasons for project failure include

- Unclear or missing business requirements
- Skipping SDLC phases
- Failure to manage project scope
  - ***Scope creep*** – occurs when the scope increases
  - ***Feature creep*** – occurs when extra features are added
- Failure to manage project plan
- Changing technology

# Why Do Technology Projects Fail?

- Unrealistic or unclear project goals
- Poor project leadership and weak executive commitment
- Inaccurate estimates of needed resources
- Badly defined system requirements and allowing "feature creep" during development
- Poor reporting of the project's status
- Poor communication among customers, developers, and users
- Use of immature technology
- Unmanaged risks
- Inability to handle the project's complexity
- Sloppy development and testing practices
- Poor project management
- Stakeholder politics
- Commercial pressures

# Successful Principles for Software Development

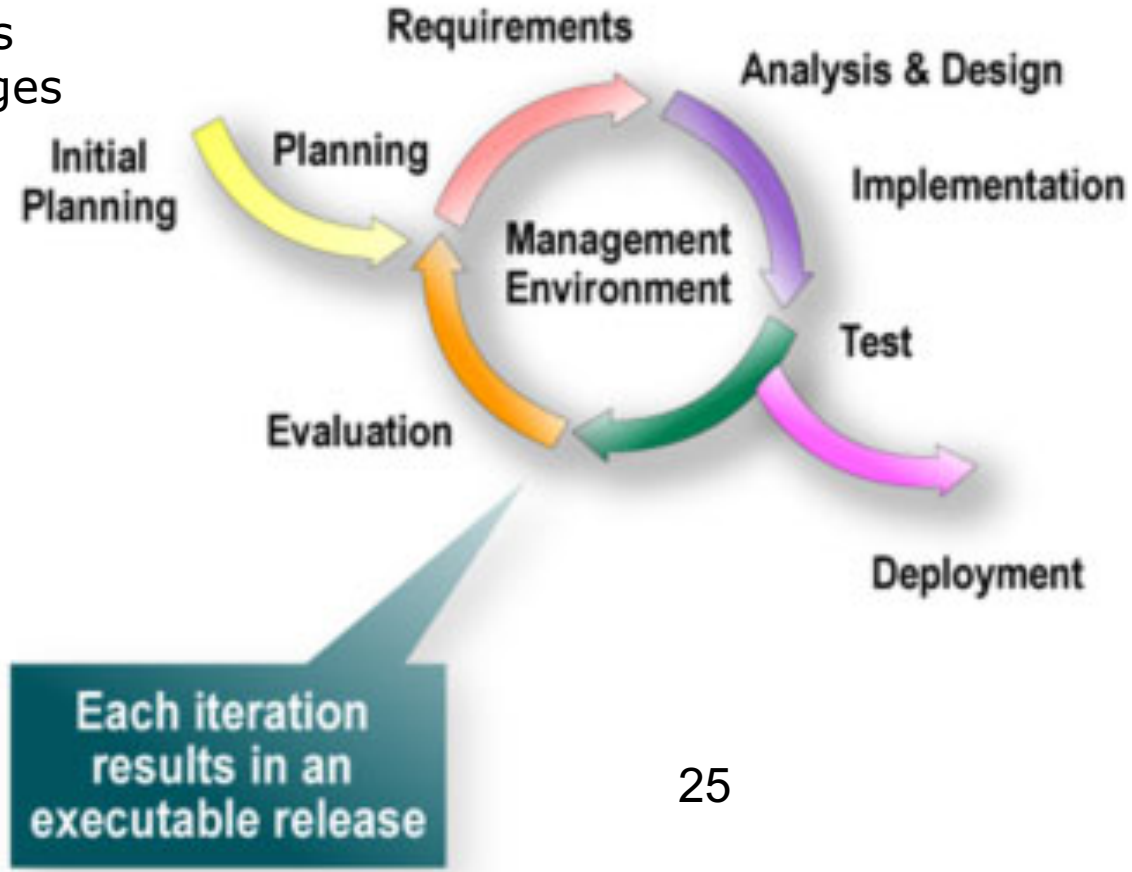
Primary principles for successful ***agile*** software development include:

