Software Engineering

D. Vermeir

September 2009

Contents

1. Introduction
2. The Software Engineering Process
3. Project Management
4. Requirements Analysis
5. Design
6. Implementation
7. Integration and Testing

Part I

Introduction

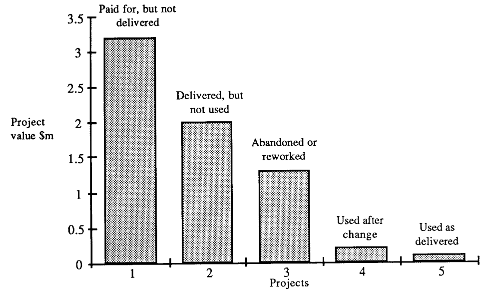
Introduction

1. Software Engineering
2. Software Engineering Activities
3. Software Project Artifacts
4. Software Project Quality

Introduction

1. Software Engineering
2. Software Engineering Activities
3. Software Project Artifacts
4. Software Project Quality

DOD real time systems

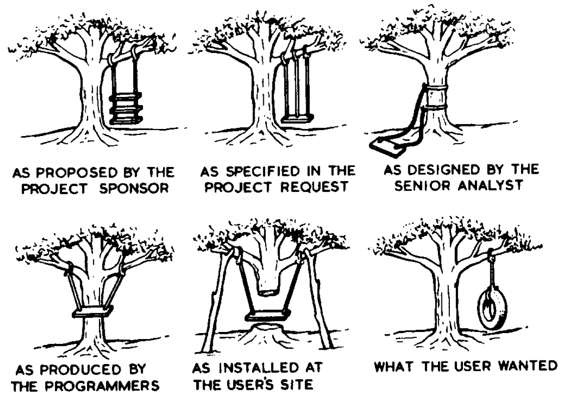


- Late, over budget
- Not according to expectations
- Contains errors
- Compare with other engineering disciplines

Standish report (all software, 2001)

- US spents 275 billion $ per year on software development
- 23% of projects are cancelled (after considerable expenditure) without delivering anything
- An average "successful" project:
  - is 45% over budget
  - is 63% over schedule
  - delivers 67% of the originally planned features and functions.

Software Engineering



- Definition of **what**: *requirements & specifications*
- Preservation of semantics when specifying **how**: *design & implementation*

Successful Projects

Name	LOC	Files	Directories
Linux kernel	9,257,383	23,810	1,417
gcc 4.4	10,187,740	38,264	3,563
KDE 4.0	25,325,252	100,688	7,515
Gnome 2.23	4,780,168	8,278	573
X Window System	21,674,310	14,976	1,023
Eclipse 3.4	94,187,895	912,309	297,500

Software Engineering: Definitions

- "The technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates." *(Fairley 1985)*
- "The practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required to develop, operate and maintain them." *(Boehm 1976)*
- *keywords*:
  - management, cost
  - development and maintenance
  - documentation
  - according to expectation

Introduction

1. Software Engineering
2. Software Engineering Activities
3. Software Project Artifacts
4. Software Project Quality

Software Engineering Activities

- Defining the software development **process** to be used.
- **Managing** the **project**.
- Describing the intended product.
- Designing the product.
- Implementing the product.
- Testing the parts of the product.
- Integrating and testing the parts of the product.
- Maintaining the product.

Thus. . . P^4

- People
- Process
- Project
- Product

Introduction

1 Software Engineering
2 Software Engineering Activities
3 Software Project Artifacts
4 Software Project Quality

Project Artifacts

- Requirements specification.
- Software architecture documentation.
- Design documentation.
- Source code.
- Test procedures, cases, ...

All under **configuration management**.

Introduction

1 Software Engineering
2 Software Engineering Activities
3 Software Project Artifacts
4 Software Project Quality

Quality: how to achieve

- Inspections.
- Formal methods.
- Testing.
- Project control techniques:
  - Predict cost.
  - Manage risks.
  - Control artifacts (configuration management).

Part II

The Software Engineering Process

The Software Engineering Process

5 A typical roadmap

6 Perspectives on software engineering

7 Key expectations (Humphrey)

8 Process alternatives

9 Documentation and Configuration Management

10 Quality

11 Capability assessment

The Software Engineering Process

5 A typical roadmap

6 Perspectives on software engineering

7 Key expectations (Humphrey)

8 Process alternatives

9 Documentation and Configuration Management

10 Quality

11 Capability assessment

A typical roadmap

1. Understand nature and scope of the product
2. Select process and create plan(s).
3. Gather requirements
4. Design and build the product.
5. Test the product.
6. Deliver and maintain the product.

A typical roadmap

1. Understand nature and scope of the product
2. Select process and create plan(s).
3. Gather requirements
4. Design and build the product.
5. Test the product.
6. Deliver and maintain the product.

A typical roadmap

1. Understand nature and scope of the product
2. Select process and create plan(s).
3. Gather requirements
4. Design and build the product.
5. Test the product.
6. Deliver and maintain the product.

Select process and create plan(s).

- Determine means to keep track of changes to documents and code

Configuration Management

- Develop overall plan for the project, including a schedule.

Software Project Management Plan

A typical roadmap

1. Understand nature and scope of the product
2. Select process and create plan(s).
3. Gather requirements
4. Design and build the product.
5. Test the product.
6. Deliver and maintain the product.

Gather requirements

- By communicating with the stakeholders (sponsor, user, ...).
- Steps (3, gather requirements) and (4, design and build) may be repeated, depending on the selected process.

A typical roadmap

1. Understand nature and scope of the product
2. Select process and create plan(s).
3. Gather requirements
4. Design and build the product.
5. Test the product.
6. Deliver and maintain the product.

A typical roadmap

1. Understand nature and scope of the product
2. Select process and create plan(s).
3. Gather requirements
4. Design and build the product.
5. Test the product.
6. Deliver and maintain the product.

A typical roadmap

1. Understand nature and scope of the product
2. Select process and create plan(s).
3. Gather requirements
4. Design and build the product.
5. Test the product.
6. Deliver and maintain the product.

Maintenance.

Consumes up to 80% of the budget.

The Software Engineering Process

5 A typical roadmap
6 Perspectives on software engineering
7 Key expectations (Humphrey)
8 Process alternatives
9 Documentation and Configuration Management
10 Quality
11 Capability assessment

Perspectives on software engineering

1. Structured programming
2. Object-oriented programming
3. Reuse and components
4. Formal methods

Perspectives on software engineering

1. Structured programming
2. Object-oriented programming
3. Reuse and components
4. Formal methods

Structured programming

- Top-down development method.
- (Recursively) decompose functions into smaller steps, using a limited set of composition patterns: **while**, **if . . . else . . .** , sequence, **not** goto.
- Influenced control statements in programming languages.
- Stress functionality, not data.
- Sensitive to change in requirements (e.g. change in data representation . . .)

Perspectives on software engineering

1. Structured programming
2. Object-oriented programming
3. Reuse and components
4. Formal methods

Object-oriented programming

- Encapsulates data in ADT.
- Correspondence with “real” application objects.
- Easier to understand, evolve.
- **Design patterns** can be used to describe reusable design solutions.

Perspectives on software engineering

1. Structured programming
2. Object-oriented programming
3. Reuse and components
4. Formal methods

Reuse and components

- Compare with other engineering disciplines (e.g. car models).
- Reuse should be aimed for from the start:
  - design **modular** systems with future reuse in mind
  - knowledge of what is available for reuse
- See also: components (javabeans, COM: reuse binaries) and frameworks.

Perspectives on software engineering

1. Structured programming
2. Object-oriented programming
3. Reuse and components
4. Formal methods

Formal methods

- Compare with other engineering disciplines that have a solid supporting base in mathematics.
- Formal specifications: use (first order) logic $\Rightarrow$ unambiguous, can be formally studied (e.g. consistency).
- Formal transformations: from specifications over design to code $\Rightarrow$ code is guaranteed to be equivalent with specifications.

The Software Engineering Process

5 A typical roadmap
6 Perspectives on software engineering
7 Key expectations (Humphrey)
8 Process alternatives
9 Documentation and Configuration Management
10 Quality
11 Capability assessment

Key expectations (Humphrey)

- Predetermine quantitative quality goals
  - E.g. “500 lines/mm”, “$<$ 3 defects/Klines”.
- Accumulate data for use in subsequent projects (and estimations).
- Keep all work visible (to everyone involved in the project).
- Design only against requirements; program only against design; test only against design and requirements.
- Measure (and achieve) quality goals.

