Software Engineering

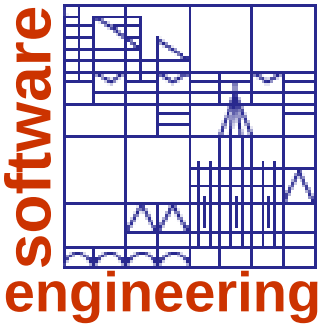
Part 6: *Software Production Process*

Prof. Dr. Stefan Leue

University of Konstanz
Chair for Software Engineering

Stefan.Leue@uni-konstanz.de
http://www.inf.uni-konstanz.de/~soft

Winter Term 2005/2006



Course Outline

1. Introduction and History

- History and Motivation
- Software Crisis
- Software Process

2. Object-Oriented Modeling

- Software Modelling
- Modelling Object Oriented Systems
- The Unified Modeling Language

3. Requirements and Early Life-Cycle Engineering

- Requirements Elicitation
- Software Specification
- State Machines and Petri Nets
- Object Oriented Analysis

Course Outline

4. Software Design

- Classical and Object-Oriented Design
- Design of Concurrent and Distributed Systems
- Interfaces and Contracts
- Software Architecture and Design Patterns

5. Software Quality Assurance

- Reviews and Inspections
- Testing
- Correctness Proofs
- (Software Metrics)

Course Outline

6. Software Production Process

- Evolutionary Models
- Spiral Model
- Unified Process
- Maturity Assessment

7. Software Project Planning and Management

- Project Group
- Staffing and Scheduling
- Software Size Metrics
- Cost and Effort Estimation

Software Production Processs

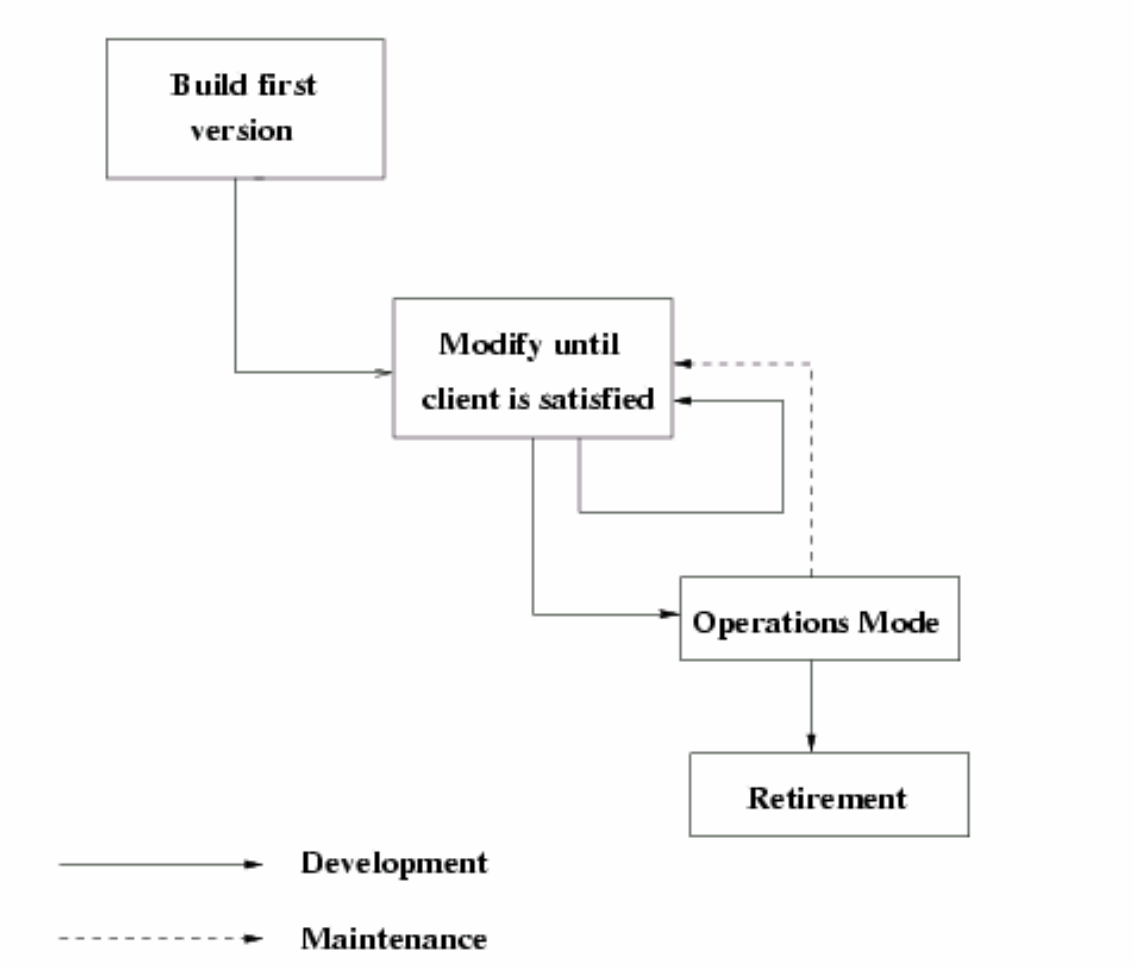
- The process (= organisational scheme) that we follow for the production of a software product. From the first idea throughout deployment and final delivery. ($\rightsquigarrow$ the process covers all phases of the life cycle).
- The process should lead to a high quality product.
- Defining and following a successful process-model for the production process most likely entails high quality of all products manufactured according to that model. $\rightsquigarrow$ incentive to standardize process models.
- Certification of process models that satisfy certain quality demands according to **ISO 9000**.

Software Production Processs

Software Process Models

- Process Model : is the series of steps through which a software product progresses
- Typically, the product is specified, designed, implemented, and once it is operational, it is maintained.
- We will focus on the examination of five process models :
  - Build-and-fix Model
  - Waterfall Model
  - Rapid Prototyping Model
  - Incremental Model
  - Spiral Model
- In addition there is a classification scheme called the *Capability Maturity Model* that is essentially a strategy for improving the software process irrespective of the process model used.