- Slash the budget
- If it doesn't work, kill it
- Keep requirements to a minimum
- Test and deliver frequently
- Assign non-IT executives to software projects

# The Ten Essentials of RUP

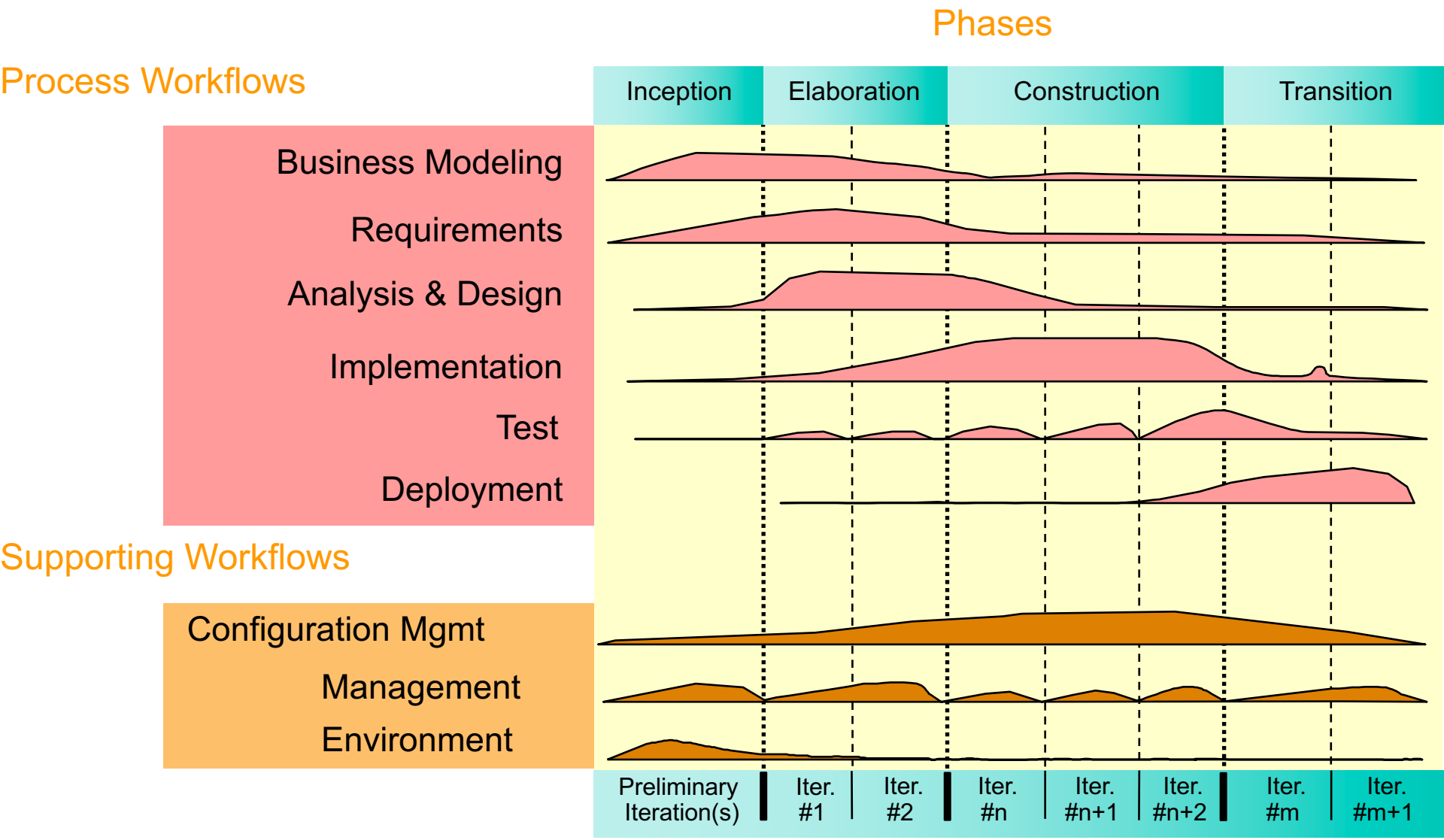
The Ten Essentials of RUP

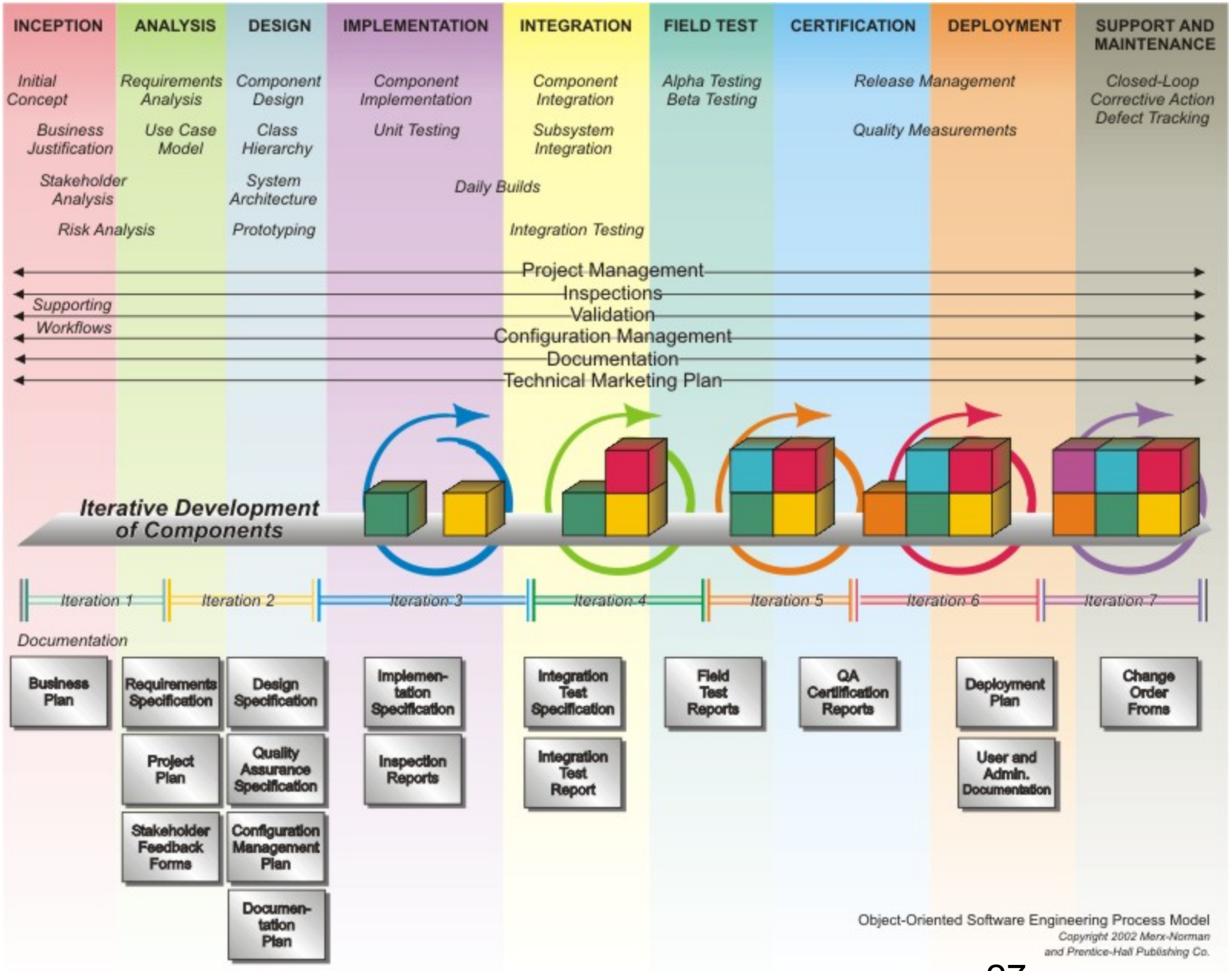
1. Develop a Vision
2. Manage to the Plan
3. Identify and Mitigate Risks
4. Assign and Track Issues
5. Examine the Business Case
6. Design a Component Architecture
7. Incrementally Build and Test the Product
8. Verify and Evaluate Results
9. Manage and Control Changes
10. Provide User Support