The Software Engineering Process

5 A typical roadmap
6 Perspectives on software engineering
7 Key expectations (Humphrey)
8 Process alternatives
9 Documentation and Configuration Management
10 Quality
11 Capability assessment

Process alternatives

1. The waterfall process model
2. The spiral model
3. The incremental process model
4. Trade-offs

Process alternatives

1. The waterfall process model
2. The spiral model
3. The incremental process model
4. Trade-offs

The waterfall process model

1. **Requirements analysis** produces specification (text).
2. **Design** produces diagrams & text.
3. **Implementation** produces code & comments.
4. **Test** produces test reports and defect descriptions.

The extended waterfall model

1. **Requirements analysis**:
   - Concept analysis: overall definition of application philosophy.
2. **Design** produces diagrams & text.
   - Architectural design.
   - Object-oriented analysis: determine key classes.
   - Detailed design.
3. **Implementation** produces code & comments.
4. **Test** produces test reports and defect descriptions.
   - Unit testing.
   - Integration testing.
   - Acceptance test.

Process alternatives

1. The waterfall process model
2. The spiral model
3. The incremental process model
4. Trade-offs

The spiral model

- Several waterfall cycles.
- Motivation:
  - Early retirement of risk.
  - Partial versions to show to the customer for feedback.
  - Avoid “big bang” integration.

Process alternatives

1. The waterfall process model
2. The spiral model
3. The incremental process model
4. Trade-offs

The incremental process model

(and derivatives such as extreme programming)

- One cycle per time unit (e.g. week).
- “Sync and stabilize” (e.g. daily build).
- Continual process.
- Architecture must be stable, configuration management must be excellent.
- See also: extreme programming.

Extreme programming

A project management and development methodology created by K. Beck.

reasonable	**extreme**
customer separated	customer on team
up-front design	evolving design
built for future too	just in time
complexity allowed	radical simplicity
tasks assigned	tasks self-chosen
developers isolated	pair programming
infrequent integration	continuous integration
limited communication	continual communication

Process alternatives

1. The waterfall process model
2. The spiral model
3. The incremental process model
4. Trade-offs

Trade-offs

factor	**waterfall**	**spiral**	**incremental**
Ease of documentation control	easier	harder	harder but..
Enable customer interaction	harder	easier	easier
Promote good design	medium easier	easier	harder
Leverage metrics in project	harder	medium easier	medium easier

The Software Engineering Process

5 A typical roadmap

6 Perspectives on software engineering

7 Key expectations (Humphrey)

8 Process alternatives

9 Documentation and Configuration Management

10 Quality

11 Capability assessment

Documentation and Configuration Management

1. Introduction
2. Documentation Standards
3. An Approach
4. Document management
5. Configuration Management

Documentation and Configuration Management

1. Introduction
2. Documentation Standards
3. An Approach
4. Document management
5. Configuration Management

Documentation introduction

- Usual rules about documenting code.
- In addition, context should be documented.
  - Relationship of code/class design to requirements. (*Implements requirement 1.2*)
- A project is the whole set of coordinated, well-engineered artifacts, including the documentation suite, the test results and the code.

Documentation and Configuration Management

1. Introduction
2. Documentation Standards
3. An Approach
4. Document management
5. Configuration Management

Documentation standards

- Standards improve communication among engineers.
- To be effective, standards must be perceived by engineers as being helpful to them

⇒ Let development team decide on standards to apply.
  - \+ Motivation.
  - \- Groups with different standards in organization: makes process comparison & improvement (CMM) more difficult.

⇒ Standards should be simple and clear.

Organizations that publish standards

- IEEE (International Institute of Electronic and Electrical Engineering), ANSI (American National Standards Institute).
- ISO (International Standards Organization, pushed by EU)
- SEI (Software Engineering Institute): e.g. CMM (Capability Maturity Model).
- OMG (Object Management Group, approx. 700 company members): UML (Unified Modeling Language).

IEEE project documentation set

- SVVP – Software Validation & Verification Plan (often by external organization).
- SQAP – Software Quality Assurance Plan.
- **SCMP** – Software Configuration Management Plan.
- **SPMP – Software Project Management Plan**.
- **SRS** – Software Requirements Specification.
- SDD – Software Design Document.
- Source code.
- STD – Software Test Document.
- User manuals.

Documentation and Configuration Management

1. Introduction
2. Documentation Standards
3. An Approach
4. Document management
5. Configuration Management

One way to define documentation needs

1. Specify how documents and code will be accessed ⇒ SCMP
2. Specify who will do what when ⇒ SPMP
3. Document what will be implemented ⇒ SRS
4. Document design ⇒ SDD
5. Write & document code.
6. Document tests performed so that they can be run again (STD):
7. Decide for each document how it will evolve: update or append.

Documentation and Configuration Management

1. Introduction
2. Documentation Standards
3. An Approach
4. Document management
5. Configuration Management

Document management

Document management requires

- Completeness (e.g. IEEE set).
- Consistency.
  - single-source documentation: specify each entity in only one place (as much as possible, e.g. user manual..).
  - use hyperlinks, if possible, to refer to entities.
- Configuration (coordination of versions).

Documentation and Configuration Management

1. Introduction
2. Documentation Standards
3. An Approach
4. Document management
5. Configuration Management

The Configuration Management Plan

- The SCMP specifies how to deal with changes to documents: e.g. *To change the API of a module, all clients must be asked for approval by email.*
- Use system to keep track of *configuration items* and valid combinations thereof.
- See **CVS**: check-in, check-out, tagging procedures.
- Example SCMP: page 63 . . . in book.

The Software Engineering Process

5 A typical roadmap

6 Perspectives on software engineering

7 Key expectations (Humphrey)

8 Process alternatives

9 Documentation and Configuration Management

10 Quality

11 Capability assessment

Quality

1. Quality attributes
2. Quality metrics
3. Quality assurance
4. Inspections
5. Verification and Validation

Quality

1. Quality attributes
2. Quality metrics
3. Quality assurance
4. Inspections
5. Verification and Validation

Quality attributes

- Quality attributes for code (function):
  - Satisfies stated requirements.
  - Checks inputs, reacts predictably to illegal input.
  - Has been inspected by others.
  - Has been tested thoroughly.
  - Is thoroughly documented.
  - Has confidently known defect rate, if any.
- Quality attributes for design:
  - Extensible (to provide additional functionality).
  - Evolvable (to accommodate altered requirements).
  - Portable (applicable to several environments).
  - General and reusable (applicable to several situations).

Quality

1. Quality attributes
2. Quality metrics
3. Quality assurance
4. Inspections
5. Verification and Validation

Quality metrics

- Quantification is essential part of engineering.
- Metrics only make sense in context (to compare): e.g. different amount of lines of code needed by different programmers to implement the same function. Lines of code becomes meaningful again when taken over a large number of samples.
- Example metrics
  - Amount of work (lines of code).
  - Time spent on work (lines of code).
  - Defect rate (e.g. number of defects per KLOC, per page of documentation, . . .)
  - Subjective evaluation (quality: 1 . . . 5).
- Goals specify desired values of metrics.

Quality

1. Quality attributes
2. Quality metrics
3. Quality assurance
4. Inspections
5. Verification and Validation

The quality assurance process

- Reviews: SCMP, process, SPMP
- Inspections: requirements, design, code, . . .
- Testing
  - Black box.
  - White (glass) box.
  - Grey box.
- Ideally, by external organization.

Quality

1. Quality attributes
2. Quality metrics
3. Quality assurance
4. Inspections
5. Verification and Validation

Inspections

- White box technique.
- Principle:

  Authors can usually repair defects that they recognize.
  ⇒ Help authors to recognize defects before they deliver.
  ⇒ Have peers seek defects.
- Examine part of project
- **Much more efficient than testing**:
  - Time spent per fault is (much) less than with testing.
  - Earlier detection: easier to fix.

Rules about inspections

- Defect **detection** only.
- Peer (not supervisor-subordinate) process.
- Only **best effort** of author should be examined.
- Specified roles:
  - Moderator (is also inspector).
  - Author (is also inspector, answers questions)
  - Reader (is also inspector): leads team through the work.
  - Recorder (is also inspector).
- Inspectors should **prepare** the inspection.

The inspection process

1. *Plan*: which metrics to collect, tools for recording, . . .
2. Optional *overview* meeting to decide who inspects what.
3. *Preparation*: inspectors review work, note possible defects (perhaps in common database).
4. The *meeting* (1-3 hours).
5. Author repairs defect (*rework*).
6. Optional *causal analysis* meeting if (some) defects due to widespread misconception.
7. *Follow-up* (meeting?) to confirm that defects have been fixed.