Build-and-Fix



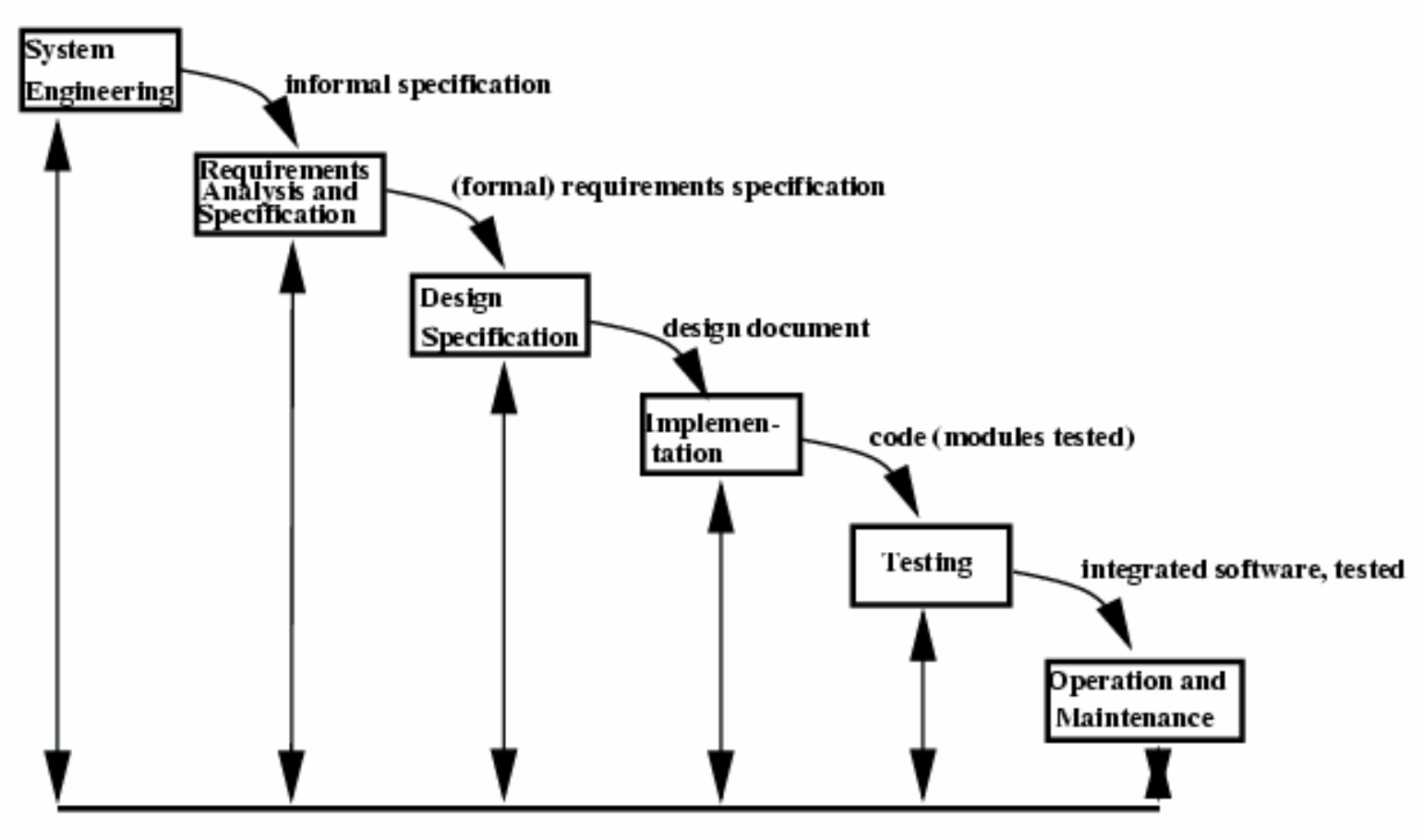
Build And Fix Model (Schematic)

Build-and-Fix

- *Build-and-Fix Model* is the **simplest** process model. The product is constructed **without any design or requirement specifications**.

- Main Points :
  - Specifications and Design are **not considered** as initial steps in the life-cycle
  - The developers simply **build** a product which is **reworked** as many times as necessary until it satisfies the client
  - It works for **small** programming exercises (100-200 lines long) but fails for products of any reasonable size.
  - Does not produce any specification or design documents $\rightsquigarrow$ the overall **maintenance** cost is considerably higher
  - Alternative: Instead of the build-and-fix approach, the initial time is better to be spent to chose an overall process model that will specify the requirements, specification, planning, design, implementation, integration, and maintenance phases

Review of Waterfall Model



Review of Waterfall Model

- *The Waterfall Model* was the only widely accepted process model until the early '1980s

- Main Points:
  - Each major **phase** in the software life-cycle (*Requirements*, *Specification*, *Planning*, *Design*, *Implementation*, *Integration* commences only when the **previous phase** has been checked and approved by the SQA group and the client
  - At each step **flaws** and inconsistencies with respect to the previous step may be discovered. At this point the process **loops back** to the previous phase and it does not continue until all the problems have been resolved.
  - The Waterfall model with its feedback loops permits **modifications** to be made to any phase that precedes the current phase
  - A phase is deemed to be complete only when the **documentation** for this phase is complete, checked, and approved

Review of Waterfall Model

- When the system has been implemented then it is given to the client for *testing*. User manuals and other documents listed in the contract will be delivered at this point
- Once the client agrees that the product satisfies its specification document and the product is **accepted**, any other modification to the product (fixes of faults, extensions) is considered **maintenance**

Analysis of Waterfall Model

- Advantages:
  1. Enforces a **disciplined** approach (i.e. take away the fun of programming)
  2. **Testing** and **verification** is enforced at every phase of the life-cycle
  3. **Documentation** produced can be used to **reduce maintenance costs**

- Disadvantages:
  1. The first time the **client sees** a working arifact is only after the entire product has been coded. This is the syndrome of *"I know this is what I asked for, but it is not really what I wanted"* client response
  2. The model depends heavily on **written specifications**, requirements, design documents etc, the process therefore tends to be **bureaucratic**. Only documentation can not describe what the product will look like how it will really performs, and it does meet the client's real needs

Analysis of Waterfall Model

Benefits of waterfall model (compared to *code and fix* process model):

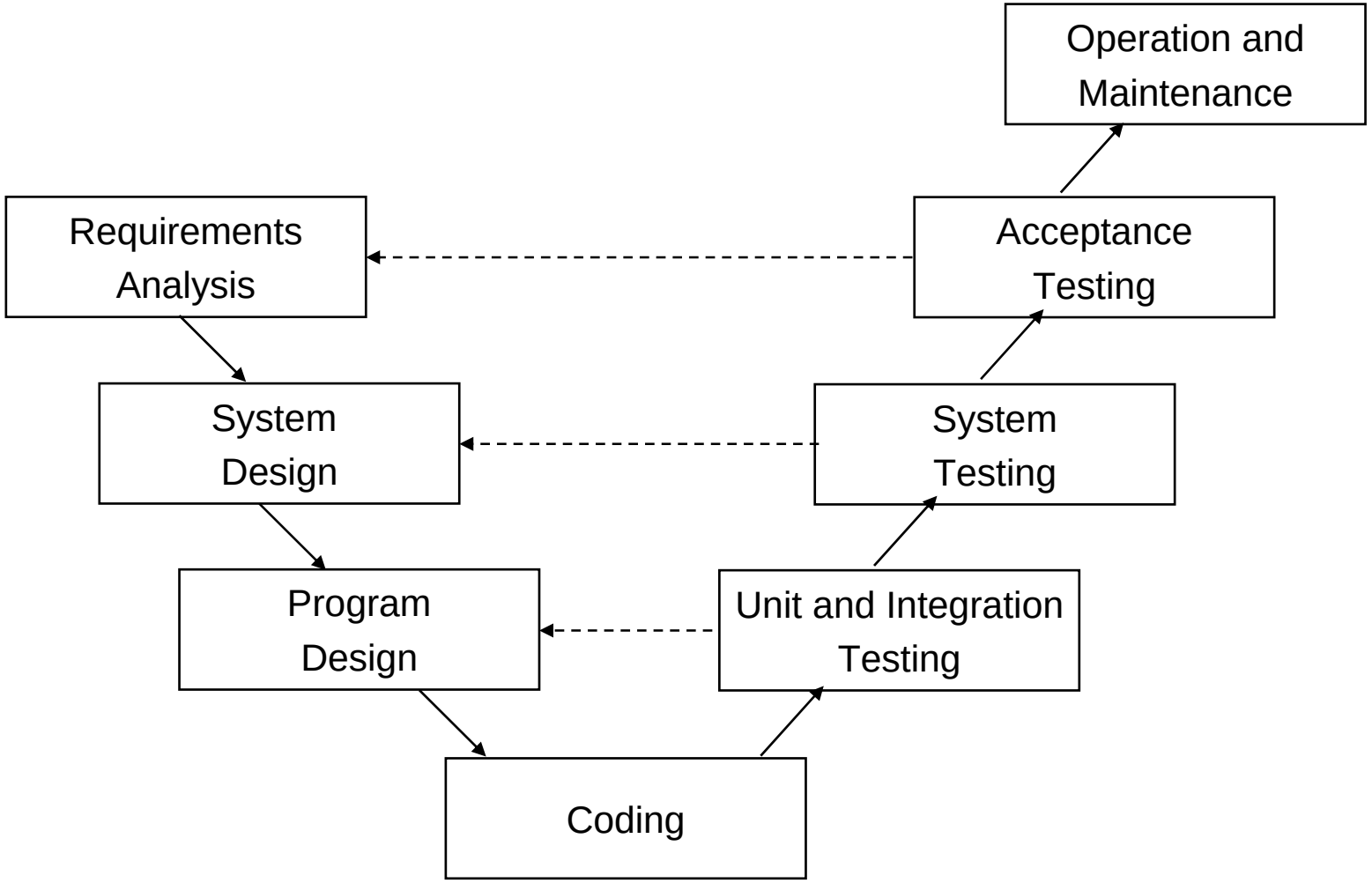
1. Forces to think about well-defined process.
2. Distinction of phases and transitions between them.
3. Allows for a seperation of tasks (SRS, design, implementation).
4. Supports software process as group activity.