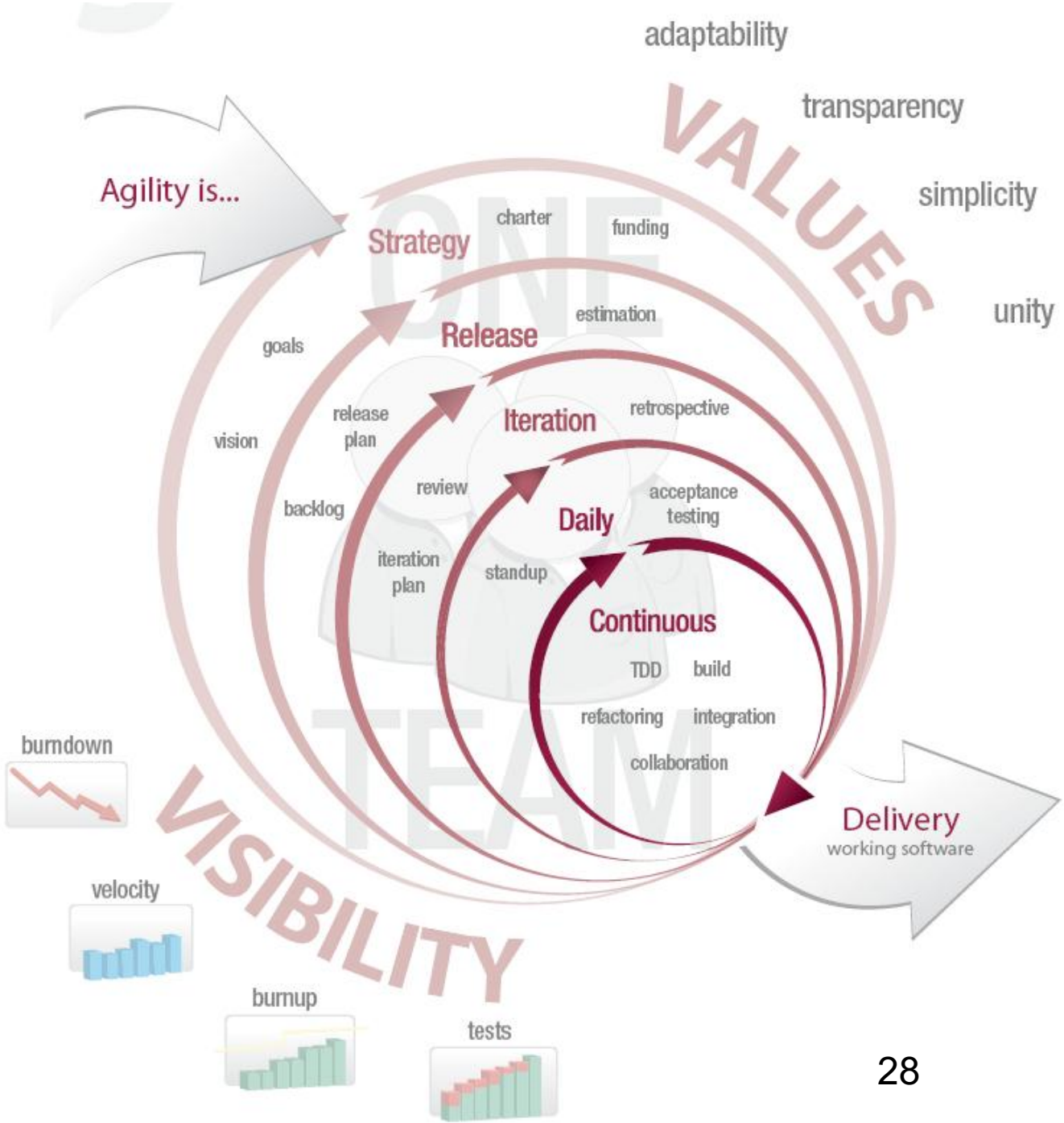



Source:
http://www.therationaledge.com/content/dec_00/f_rup.html

# Unified Process Structure



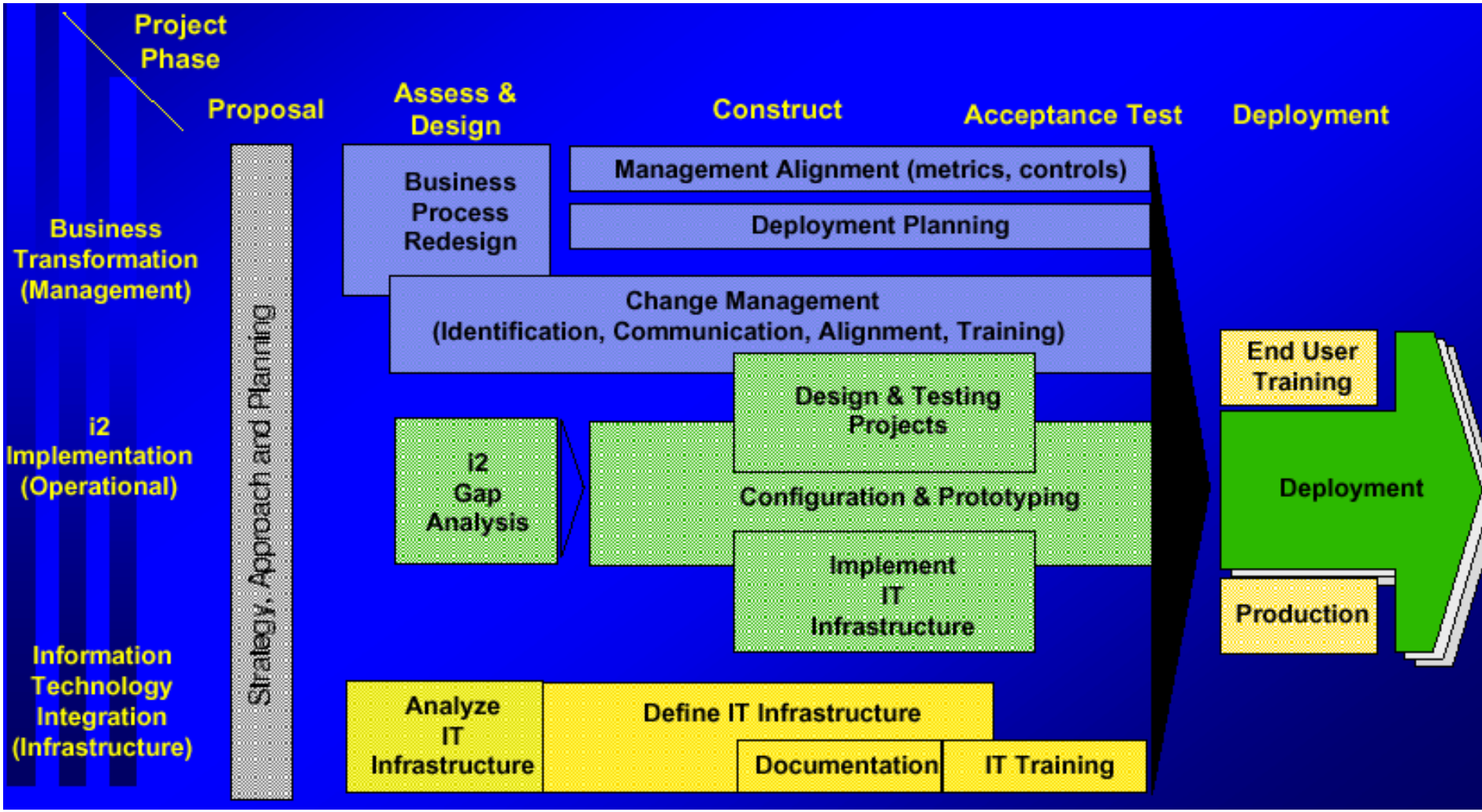

INCEPTION
ANALYSIS
DESIGN
IMPLEMENTATION
INTEGRATION
FIELD TEST
CERTIFICATION
DEPLOYMENT
SUPPORT AND MAINTENANCE
Initial Concept
Business Justification
Stakeholder Analysis
Risk Analysis
Requirements Analysis
Use Case Model
Component Design
Class Hierarchy
System Architecture
Prototyping
Component Implementation
Unit Testing
Daily Builds
Component Integration
Subsystem Integration
Integration Testing
Alpha Testing
Beta Testing
Release Management
Quality Measurements
Closed-Loop
Corrective Action
Defect Tracking
Project Management
Inspections
Validation
Configuration Management
Documentation
Technical Marketing Plan
Supporting Workflows
Iterative Development of Components
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Documentation
Business Plan
Requirements Specification
Project Plan
Stakeholder Feedback Forms
Design Specification
Quality Assurance Specification
Configuration Management Plan
Documentation Plan
Implementation Specification
Inspection Reports
Integration Test Specification
Integration Test Report
Field Test Reports
QA Certification Reports
Deployment Plan
User and Admin. Documentation
Change Order Forms
Object-Oriented Software Engineering Process Model
Copyright 2002 Marx-Norman and Prentice-Hall Publishing Co.