Example

Inspecting requirements.

faulty *If the temperature is within* 5.02% *of the maximum allowable limit, as defined by standard* 67892*, then the motor is to be shut down.*

orrect *If the temperature is within* 5.02% *of the maximum allowable limit, as defined by standard* 67892*, then the motor is to be powered down.*

! "shut down" $\neq$ "power down"

! Very expensive to find and fix after implementation.

One way to prepare & conduct inspections

- Build inspections into schedule (time for preparation, meeting).
- Prepare for collection of inspection data.
  - Number of defects/KLOC, time spent.
  - Form, e.g. with *description*, *severity*.
  - Who keeps inspection data, usage of . . .
- Assign roles.E.g. author, moderator/recorder, reader or, minimally, author/inspector.
- Ensure that each participant prepares: bring filled defect forms to meeting.

Quality

1. Quality attributes
2. Quality metrics
3. Quality assurance
4. Inspections
5. Verification and Validation

Verification and Validation

- **Validation**: are we building the right product?

  Test product (does it actually work)?
- **Verification**: are we building the product right (process, should it work "on paper")?
  - Do the requirements express what the customer wants? (inspection requirements, . . .)
  - Does the code implement the requirements? (inspection)
  - Do the tests cover the application (inspect STD).

Example SQAP

Page 68 – 72 and 112 – 113 in book.

The Software Engineering Process

5 A typical roadmap

6 Perspectives on software engineering

7 Key expectations (Humphrey)

8 Process alternatives

9 Documentation and Configuration Management

10 Quality

11 Capability assessment

Capability assessment

1. Personal Software Process (PSP)
2. Team Software Process (TSP)
3. Capability Maturity Model (CMM)

Capability assessment

1. Personal Software Process (PSP)
2. Team Software Process (TSP)
3. Capability Maturity Model (CMM)

Personal Software Process (PSP)

PSP0 **Baseline Process**: current process with basic measurements taken. Track time spent, record defects found, record the types of defects.

PSP1 **Personal Planning Process**. PSP0 + ability to estimate size, framework for reporting test results.

PSP2 **Personal Quality Management Process**: PSP1 + personal design and code reviewing.

PSP3 **Cyclic Personal Process**: scale PSP2 to larger units: regression testing, apply PSP to each increment.

Capability assessment

1. Personal Software Process (PSP)
2. Team Software Process (TSP)
3. Capability Maturity Model (CMM)

Team Software Process (TSP)

Objectives:

- Build self-directed teams (3-20 engineers) that establish own goals, process, plans, track work.
- Show managers how to manage teams: coach, motivate, sustain peak performance.
- Accelerate CMM improvement.
- ...

Capability assessment

1. Personal Software Process (PSP)
2. Team Software Process (TSP)
3. Capability Maturity Model (CMM)

Capability Maturity Model (CMM)

CMM1 **Initial**: undefined ad-hoc process, outcome depends on individuals (heroes).

CMM2 **Repeatable**: track documents (CM), schedule, functionality. Can predict performance of same team on similar project.

CMM3 **Defined**: CMM2 + documented standard process that can be tailored.

CMM4 **Managed**: CMM3 + ability to predict quality & cost of new project, depending on the attributes of its parts, based on historical data.

CMM5 **Optimized**: CMM4 + continuous process improvement, introduction of innovative ideas and technologies.

Part III

Project Management

Project Management

12 Introduction

13 Teams

14 Risk Management

15 Choosing tools and support

16 Planning

17 Feasability Analysis

18 Cost Estimation

19 The Project Management Plan

Project Management

12 Introduction

13 Teams

14 Risk Management

15 Choosing tools and support

16 Planning

17 Feasability Analysis

18 Cost Estimation

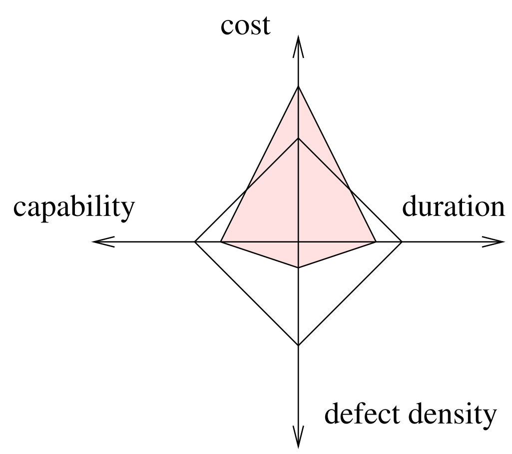
19 The Project Management Plan

Project Management Variables

- Total cost of the project (*increase expenditure*).
- Capabilities of the product (*remove feature*).
- Quality of the product (*increase MTTF*).
- Duration of the project (*modify schedule*).

Project Management Variables

Project management deals with trade-offs among the variables.



Project Management Road Map

1. Understand project content, scope and time frame.
2. Identify development process (methods, tools, . . .).
3. Identify managerial process (team structure)
4. Develop schedule.
5. Develop staffing plan.
6. Begin risk management.
7. Identify documents to be produced.
8. Begin process itself.

Professionalism in software engineering

Professionals have societal responsibilities that supersede their requirements to satisfy the needs of their employers and supervisors.

- E.g. life-critical systems.
- E.g. billing software: public should not have to check every computation for correctness.

Conducting meetings

1. Distribute start & end time, agenda (important items first).
2. Prepare strawman items.
3. Start on time.
4. Have someone record items.
5. Get agreement on agenda and timing.
6. Watch timing throughout and end on time.
    - Allow exceptions for important discussions.
    - Stop excessive discussion; take off line.
7. Keep discussion on the subject.
8. E-mail action items and decision summary.

Specifying agendas

1. Get agreement on agenda and time allocation.
2. Get volunteers to record decisions and action items.
3. Report progress on project schedule – 10 mins.
4. Discuss strawman artifacts – n mins.
5. Discuss risk retirement – 10 mins.
6. . . . (e.g. metrics, process improvement).
7. Review action items – 5 mins.

Project Management

12 Introduction

13 Teams

14 Risk Management

15 Choosing tools and support

16 Planning

17 Feasability Analysis

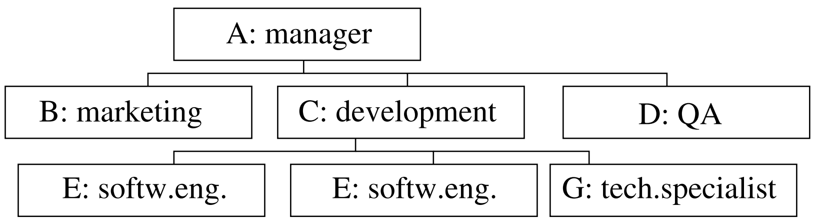
18 Cost Estimation

19 The Project Management Plan

Team structure

Influences amount of necessary communication.

- Hierarchical.



- Community of peers with equal authority.
- Horizontal peers with designated leader.
- Peer groups communicating via leaders (for larger projects).

Example team organization (1/2)

1. Select team leader: ensures all project aspects are active, fills any gaps.
2. Designate leader roles and responsibilities:
   - team leader (SPMP)
   - configuration management leader (SCMP)
   - quality assurance leader (SQAP, STD)
   - requirements management leader (SRS)
   - design leader (SDD)
   - implementation leader (code base)

Example team organization (2/2)

3. leader responsibilities:
   - propose strawman artifact (e.g. SRS, design)
   - seek team enhancement and acceptance
   - ensure designated artifact maintained and observed
   - maintain corresponding metrics, if any
4. Designate backup for each leader. E.g. team leader backs up implementation leader, CM leader backs up team leader etc.

Project Management

12 Introduction

13 Teams

14 Risk Management

15 Choosing tools and support

16 Planning

17 Feasability Analysis

18 Cost Estimation

19 The Project Management Plan

Identifying and retiring risks

A *risk* is something which may occur in the course of a project and which, under the worst outcome, would affect it negatively and significantly.
There are 2 types of risks:

- Risks that can be avoided or worked around (*retired*), e.g. “project leader leaves”; retire by designating backup person.
- Risks that cannot be avoided.

Risk management activities

1. Identification: continually try to identify risks.
   Sources:
   - Lack of top management commitment.
   - Failure to gain user commitment.
   - Misunderstanding of requirements.
   - Inadequate user involvement.
   - Failure to manage end-user expectations.
   - Changing scope and/or requirements.
   - Personnel lack required knowledge or skills.
2. Retirement planning.
3. Prioritizing.
4. Retirement or mitigation.