Analysis of Waterfall Model

Problems with waterfall model:

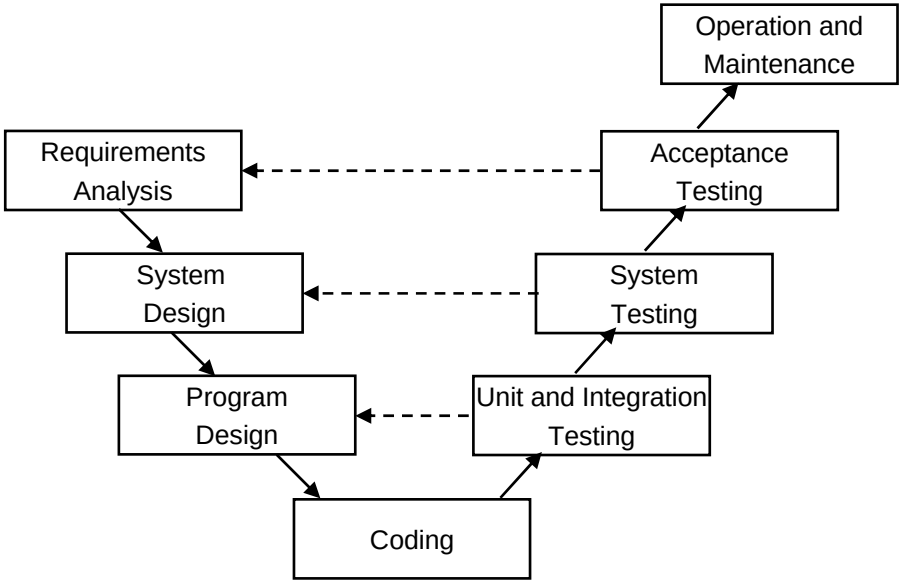
1. Difficult to estimate ressources at early stages.
2. Difficult to assess whether final product meets expectations.
3. Users don't know requirements beforehand, at early stages.
4. No emphasis on anticipation for change.
5. Process is "document-driven" $\leadsto$ bureaucratic.
6. **Not adequate to represent evolving software production process.**
7. **Nothing is functional and delivered to the user until the end of the development process.**

V-Model



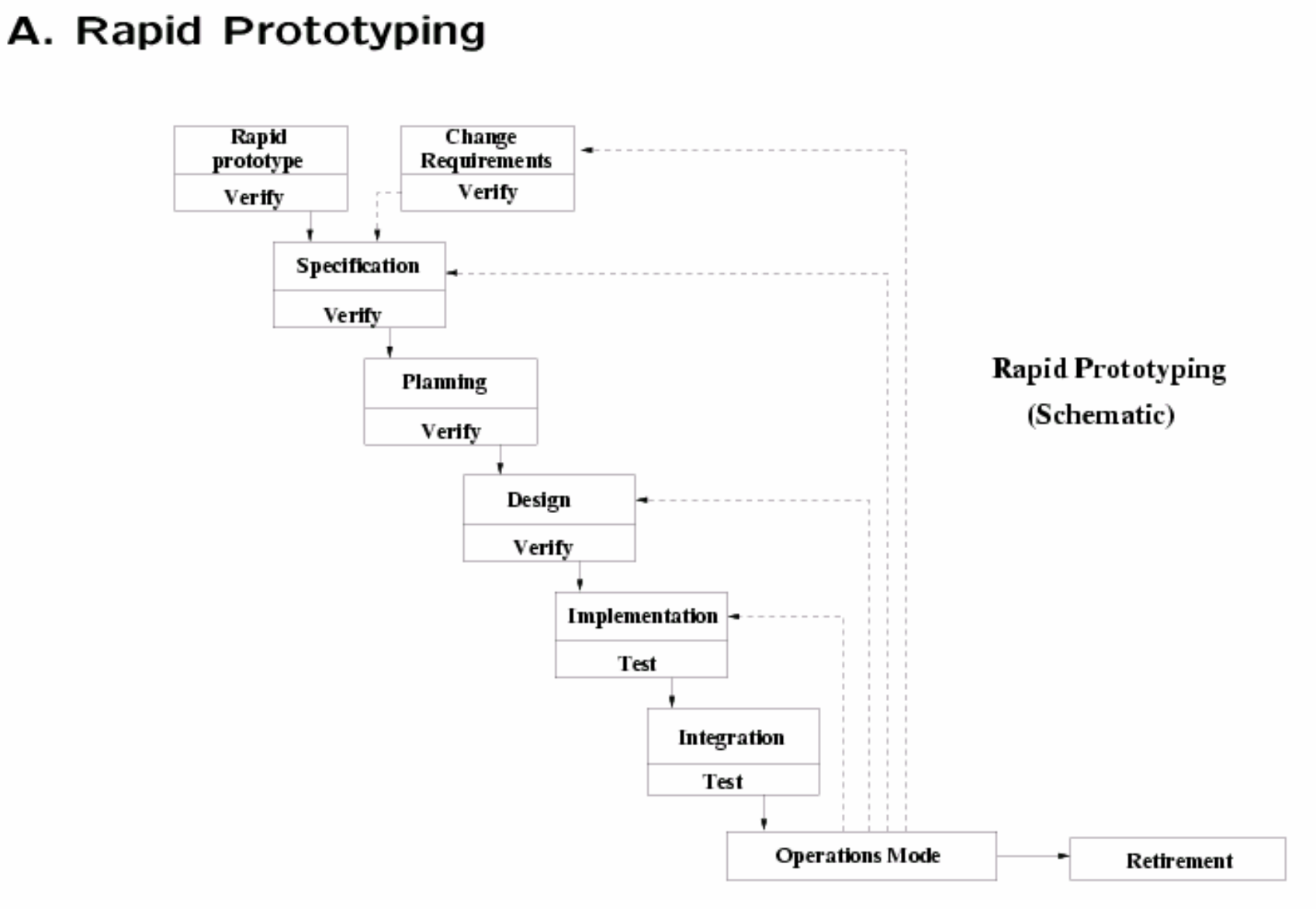
- variant of Waterfall model
- proposed by German Ministry of Defense

V-Model



- **Principal Ideas**
  - requirements and design to the left, testing and operation to the right
  - focus on testing-based verification of the design
    - unit and integration testing verifies design
    - system testing verifies system design
    - acceptance testing (customer) verifies that all requirements have been properly implemented
      - at least one test per requirement

Evolutionary Process Models



Evolutionary Process Models

A. Rapid Prototyping

- Quickly realize a first prototype of the system by following a waterfall-like process.
- When the first prototype is deployed, gather feedback from the client/user, and feed this information back into the SRS, design and implementation phases.
- Discard the prototype, and implement the "real" product based on experience gained.