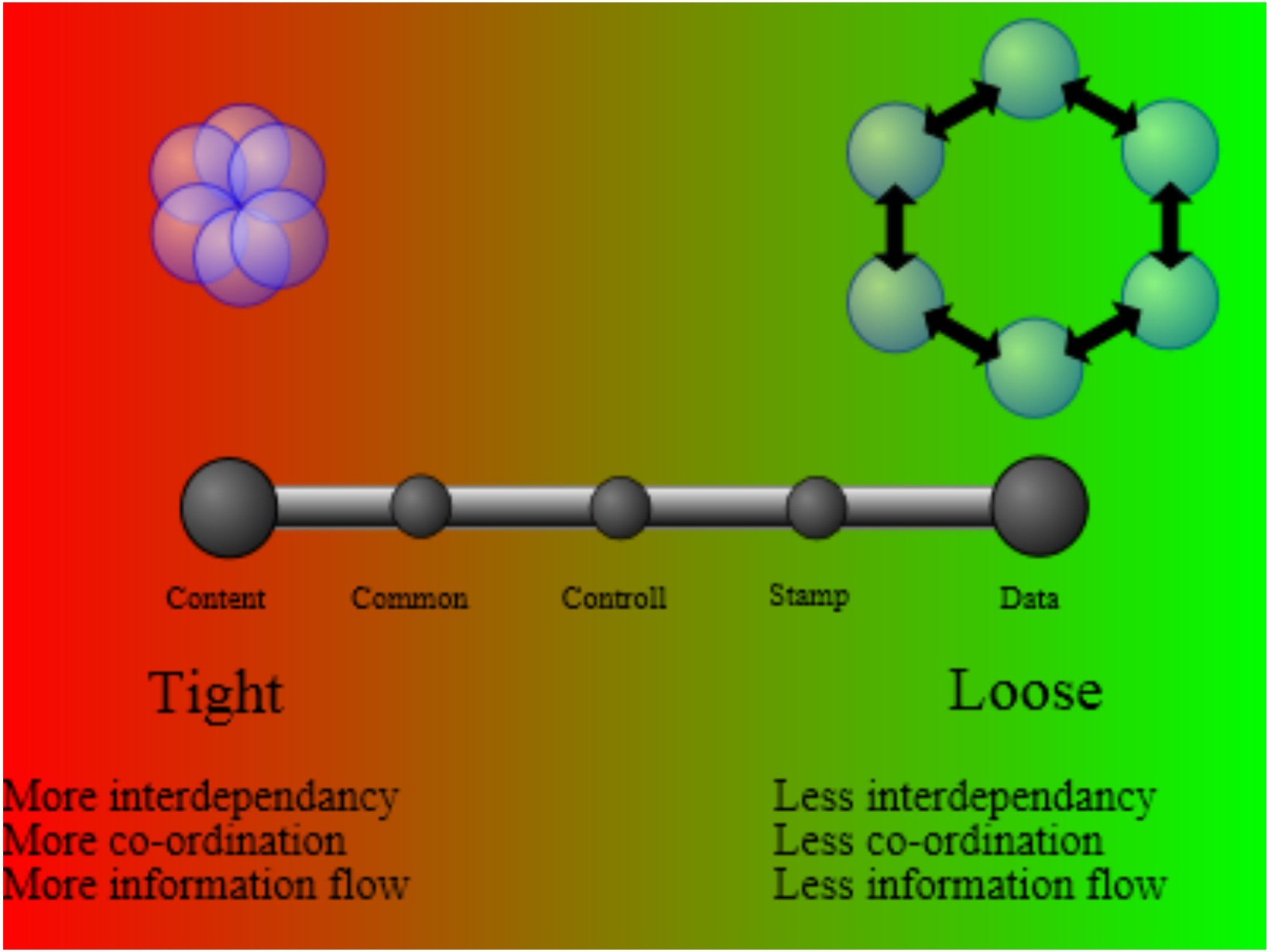

adaptability
transparency
simplicity
unity
VALUES
Agility is...
Strategy
charter
funding
Release
estimation
goals
vision
release plan
Iteration
retrospective
review
backlog
Daily
acceptance testing
iteration plan
standup
Continuous
TDD
build
refactoring
integration
collaboration
Delivery
working software
ONE
TEAM
VISIBILITY
burndown
velocity
burnup
tests

| | |
|---|---|
| **Agile software development (Agile)** | |
| Pros | Minimizes feature creep by developing in short intervals resulting in miniature software projects and releasing the product in mini-increments. |
| Cons | Short iteration may add too little functionality, leading to significant delays in final iterations. Since Agile emphasizes real-time communication (preferably face-to-face), using it is problematic for large multi-team distributed system development. Agile methods produce very little written documentation and require a significant amount of post-project documentation. |
| **Extreme Programming (XP)** | |
| Pros | Lowers the cost of changes through quick spirals of new requirements. Most design activity occurs incrementally and on the fly. |
| Cons | Programmers must work in pairs, which is difficult for some people. No up-front "detailed design" occurs, which can result in more redesign effort in the long term. The business champion attached to the project full time can potentially become a single point of failure for the project and a major source of stress for a team. |
| **Joint application design (JAD)** | |
| Pros | Captures the voice of the customer by involving them in the design and development of the application through a series of collaborative workshops called JAD sessions. |