Risk retirement

Risk retirement is the process whereby risks are reduced or eliminated:

- risk avoidance: change project so that risk is no longer present; e.g. switch to programming language where team has experience.
- risk conquest: change project so that risk is no longer present; e.g.
  - buy training for the new programming language
  - use rapid prototyping to verify suitability of external library

Risk retirement planning

Retirement planning involves prioritizing of risks, e.g. based on $(11 - p) \times (11 - i) \times c$ where lower numbers represent higher priority.

- likelihood $p \in [1 \ldots 10]$, 1 is least likely.
- impact $i \in [1 \ldots 10]$, 1 is least impact.
- retirement cost $c \in [1 \ldots 10]$, 1 is lowest cost.

But leave room for exceptional cases, or risks where the retirement has a long lead time.

Risk management roadmap (1/2)

1. Each team member spends 10 mins. exploring her greatest fears for the project's success (in advance of meeting).
2. Each member specifies these risks in concrete language, weighs them, writes retirement plans and emails to the team leader.
3. The team leader integrates and prioritizes the results.
4. The team spends 10 mins. seeking additional risks.
5. The team spends 10 mins. finalizing the risk table (e.g. p. 89), which include responsible retirement engineers.

Risk management roadmap (2/2)

6. Responsible engineers do retirement work.
7. Team reviews risks for 10 mins. at weekly meetings:
   - responsible engineers report progress
   - team discusses new risks and adds them

Project Management

12 Introduction
13 Teams
14 Risk Management
15 Choosing tools and support
16 Planning
17 Feasability Analysis
18 Cost Estimation
19 The Project Management Plan

Choosing development tools and support

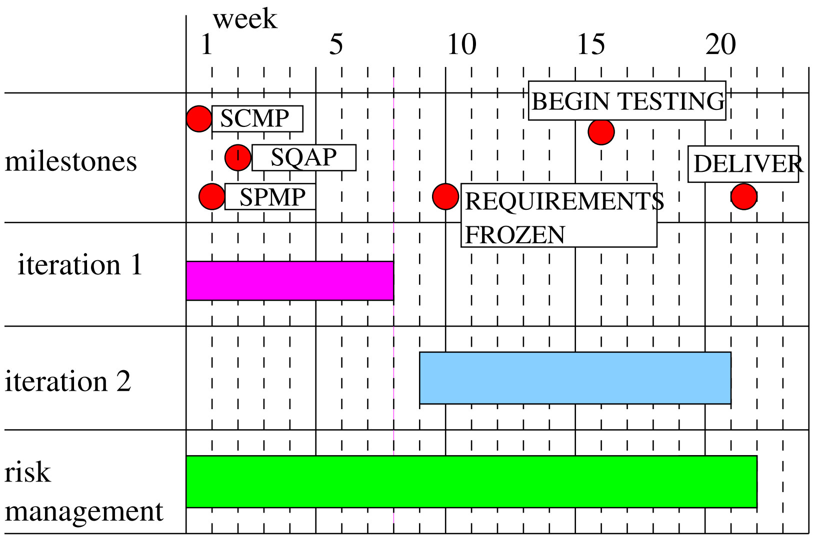
- project management: for scheduling, work breakdown. (e.g. trac, . . .)
- configuration management (e.g. cvs, . . .)
- managing requirements (docbook, latex)
- drawing designs (doxygen, dia, . . .)
- tracing tools: requirements → design → code (?)
- testing (e.g. junit, automake, dejagnu, . . .)
- maintenance (e.g. gnats, bugzilla, . . .)
- build (e.g. maven, ant, make, . . .)

Make build vs. buy decisions based on total cost comparison.

Project Management

12 Introduction
13 Teams
14 Risk Management
15 Choosing tools and support
16 Planning
17 Feasability Analysis
18 Cost Estimation
19 The Project Management Plan

High level planning



Assumes 2 iterations.

Making an initial schedule

1. External milestones (e.g. *delivery*).
2. Internal milestones to back up external ones (*start testing*).
3. Show first iteration, establish minimal capability (exercising process itself)
4. Show task for risk identification & retirement.
5. Leave some unassigned time.
6. Assign manpower (book p. 94).

The schedule will become more detailed as the project progresses and the schedule is revisited.

Project Management

12 Introduction
13 Teams
14 Risk Management
15 Choosing tools and support
16 Planning
17 Feasability Analysis
18 Cost Estimation
19 The Project Management Plan

Feasibility Analysis

Analysis based on the means (costs) and benefits of a proposed project. These are computed over the lifetime (incl. the development) of the system.
Estimate:

- Lifetime (e.g. 5yrs).
- Costs: development (see below), maintenance, operation.
- Benefits: savings, productivity gains, increased production, etc.

How to compare costs vs benefits?

- Net present value.
- Internal rate of return.
- Pay-back period.

Example

New system for stock management. Lifetime: 5 years.
Estimated costs (development) and benefits (= profit - operation costs)

Year	Costs	Benefits
0	5000	0
1		2500
2		2500
3		2500
4		2500
5		2500
	5000	12500

Net Present Value

Value F after n years of an investment of P at an interest rate i:

$$F = P \times (1 + i)^n$$

Present value of an amount F, available after n years, with an assumed interest rate i:

$$P = \frac{F}{(1 + i)^n}$$

Example (cont'ed)

Assume an interest rate of 12% ($i = 0.12$).

Year	Benefits		Costs	
0	0	0	5000	**5000**
1	2500	**2234**		
2	2500	**1993**		
3	2500	**1779**		
4	2500	**1589**		
5	2500	**1419**		
	12500	**9014**	5000	**5000**

Present value of profit:

$$NPV = 9014 - 5000 = 4014$$

Pay-back Period

Time needed for net present value of accumulated profits to exceed the value of the investment (initial cost).
In the example, the the pay-back period is 3 years.

Year	Benefits		Costs	
0	0	0	5000	**5000**
1	2500	**2234**		
2	2500	**1993**		
3	2500	**1779**		
4	2500	**1589**		
5	2500	**1419**		
	12500	**9014**	5000	**5000**

Internal Rate of Return

Assume that the initial cost is invested and that each year, the benefits are taken out, until nothing remains after the lifetime of the system. What interest rate i is needed to accomplish this?
$\Rightarrow$ The sum of the present value (at an interest rate i) of the benefits must equal the initial cost:

$$P = \Sigma_{j=1}^{j=n} F_j [\frac{1}{(1+i)^j}]$$

where F_j is the benefit in year j.
$\Rightarrow$ Compute the solution of

$$\Sigma_{j=0}^{j=n} F_j X^j = 0$$

where $F_0 = -P$ and $X = \frac{1}{1+i}$
In the example, the internal rate of return is $\pm 41\%$

Project Management

12 Introduction
13 Teams
14 Risk Management
15 Choosing tools and support
16 Planning
17 Feasability Analysis
18 Cost Estimation
19 The Project Management Plan

Estimating costs

- Even before architecture and design! ((Compare: $N€/m^3$ in building industry).
- Cost estimates can be refined later in the project.
- Based on KLOC estimate.
- KLOC estimate based on experience, outline architecture, ... or on function points estimation (see book p. 97 – 104).
- KLOC $\rightarrow$ cost using COCOMO.

The COCOMO cost model (Boehm)

The COCOMO model distinguishes 3 types of projects:

- **simple**: small team, familiar environment, familiar type of application
- **moderate**: experienced people, unfamiliar environment or new type of application
- **embedded**: rigid constraints, application embedded in complex hard/software system, rigid requirements, high validation requirements, little experience

Effort estimation using COCOMO

type	person-months
simple	$PM = 2.4 \times KLOC^{1.05}$
moderate	$PM = 3 \times KLOC^{1.12}$
embedded	$PM = 3.6 \times KLOC^{1.2}$

Note: KLOC excludes comments and support software (e.g. test drivers); $1 PM = 152 hrs$, excluding vacations, training, illness.