Problems:

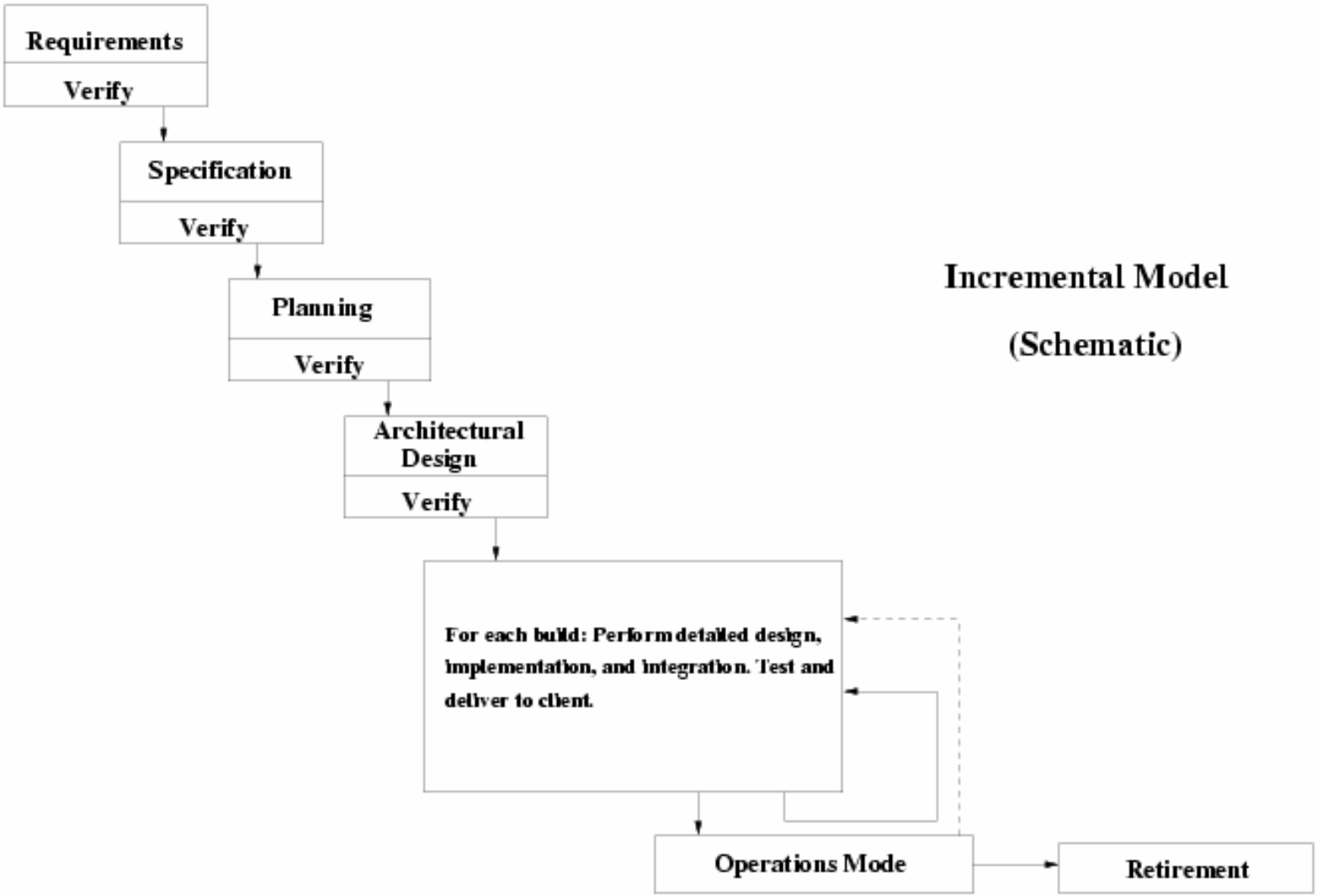
- Pay-off for increased development effort?
- If the customer sees something that works, he is unlikely willing to pay $$ for the "real thing".
- Does not remedy the inherent problems in the waterfall model (e.g. temporal gap between requirements specification and deployment.)

Evolutionary Process Models

Integrating the Rapid Prototyping and the Waterfall Model

- The objective is to eliminate the drawback of the Waterfall model in that what is delivered to the client may not be what the client really needs
- In the integrated model
  1. The rapid prototype is used as input to the waterfall model
  2. The organization has the opportunity to assess the technique while minimizing the associated risk
- $\rightsquigarrow$ Incremental Process Models.

Evolutionary Process Models



Evolutionary Process Models

B. Incremental Implementation Model.

- Follow waterfall model up to implementation and module testing.
- Early on, determine useful subsets (=increments) of the system that can be deployed and define interfaces between them.
- Different increments implemented according to priorities and at differnt times.
- Results in multistage implement-test-integrate-test cycles in which feedback should be taken into account.

Evolutionary Process Models

C. Incremental Delivery Model.

- Implement product incrementally, where each increment is a *self-contained* functional unit that performs some useful purpose.
- Deliver increments + supporting documentation to user / customer.
- Get early user feed-back that will be taken into account for the next increment

Attention: danger to fall back to "code & fix" – ensure that waterfall phases are respected (e.g., that apparently necessary changes are propagated back to the SRS).

Benefits: a) early delivery to user, b) adjustment of design and objectives (requirements) based on observed realities.

Evolutionary Process Models

D. Incremental Development & Delivery Model.

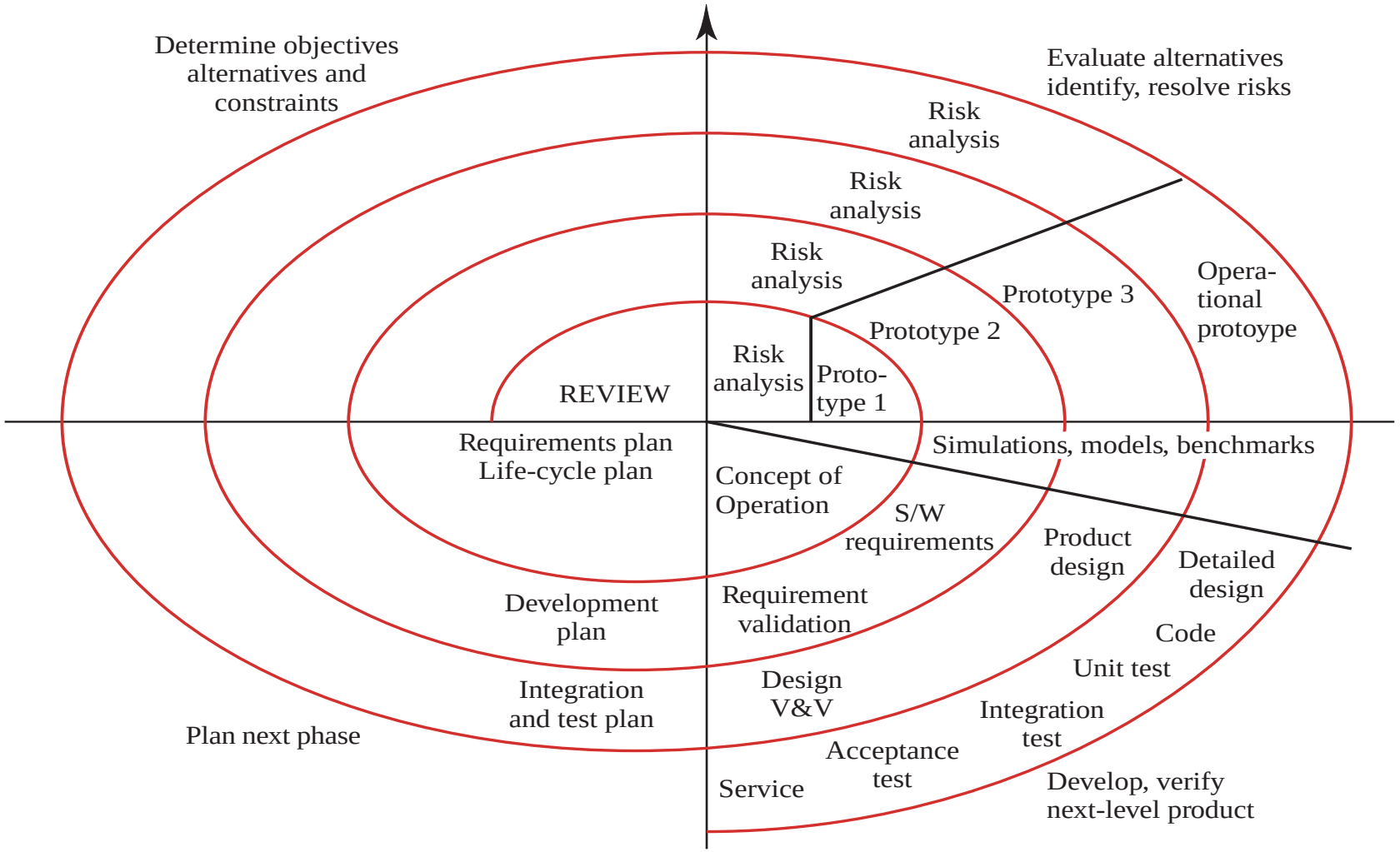
- Analyze an increment at SRS level.
- Design, code, test, integrate, test and deploy increment. (Waterfall model is followed, the process consists of many mini-waterfalls).
- Users use delivered parts to "understand what they want".
- Maintenance disppears largely from life-cycle, or rather: the life-cycle becomes maintenance.