| Cons | The client may create an unrealistic product vision and request extensive gold-plating, leading a team to over- or under-develop functionality. |
| **Lean software development (LD)** | |
| Pros | Creates minimalist solutions (i.e., needs determine technology) and delivers less functionality earlier; per the policy that 80% today is better than 100% tomorrow. |
| Cons | Product may lose its competitive edge because of insufficient core functionality and may exhibit poor overall quality. |
| **Rapid application development (RAD)** | |
| Pros | Promotes strong collaborative atmosphere and dynamic gathering of requirements. Business owner actively participates in prototyping, writing test cases and performing unit testing. |
| Cons | Dependence on strong cohesive teams and individual commitment to the project. Decision making relies on the feature functionality team and a communal decision-making process with lesser degree of centralized PM and engineering authority. |
| **Scrum** | |
| Pros | Improved productivity in teams previously paralyzed by heavy "process", ability to prioritize work, use of backlog for completing items in a series of short iterations or sprints, daily measured progress and communications. |
| Cons | Reliance on facilitation by a master who may lack the political skills to remove impediments and deliver the sprint goal. Due to relying on self-organizing teams and rejecting traditional centralized "process control", internal power struggles can paralyze a team. |

http://en.wikipedia.org/wiki/Rapid_application_development

# Application Package Life Cycle




Content
Common
Controll
Stamp
Data
Tight
Loose
More interdependancy
More co-ordination
More information flow
Less interdependancy
Less co-ordination
Less information flow

# The End