Estimating duration using COCOMO

Duration: *TDEV* (in months).

type	duration
simple	$TDEV = 2.5 \times PM^{0.38}$
moderate	$TDEV = 2.5 \times PM^{0.35}$
embedded	$TDEV = 2.5 \times PM^{0.32}$

COCOMO example 1

Simple project, 32,000 lines:

- $PM = 2.4 \times (32)^{1.05} = 91$
- $TDEV = 2.5(91)^{0.38} = 14$
- Number of people needed:

$$N = \frac{PM}{TDEV} = \frac{91}{14} = 6.5$$

COCOMO example 2

Example: embedded system, 128,000 lines:

- $PM = 3.6 \times (128)^{1.2} = 1216$
- $TDEV = 2.5 \times (1216)^{0.32} = 24$
- Number of people:

$$N = \frac{PM}{TDEV} = \frac{1216}{24} = 51$$

Intermediate COCOMO (1/2)

The basic COCOMO model yields a rough estimate, based on assumptions about productivity:

- 16 LOC/day for simple projects
- 4 LOC/day for embedded projects

Intermediate COCOMO (2/2)

Based on *PM*, *TDEV* from the basic model, in the intermediate model, the basic *PM* estimate is multiplied with factors depending on:

- Product attributes: reliability, database size, complexity.
- Computer attributes: resource constraints, stability hard/software environment.
- Personnel attributes: experience with application, programming language, etc.
- Project attributes: use of software tools, project schedule.

The model can be calibrated, based on experience and local factors.

Project Management

12 Introduction

13 Teams

14 Risk Management

15 Choosing tools and support

16 Planning

17 Feasability Analysis

18 Cost Estimation

19 The Project Management Plan

The SPMP (1/3)

1. Introduction
   1. Project overview.
   2. Project deliverables.
   3. Evolution of the SPMP.
   4. Reference materials.
   5. Definitions and acronyms.
2. Project organization
   1. Process model (e.g. spiral, 2cycles).
   2. Organizational structure (roles, no names).
   3. Organizational boundaries and interfaces (e.g. with customer, marketing, . . .).
   4. Project responsibilities (of various roles).

The SPMP (2/3)

3. Managerial process.
   1. Objectives and priorities (e.g. safety before features).
   2. Assumptions, dependencies and constraints.
   3. Risk management.
   4. Monitoring and controlling mechanisms (who, (e.g. Sr. management)? when? how? will review)
   5. Staffing plan (names for roles)
4. Technical process.
   1. Methods, tools and techniques (e.g. C++, design patterns, . . .)
   2. Software documentation (refer to SQAP)
   3. Project support functions (e.g. DVD will consult on . . .)

The SPMP (3/3)

5. Work packages, schedule and budget.
   1. Work packages (sketchy before architecture is established)
   2. Dependencies.
   3. Resource requirements (estimates)
   4. Budget and resource allocation (person-days, money for S&HW)
   5. Schedule.

See example SPMP on p. 123 – 134.

Project Management

12 Introduction
13 Teams
14 Risk Management
15 Choosing tools and support
16 Planning
17 Feasability Analysis
18 Cost Estimation
19 The Project Management Plan

Quality in process management

- Establish process metrics and targets.
- Collect data.
- Improve process, based on data.

Example process metrics

- Number of defects per KLOC detected within 12 months of delivery.
- Variance in schedule on each phase: $\frac{duration_{actual} - duration_{projected}}{duration_{projected}}$
- Variance in cost $\frac{cost_{actual} - cost_{projected}}{cost_{projected}}$
- Total design time as % of total programming time.
- Defect injection and detection rates per phase. E.g. "one defect in requirements detected during implementation".

Defect detection rate by phase

Detection phase	**Injection Phase**		
	detailed requirements	design	implementation
detailed requirements	2 (5)		
design	0.5 (1.5)	3 (1)	
implementation	0.1 (0.3)	1 (3)	2 (2)

Numbers between parentheses are planned, others are actual results.

SQAP Part 2

- A table per phase (example on p. 113) containing actual data and company norms (or goals).
- The example on the next slide concerns requirements, 200 of which have been established in 22 hrs., a productivity of 9.9 requirements/hr.
- Since we found 6/100 defects by inspection, vs. the norm of 4/100, we can predict that the defect rate will be the same and thus there will be $6/4 \times r$, where r is the historic defect rate, defects in the requirements.

Metrics collection for requirements

	meeting	research	exec-ution	personal review	inspec-tion
hours spent	1 × 4	4	5	3	6
% of total time	10%	20%	25%	15%	30%
norm %	15%	15%	30%	15%	25%
quality (self-assessed)	2	8	5	4	6
defects/100				6	6
norm/100				3	4
hrs/requirement	0.01	0.02	0.025	0.015	0.03
norm hrs/req.	0.02	0.02	0.04	0.01	0.03

Part IV

Requirements analysis

Requirements analysis

21 Inroduction
22 Expressing Requirements
23 Rapid Prototyping
24 Detailed Requirements
25 Desired Properties of D-requirements
26 Organizing D-requirements
27 Metrics for Requirements

Requirements analysis

21 Inroduction
22 Expressing Requirements
23 Rapid Prototyping
24 Detailed Requirements
25 Desired Properties of D-requirements
26 Organizing D-requirements
27 Metrics for Requirements

Introduction

- A requirement is about **what**, not **how** (unless customer demands . . .)
- C (“customer”) requirements are intended mainly for the customer, in language that is clear to her.
- D (“detailed”) requirements are mainly intended for the developers, organized in a specific structure.
- SRS (System Requirements Document) (p. 140 for IEEE detail)
  1. Introduction.
  2. Overall description (C-requirements).
  3. Specific requirements (D-requirement).
  4. Supporting information.

Why requirements (document)?

- To verify (test, . . .) against
- For the engineers: to know what the goal is.
- “To write is to think”.
- Contract.

Each requirement must be

- expressed properly
- made easily accessible
- numbered
- accompanied by test that verifies it
- provided for in the design
- accounted for by code
- validated

C-requirements roadmap

1. Identify customer.
2. Interview customer representatives
3. Write C-requirements.
4. Inspect C requirements
5. Review with customer, iterate until approved.

Track metrics: time spent, quantity, self-assessed quality, defect rates from inspections.

Requirements sources

- People: less constrained.
- Other (e.g. physics): highly constrained.

Stakeholders

Example: e-commerce website application.

- Visitors.
- Management (e.g. requirements about tracking customers).
- Developers (e.g. learn about new technology).

Watch for inconsistent requirements due to different stakeholder interests.

Professional responsibilities

Do not accept requirements that are

- unrealistic (e.g. not within budget)
- untestable

especially for critical (e.g. medical) systems.

Customer interview

Compare architect - client.

1. List and prioritize customer interviewees.
2. Get strawman requirements from “primary” interviewees and solicit comments from others.
3. Schedule interview with fixed start, end time. At least two developers should attend.
4. Probe customer during interview.
5. Draft C-requirements.
6. Email result to customer(s).

Requirements analysis

21 Inroduction

22 Expressing Requirements

23 Rapid Prototyping

24 Detailed Requirements

25 Desired Properties of D-requirements

26 Organizing D-requirements

27 Metrics for Requirements

Expressing requirements

- Conceptual model.
- Use cases.
- Data Flow Diagrams.
- State transition diagrams.
- Class diagram or EAR diagram (for data).
- GUI mock screen shots (p.m.)

Use cases

An informal description of an interaction with the system (scenario).

- There should be a use-case for each system function.
- Frequently occurring sub-scenarios may be factored as separate use cases (e.g. “login”).
- Jacobson suggests deriving (domain) classes from use cases (via sequence diagrams).

Use case

A use-case consists of:

name

summary

actors involved (an actor is an entity that communicates with the system, e.g. a user, another system).

preconditions on the system’s state at the start of the case (informal)

description should be informal but complete, especially on the actor-system interaction (but not on details like GUI)

exceptions i.e. special cases

result i.e. postconditions (informal)

Example use case

name ATM withdrawal

summary Cash withdrawal from an account associated with a cash card.

actors customer

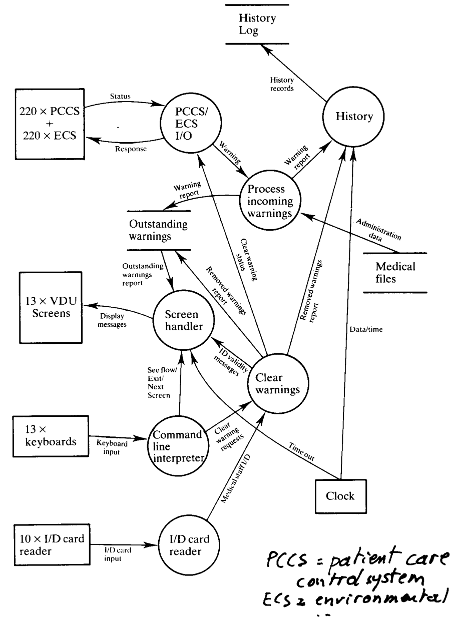
preconditions The customer has a valid cash card

description The customer inserts the cash card. The system prompts for a password and then verifies that the cash card corresponds to an existing account, and that the password is valid...

exceptions If the link is down, the ATM displays “out of order” message.