Evolutionary Process Models

E. Spiral Model.

- The spiral model focuses on the idea of minimizing the development **risk** via the use of prototypes and other means.
- Main Points:
  - A simplistic view is to view the *spiral model* as a *waterfall* model with each phase preceded by **risk analysis**.
  - **Before commencing** each phase, an attempt is made to **control or resolve the risks**.
  - If it proves to be impossible to resolve all the significant risks at that stage then the project is immediately **terminated**.
  - **Prototypes** can be effectively used to provide information with regard to certain classes of risk
  - The dimensions in the *Spiral Model* represent **cumulative cost** and the **progress** of the development
- **Encompasses** waterfall model, prototyping, and evolutionary model.
- Is a **meta-model** (i.e., it can be applied to any of the above models).
- The model has been successfully used to develop a wide variety of products (mainly of very complex nature)

Spiral Model (Boehm, 1988)

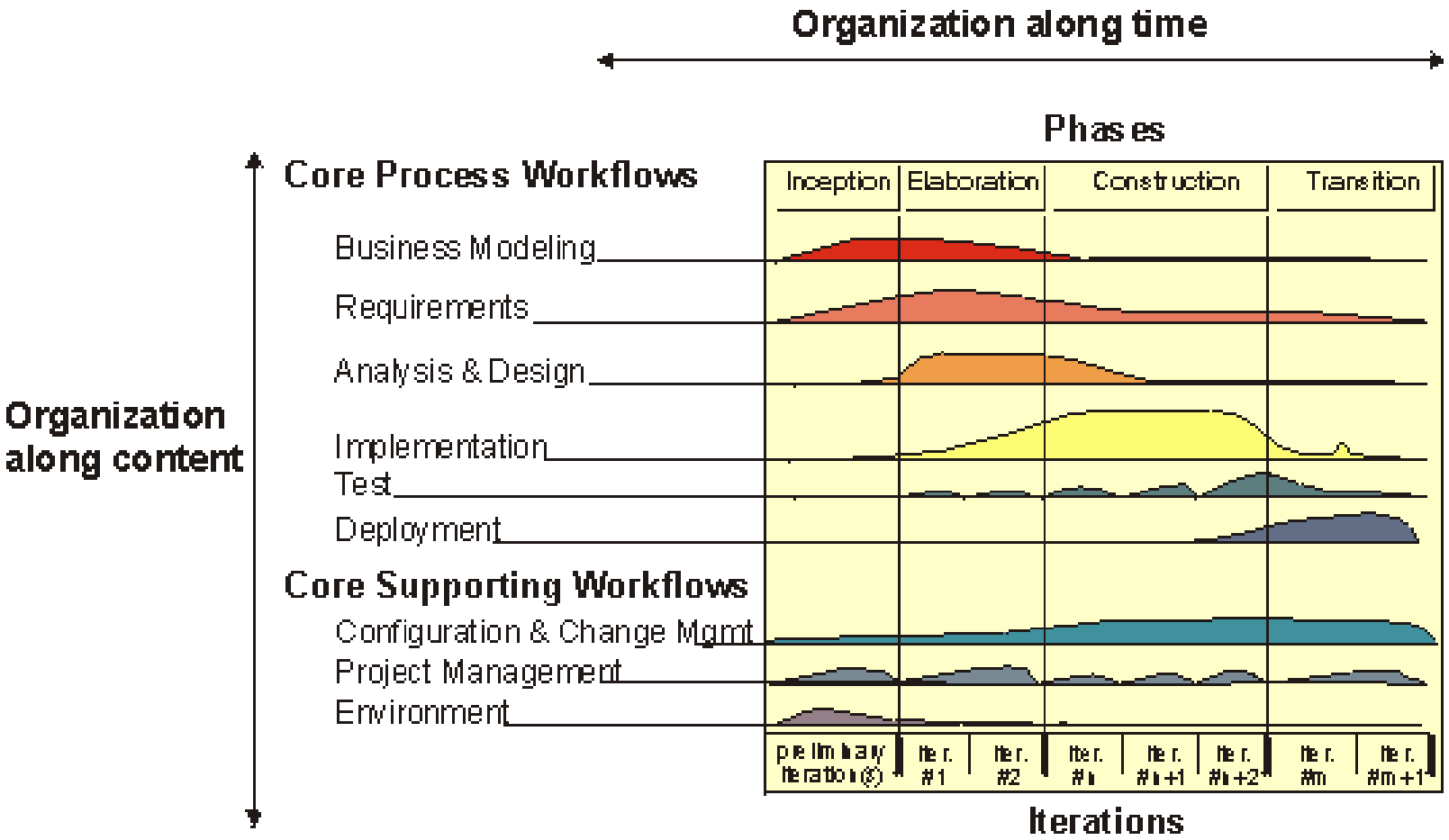


Rational Unified Process

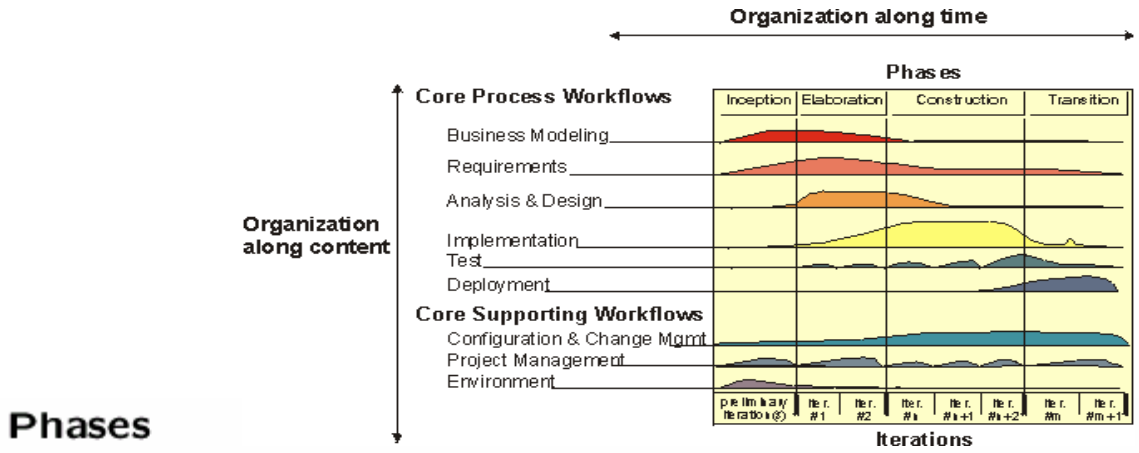
- Practical software process model, currently followed by many object-oriented, industrial projects.
- "Sold" by Rational Corp., see