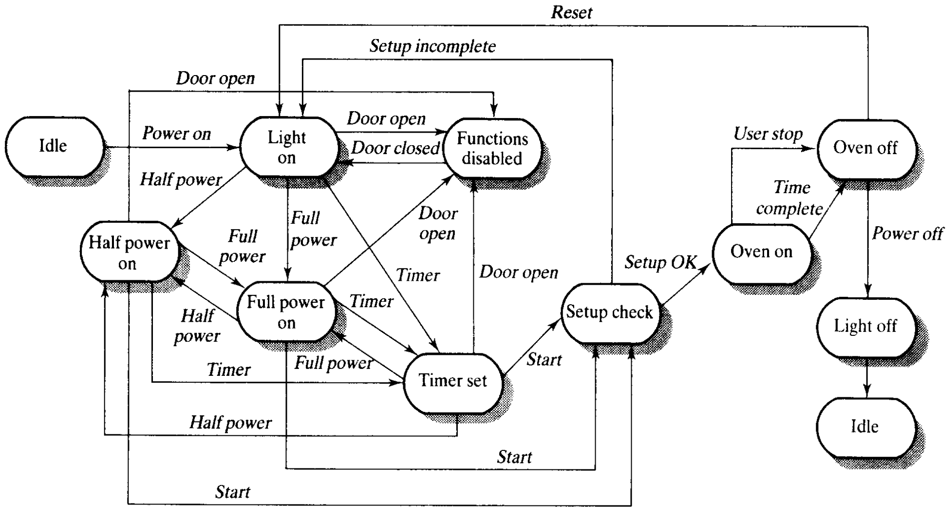
result The account corresponding to the cash card is updated.

Data Flow Diagrams



- information flow
- process
- data store
- data source/sink

State Transition Diagrams



State Transition Diagrams (cont’d)

State	Description
Half power on	The power output is set to half power when the half power button is pressed.
Light on	The indicator light is switched on showing that the oven is active.
Functions disabled	Function settings are disabled when the oven door is open. Included for safety reasons.
Setup check	Checks that a valid set of parameters has been set by the user.
Timer set	The timer is set to the user provided value.

Expressing C requirements

- If the requirement is simple and stands alone, express it in clear sentences within an appropriate section of the SRS.
- If the requirement is an interaction involving the application, express it via a use case.
- If the requirement involves process elements taking input and producing output, use a DFD.
- If the requirement involves states that (part of) the application can be in, use state transition diagrams.
- Use whatever else is appropriate (e.g. decision tables) . . . .

Requirements analysis

21 Inroduction
22 Expressing Requirements
23 Rapid Prototyping
24 Detailed Requirements
25 Desired Properties of D-requirements
26 Organizing D-requirements
27 Metrics for Requirements

Rapid prototyping

A rapid prototype is a partial implementation of the application, often involving GUI components.
Useful for:

- Eliciting customer comments (understanding requirements).
- Retiring risks.
- Proof of concept.

May be throw-away (scripts) or (partly) reusable.

To prototype or not?

Possible benefits (quantify in €)

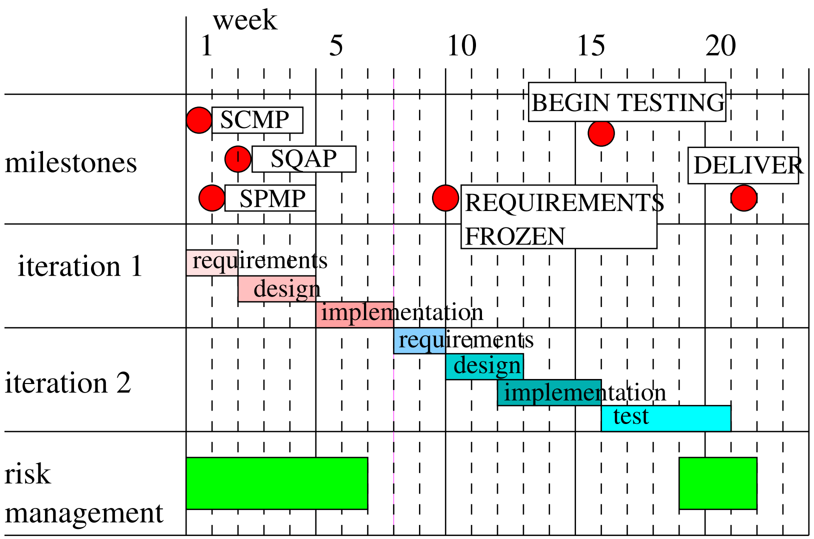
- Time wasted on requirements that turn out to be not really needed.
- Retiring risks (e.g. test new technology).
- Avoid having to rework because of wrong requirements.

Costs

- of developing prototype,
- \- money saved by expected reuse of (parts of) prototype.

See book p. 162-164.

Updating the project plan



Requirements analysis

21 Inroduction
22 Expressing Requirements
23 Rapid Prototyping
24 Detailed Requirements
25 Desired Properties of D-requirements
26 Organizing D-requirements
27 Metrics for Requirements

D(etailed) requirements

- Functional requirements.
- Nonfunctional requirements:
  - Performance: time, space (RAM,disk), traffic rate.
  - Reliability and availability.
  - Error handling
  - Interface requirements
  - Constraints (tool, language, precision, design, standards, HW)
- Inverse requirements

Functional requirements

2.3 Each submission has a *status* consisting of a set of reports submitted by PC members (see section 1) and a summarizing value which is one of . . . .

. . .

2.4 An author is able to view the status of her submissions via the website, after proper authorization, see section 3.

. . .

Performance requirements

4.1 Excluding network delays, the public web site component of the system will generate an answer to each request within a second, provided the overall load on the system does not exceed 1.4.

4.2 The size of the executable for the main application program (middle tier) will not exceed 6MB.

4.3 The size of the database will not exceed $n \times 2000$ bytes, excluding the size of the submissions and PC reports, where n is the number of submissions.

...

Reliability and availability requirements

Reliability:

7.1 The system shall experience no more than 1 level one faults per month.

...

Availability:

7.2 The system shall be available at all times, on either the primary or backup computer.

...

Error handling

7.3 The cgi program will always return a page. If the middle tier application does not return any data within the time frame mentioned in section 1, the cgi program will generate a page containing a brief description of the fault and the email address of the support organization.

7.3 There will be a maximum size, selectable by the administrator, of any page sent by the cgi program. If the system attempts to exceed this size, an appropriate error message will be sent instead.

...

Interface requirements

5.1 The protocol of the application server has the following syntax:

request	:	*N* `\n` [key = value`\n\n`]*
reply	:	*N* `\n` [key = value `\n\n`]*
name	:	any string not containing `\n` or “=”
value	:	any string not containing the sequence “`\n\n`”

where *N* indicates the number of (*key,value*) pairs in the message.

...

Constraints

9.1 The system will use the mysql database management system.
9.2 The target system is any linux system with a kernel v.2.4 or higher.
9.3 The cgi program will generate only html according to the WC3 standard v.2. No frames, style sheets, or images will be used, making it usable for text-only browsers.

...

Inverse requirements

What the system will *not* do.

10.1 The system will not provide facilities for backup. This is the responsibility of other programs.

...

Requirements analysis

21 Inroduction
22 Expressing Requirements
23 Rapid Prototyping
24 Detailed Requirements
25 Desired Properties of D-requirements
26 Organizing D-requirements
27 Metrics for Requirements

Desired properties of D-requirements

- Traceability.
- Testability and nonambiguity.
- Priority.
- Completeness.
- Consistency.

Traceability of D-Requirements

- Backward to C-requirements.
- Forward to
  - → Design (module, class)
  - → Code ((member) function)
  - → Test.

  Example: Req. 2.3 → `class SubmissionStatus` → Test 35
- Also for nonfunctional requirements: e.g. a performance requirement probably maps to only a few modules (90/10 rule). Example: req. 7.3 may map to a special subclass of `ostream` which limits output etc.

Testability and Nonambiguity

- "the system will generate html pages" is ambiguous

⇒ Specify exact standard or html-subset.

- "The system will have user-friendly interface"

⇒ Specify time to learn for certain category of users.

Priority

- Put each requirement in a category: "essential", "desirable" or "optional".
- 80% of benefits come from 20% of requirements.
- Should be consistent (e.g. essential requirement cannot depend on desirable one).
- The prioritization impacts the design.