  `http://www.rational.com/products/rup/index.jtmpl`
- **Iterative** model encompassing **Waterfall**.
- Two dimensions: **phases** and **workflows**.

Rational Unified Process

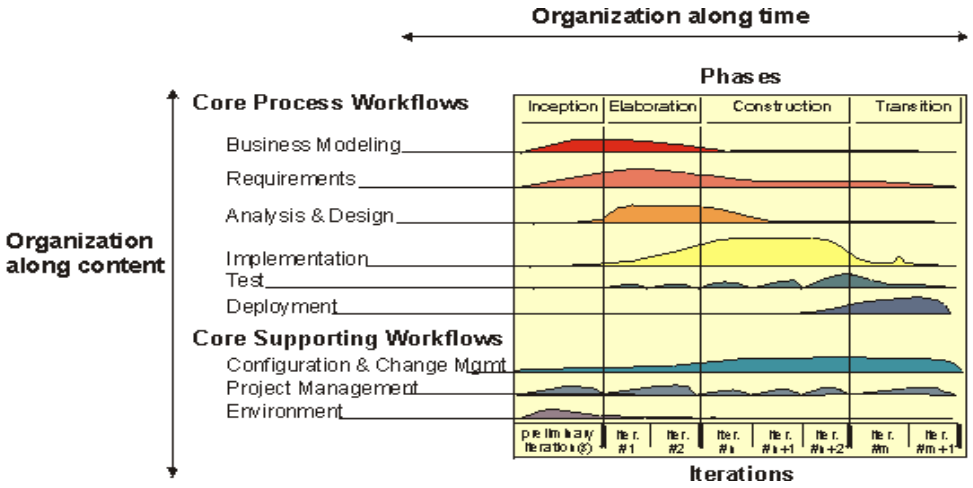


Rational Unified Process



- Project **milestone** at end of every phase.
- Different phases
  - **Inception**: "good idea", business case, end-product vision, scope of project
  - **Elaboration**: planning activities and resources, specify features, designing the architecture
  - **Construction**: building the product, evolving architecture and vision until completed product ready for transfer into user community
  - **Transition**: transition into user community (manufacturing, training, delivering, maintenance) until user is satisfied
- Each phase is subdivided into **iterations**, each iteration leads to a **release** ($\rightsquigarrow$ incremental process model)

Rational Unified Process



Workflows

- Each iteration is a mini-**waterfall** process
- leading to a new **increment**

Agile Methods

- **Agile**

  1 : marked by ready ability to move with quick easy grace
  2 : having a quick resourceful and adaptable character <*an agile mind*>

  (Merriam-Webster)

- **Agile Software Engineering Methods**
  - dissatisfaction with the overheads involved in classical software engineering design methods led to the creation of agile methods
    - customer changes late in the life cycle
    - fast product cycles require fast release cycles
  - principles
    - focus on the code rather than the design
    - based on an iterative approach to software development
    - are intended to deliver working software quickly and
    - evolve this quickly to meet changing requirements
  - agile methods are probably best suited to small/medium-sized business systems

Some of the subsequent slides on Agile Methods and Extreme Programming have been adopted from Sommerville, Software Engineering, 7th ed., and the companion web site for this book.

Agile Methods

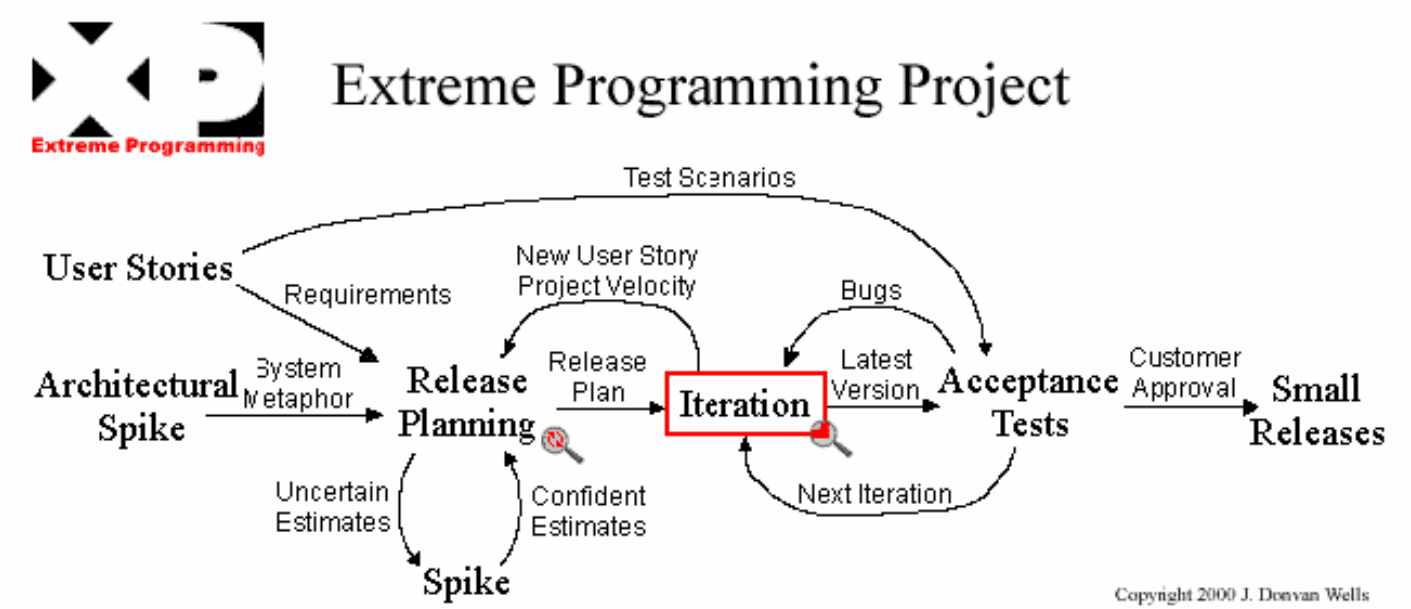
- **Web Sites**
  - Agile Methods
    - http://www.agilemanifesto.org
    - http://www.agilealliance.com
  - Extreme Programming (XP)
    - http://www.extremeprogramming.org

Principles of Agile Methods

Principle	Description
Customer involvement	The customer should be closely involved throughout the development process. Their role is provide and prioritise new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognised and exploited. The team should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and design the system so that it can accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process used. Wherever possible, actively work to eliminate complexity from the system.