Completeness

- Check that the requirements cover the use cases and the C-requirements.
- Specify *error conditions*: e.g. what does a function do when it receives bad input (in C++: throw exception).

How to write a D-requirement

1. Classify as functional/nonfunctional.
2. Size carefully: functional requirement ≈ (member) function.
3. Make traceable and testable, if at all possible.
4. Be precise: avoid ambiguity.
5. Give it a priority.
6. Check completeness, incl. error conditions.
7. Check consistency with other requirements.

Requirements analysis

21 Inroduction

22 Expressing Requirements

23 Rapid Prototyping

24 Detailed Requirements

25 Desired Properties of D-requirements

26 Organizing D-requirements

27 Metrics for Requirements

Organizing D-requirements

Alternatives: organize by (combination of)

- Feature (externally visible service).
- System mode/state.
- Use case.
- Class (if available).

A **requirements tool** may help with organizing (providing views) and tracing, incl. links with design etc.

Requirements analysis

21 Inroduction

22 Expressing Requirements

23 Rapid Prototyping

24 Detailed Requirements

25 Desired Properties of D-requirements

26 Organizing D-requirements

27 Metrics for Requirements

Metrics for requirements

- % of defective requirements (that are not testable, traceable, correctly prioritized, atomic, consistent).
- % of missing or defective requirements found per hour of inspection.
- Defect rates (later).
- Cost per requirement.
- See p. 213.

Inspection of requirements

Checklist: is the requirement backward traceable, complete, consistent, feasible, non-ambiguous, clear, precise, modifiable, testable, forward traceable.
Can be put in a form with notes for “no” answers.

Tracking requirements

RID	Priority	Responsible	Inspection	Status	Test
1.2	E	DV	OK	50%	-
...	...	...	...	...	...

SPMP after D-requirements

- More risks, some risks retired.
- More detailed cost estimate.
- More detailed schedule, milestones.
- Designate architects.

Part V

Design

Design

28 Design Steps

29 UML

30 The Domain Model

31 Architectural Design

32 Design Patterns

33 Detailed Design

Design

28 Design Steps

29 UML

30 The Domain Model

31 Architectural Design

32 Design Patterns

33 Detailed Design

Design steps

1. Build domain model.
2. Select architecture.
3. Detailed design.
4. Inspect and update project.

Design

28 Design Steps

29 UML

30 The Domain Model

31 Architectural Design

32 Design Patterns

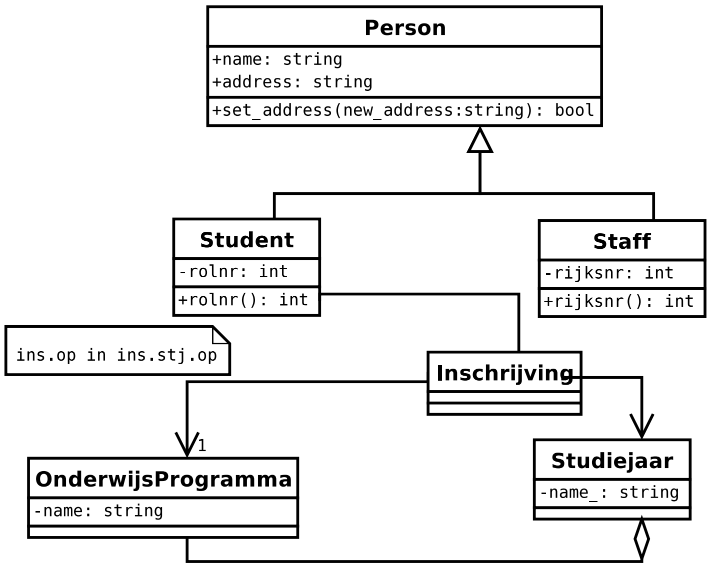
33 Detailed Design

Unified Modeling Language

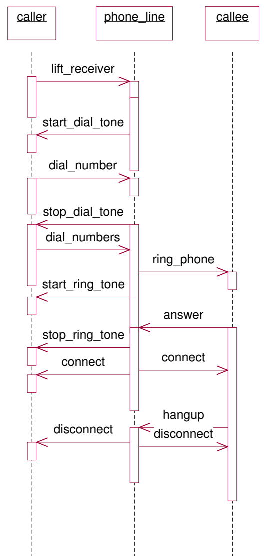
An informal notation for OO designs.

- class model: class diagram.
- dynamic model: state transition diagram.
- sequence diagram.
- collaboration diagram.

Class Model and Diagram



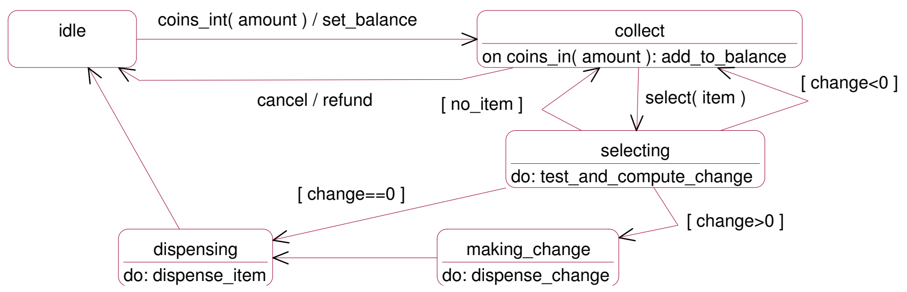
Dynamic Model: Sequence Diagram



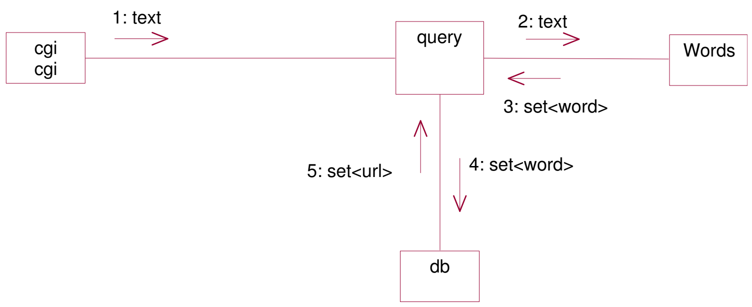
Often elaboration of use case.

Dynamic Model: Transition Diagram

For those classes where useful.



Dynamic Model: Collaboration Diagram



Design

28 Design Steps

29 UML

30 The Domain Model

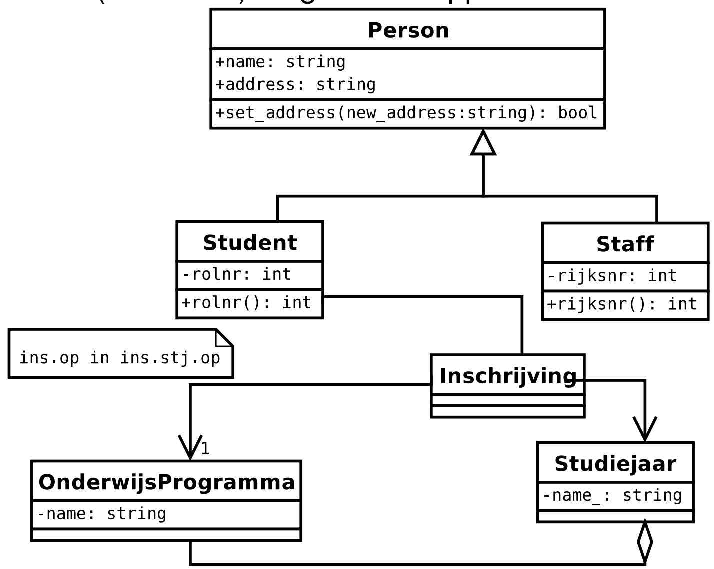
31 Architectural Design

32 Design Patterns

33 Detailed Design

The Domain Model

Class (and other) diagrams of application **domain**.



Finding Domain Classes

- Convert use cases to sequence diagrams

⇒ classes used in these diagrams

- Nouns from requirements.
- Domain knowledge.
- Requirements.
- ...

Building Domain Model

- Determine, for each class,
  - attributes,
  - relationships,
  - operations.
- Use inheritance to represent “is-a” relationship.
- Make state diagram for class if appropriate.

Domain Model Inspection

Verify w.r.t. requirements:

- All concepts represented?
- Use cases supported?
- Dynamic models correct?

Design

28 Design Steps

29 UML

30 The Domain Model

31 Architectural Design

32 Design Patterns

33 Detailed Design

Architectural Design

> *"The initial design process of identifying subsystems and establishing a framework for subsystem control and communication."*

- Architecture = the highest level design.
- Compare with bridge design: decide whether to build a suspension bridge, a cantilever bridge, a cable-stayed bridge, ...