Agile Methods

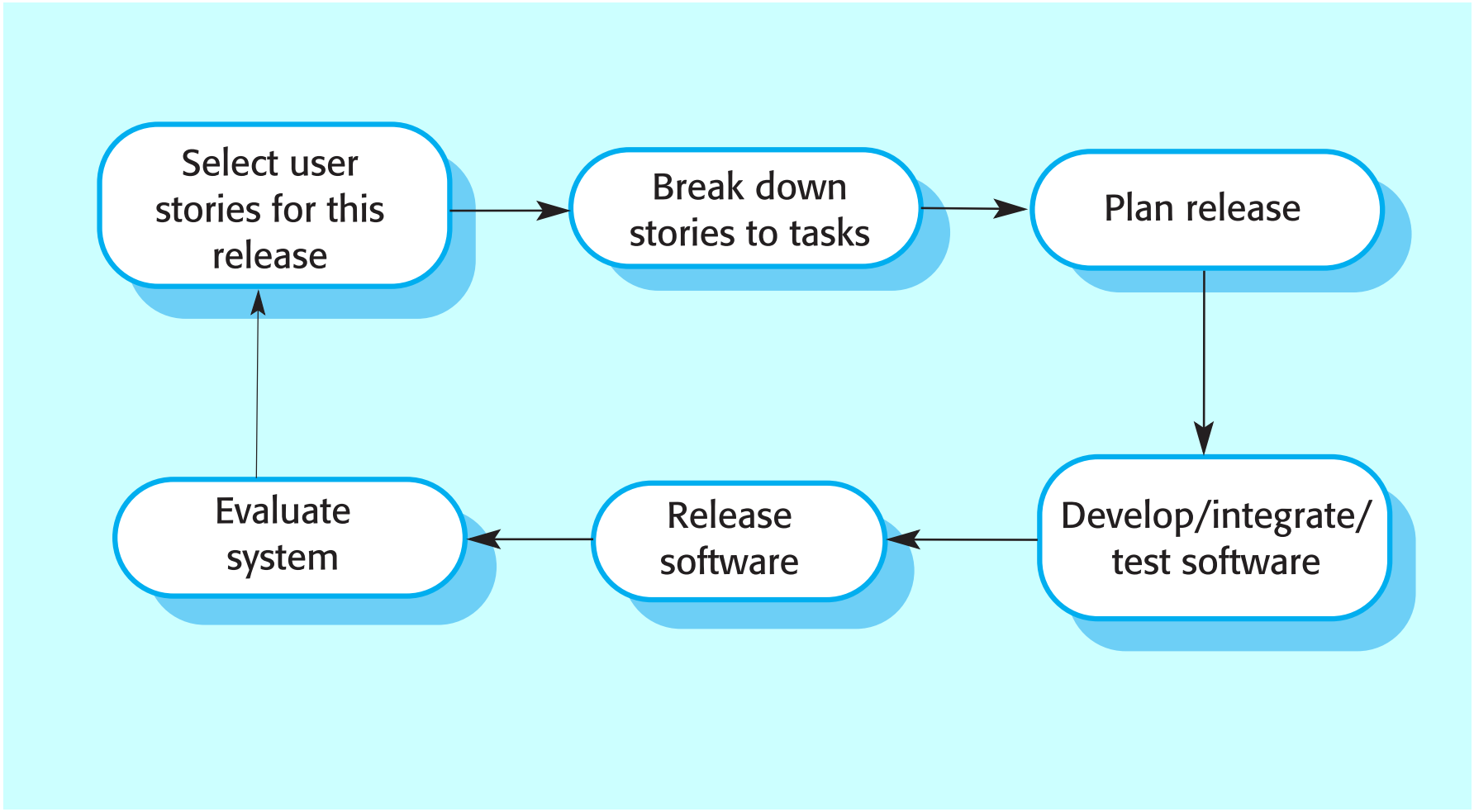
- **Extreme Programming**
  - perhaps the best-known and most widely used agile method.
  - eXtreme Programming (XP) takes an 'extreme' approach to iterative development.
    - new versions may be built several times per day;
    - increments are delivered to customers every 2 weeks;
    - all tests must be run for every build and the build is only accepted if tests run successfully.
  - selected central concepts: user stories and pair programming



software engineering

Extreme Programming

- **The XP Release Cycle**



Extreme Programming

XP Practices 1

Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development Tasks.
Small Releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple Design	Enough design is carried out to meet the current requirements and no more.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

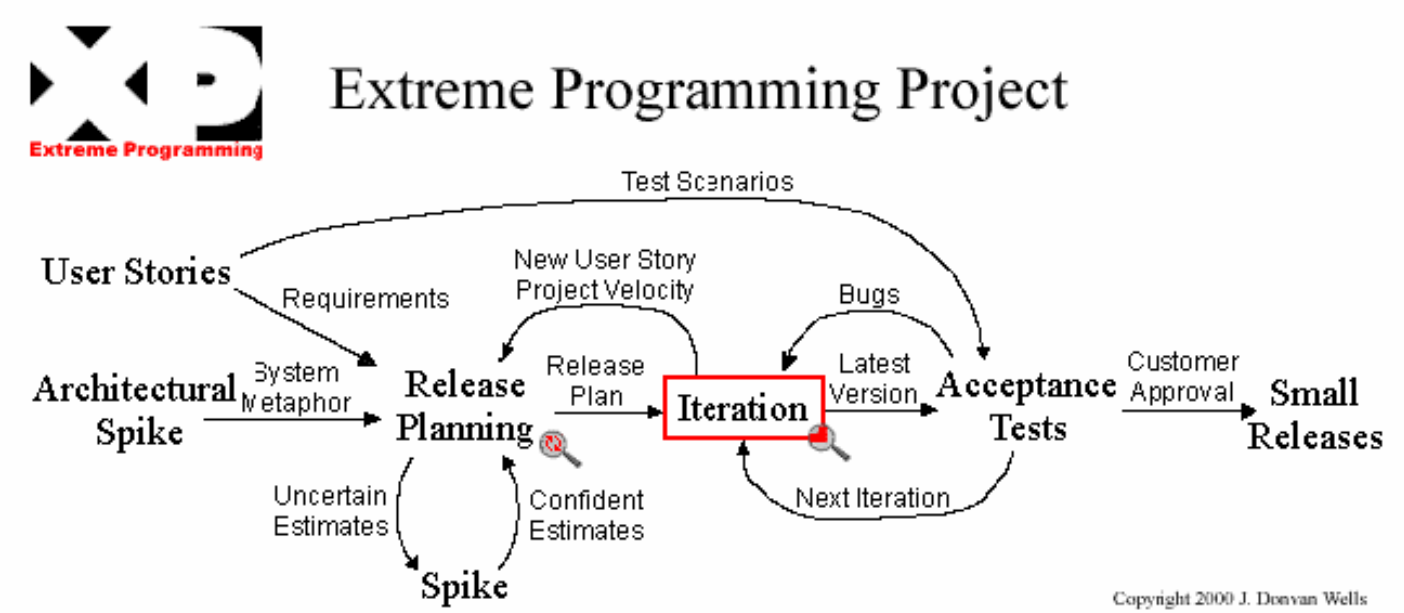
Extreme Programming

XP Practices 2

Pair Programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective Ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers own all the code. Anyone can change anything.
Continuous Integration	As soon as work on a task is complete it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of over-time are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site Customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Extreme Programming

- **Requirements in XP**
  - user requirements are expressed as scenarios or user stories.
    - written on cards and
    - the development team breaks them down into implementation tasks
    - tasks are the basis of schedule and cost estimates.
  - customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.



Extreme Programming

- **Example**
  - story card for document downloading

Downloading and printing an article

First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it - this can either be through a subscription, through a company account or by credit card.

After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer.

You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful.

If the article is a print-only article, you can't keep the PDF version so it is automatically deleted from your computer.

Extreme Programming

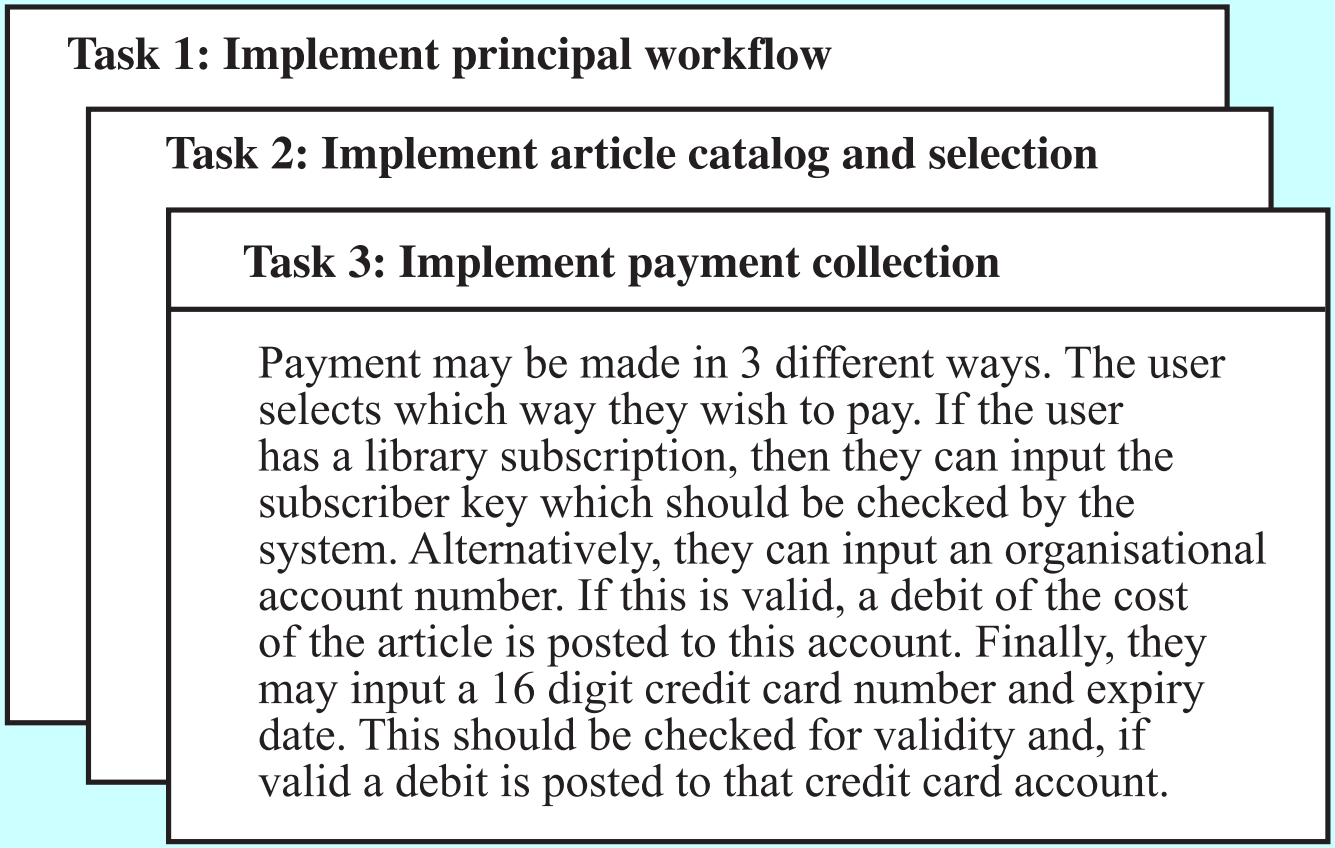
- **XP and Design for Change**
  - design for change: it is worth spending time and effort anticipating changes as this reduces costs later in the life cycle
  - XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated
  - rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented

Extreme Programming

- **Testing in XP**
  - test-first development
    - writing tests before code clarifies the requirements to be implemented
    - tests are written as programs rather than data so that they can be executed automatically
    - test includes a check that it has executed correctly
    - all previous and new tests are automatically run when new functionality is added
  - incremental test development from scenarios
  - user involvement in test development and validation
  - automated test harnesses are used to run all component tests each time that a new release is built (i.e., regression testing)

Extreme Programming

- **Example**
  - Task cards for document downloading



Extreme Programming

- **Example**
  - test case description

Test 4: Test credit card validity

Input:
A string representing the credit card number and two integers representing the month and year when the card expires
Tests:
Check that all bytes in the string are digits
Check that the month lies between 1 and 12 and the year is greater than or equal to the current year.
Using the first 4 digits of the credit card number, check that the card issuer is valid by looking up the card issuer table. Check credit card validity by submitting the card number and expiry date information to the card issuer
Output:
OK or error message indicating that the card is invalid

Extreme Programming

- **Pair Programming**
  - programmers work in pairs, sitting together to develop code
    - helps develop common ownership of code and
    - spreads knowledge across the team
  - serves as an informal review process as each line of code is looked at by more than 1 person
  - encourages refactoring
  - however,
    - measurements suggest that development productivity with pair programming is similar to that of two people working independently

Assessment of Agile Methods

- **Problems**
  - it can be difficult to keep the interest of customers who are involved in the process
  - team members may be unsuited to the intense involvement that characterises agile methods
  - prioritising changes can be difficult where there are multiple stakeholders
  - maintaining simplicity requires extra work
  - contracts may be a problem (as with other approaches to iterative development)
  - no requirements and design documentation available (the code is the documentation), therefore not suitable for long-running, evolving projects
  - limited to small project teams
    - 5 people?
    - XP thinks it is applicable for teams up to 20 programmers?

Process Certification and Assessment

Possible **certification** strategies to increase / assure software quality:

- Certify **product**: very expensive, but required by US FAA for **safety-critical** portions of avionics software systems.

  Ensures very high SQ.

- Certify **producer** (the software engineer): Not yet in place, but professional organizations working on accreditation rules for software engineers (IEEE, PEO).

  Would ensure that SW is written by people familiar with SQ measures. No guarantee, but engineering ethics rules apply.

- Certify **production process**: currently most frequently practiced $\rightsquigarrow$ ISO 9000, SEI-CMM.

  Driven by market forces, but does not in general ensure SQ. (Least efficient of the three strategies, but affordable).

Process Certification and Assessment

Capability Maturity Model (CMM)

- Developed at the **Software Engineering Institute** (SEI) at Carnegie Mellon University.
- Mid-80ies: US DoD interested in assessing capabilities of major software contractors $\rightsquigarrow$ CMM.
- Classifies the software process of a given company into one of 5 levels of maturity.

SEI Capability Maturity Model (CMM)

Continuous process improvement

Quality quantitatively assessed/controlled

Process documented and standardized

Some process discipline and tracking

Ad-hoc, chaotic, the heroic programmer

SEI Capability Maturity Model (CMM)

State for the Art (1994)

Out of 261 investigated organizations

- 75% at **level 1**,
- 24% at **level 2 or 3**,
- 2 at **level 5**.

And: US Air Force stipulates that any sw contractor must achieve CMM level 3 by 1998.

Some success stories:

- SE division of Hughes in Fullerton/CA: $ .5 Mio. investment in assessment and improvement, level 2 to level 3, annual savings $ 2 Mio. per year.
- Raytheon invested $ 1 Mio. per year in raising CMM-level and training 400 programmers:
  - level 1 to level 3 in 3 years,
  - productivity doubled,
  - return of $ 7.80 on every invested dollar,
  - most projects finished within time and budget.