Architectural Quality

- Extensible (adding features).
- Flexible (facilitate changing requirements).
- Simple (easy to understand).
- Reusable (more abstraction ⇒ more reusable).
- Efficient.

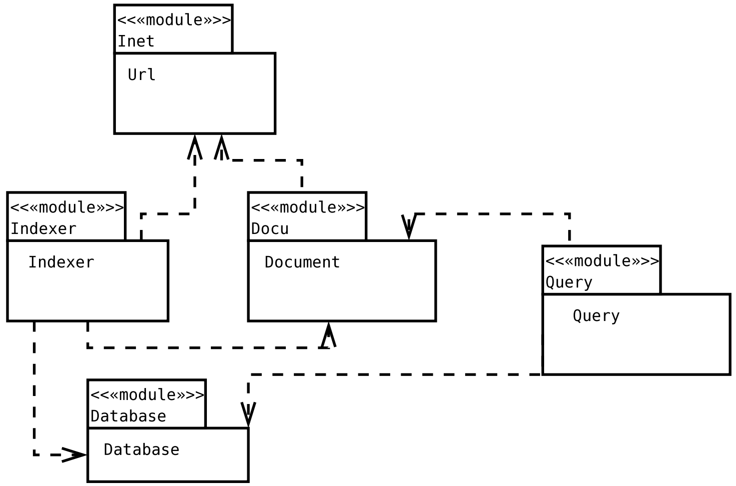
Architectural Design Activities

System structuring: determine principal subsystems and their communications.

Control modeling: establish a general model of the control relationships between the subsystems.

There are many architectural style models.

Example Architecture



Categorization of Architectures

(Shaw and Garlan).

- Data flow architectures (batch sequential, pipes and filters)
- Independent components (parallel communicating processes, client-server, event-driven)
- Virtual machines (interpreters)
- Repository architectures (database, blackboard)

Many real architectures are mix (e.g. compiler: pipe and database)

Comparing Architecture Alternatives

- Give each alternative a score (e.g. “low”, “medium”, “high”) for each quality attribute considered, e.g.
  - Extensibility (easy to add functionality).
  - Flexibility (facilitate changing requirements).
  - Simplicity (easy to understand, cohesion/coupling).
  - Reusable (more abstraction ⇒ more reusable).
  - Efficiency (time, space).
- Give a weight to each quality attribute.
- Compare total weighed scores.

See book p. 287 for example.

Architecture Inspection

Against requirements.

- Are use cases supported by components/control model?
- Can domain model be mapped to components?
- Are all components necessary?

Updating the project

SDD Have chapter/section on architecture alternatives and selection.

SPMP More detailed schedule for developing and testing modules, using dependencies between modules.

Design

28 Design Steps

29 UML

30 The Domain Model

31 Architectural Design

32 Design Patterns

33 Detailed Design

Design Patterns

See book

E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

Design

28 Design Steps

29 UML

30 The Domain Model

31 Architectural Design

32 Design Patterns

33 Detailed Design

Detailed Design

- Add support classes, member functions:
  - Requirements.
  - Data storage.
  - Control.
  - Architecture.
- Determine algorithms.
- Add invariant description to each class, where appropriate.
- Add pre/postconditions to each non-trivial member function.

Detailed Design Notation

- UML diagrams
- C++ header files + documentation generated by doxygen.
- example
- ...

Detailed Design Inspection

- Record metrics: time taken, number and severity of defects found.
- Ensure each architectural module is expanded.
- Ensure each detail (function, class) is part of a module; perhaps revise architecture.
- Ensure design completeness: e.g. covers all requirements, use cases (walk through scenario's, ensure data & functions are available for caller).
- Ensure that design is testable (e.g. special member functions for testing, printing).
- Check detailed design for simplicity, generality and reusability, expandability, efficiency, portability.
- Ensure details (invariants, pre/post conditions) are provided.

Updating Cost Estimates

- Update KLOC estimate, then reapply model.
- Use complete list of member functions (e.g. generated by doxygen under “component members”).
- Estimate the size of each function, e.g. using Humphrey's table (book p. 337).
- Use sum.

Updating Project

SDD Update to reflect design after inspection. E.g. add documentation generated by doxygen from header files.

SPMP

- Complete detail of schedule.
- Assign tasks to members.
- Improve cost and time estimates, based on detailed KLOC estimates.
- Report metrics for design: e.g. time taken for preparation, inspection and change, defects (and severity thereof) found.

SCMP Reflect new parts (e.g. subdirs, source files per module).

Part VI

Implementation

Implementation

34 Preparation and Execution

35 Hints

36 Quality

37 Personal Software Documentation

38 Updating The Project

Implementation

34 Preparation and Execution

35 Hints

36 Quality

37 Personal Software Documentation

38 Updating The Project

How to Prepare for Implementation

- Detailed design confirmed (code only from design).
- Set up process metrics.
- Prepare defect form.
- Prepare standards
  - coding
  - personal documentation

How to Implement Code

A **unit** is the smallest part of the implementation that is separately maintained (and tested): typically class or (member) function. For each unit:

- Plan structure and residual design. Fill in pre- and postconditions.
- Self-inspect residual design.
- Write code and unit test program/functions.
- Inspect.
- Compile & link.
- Apply unit tests (autotools: *make check*).

and collect process metrics and update SQAP,SCMP

Process Metrics

Time spent
- residual detailed design (extra members..)
- coding
- self-inspection
- unit testing
- review
- repair

Defects
- Severity: major (requirements unsatisfied), trivial, other.
- Type (see quality).
- Source: requirements, design, implementation.

Implementation

34 Preparation and Execution
35 Hints
36 Quality
37 Personal Software Documentation
38 Updating The Project

Implementation Hints (1)

Try reuse first. E.g. use STL instead of own container implementation.

Enforce intentions. Prevent unintended use (better: use that can lead to invariant violation).
- Strongly typed parameters: e.g. use **const**, reference parameter i/o pointer if null is not allowed.
- Define things as locally as possible.
- Use patterns such as **singleton** if appropriate.
- Make members as **private** as possible.
- **Include example usage in documentation** (E.g. **example** in doxygen)

Always initialize data members, variables.

Implementation Hints (2)

Encapsulate memory management. Inside class; consider reference-counted pointers template (**shared_ptr**).

Prefer references over pointers, if appropriate.

No operator overloading unless meaning is clear.

Check preconditions. E.g. introduce special type: one place to check.

```
template<int min, int max>
class BoundedInt {
public:
  BoundedInt(int i) throw (range_error);
  operator int() { return value_; }
private:
  int value_;
}
```


Error handling

Inspect. Prevention is better.

Stick to requirements. Refrain from ad-hoc unspecified continuation when faced with a run time error.

Use exceptions and catch them where possible.

E. Raymond's hints

From "*The Art of Unix Programming*".

- **Modularity**: Write Simple Parts connected by clean interfaces.
- **Clarity**: Clarity is better than cleverness.
- **Separation**: Separate policy from mechanism; separate interfaces from engines.
- **Simplicity**: Design for simplicity; add complexity only where you must
- **Parsimony**: Write a big program only when it is clear by demonstration that nothing else will do.
- **Transparency**: Design for visibility to make inspection and debugging easier.
- **Robustness**: Robustness is the child of transparency and simplicity.
- **Representation**: Fold Knowledge into data, so program logic can be stupid and robust.
- ...

E. Raymond's hints 2/2

- **Least Surprise**: In interface design, always do the least surprising thing.
- **Silence**: When a program has nothing surprising to say, it should say nothing.
- **Repair**: Repair what you can–but when you must fail, fail noisily and as soon as possible.
- **Economy**: Programmer time is expensive; conserve it in preference to machine time.
- **Generation**: Avoid hand-hacking; write programs to write programs when you can.
- **Optimization**: Prototype before polishing. Get it working before you optimize it.
- **Diversity**: Distrust all claims for one true way.
- **Extensibility**: Design for the future, because it will be here sooner than you think

Implementation

34 Preparation and Execution
35 Hints
36 Quality
37 Personal Software Documentation
38 Updating The Project

Coding Standards

Rules about

- Naming.
- Comments.
- Indentation.
- Unit tests.
- ...

Implementation Inspection Checklist (1)

Classes Overall

- C1 Appropriate name? consistent with requirements, design? sufficiently general/specialized?
- C2 Could it be an abstract class?
- C3 Header comment describing purpose?
- C4 Header references requirements or design element(s)?
- C5 As private as can be? (e.g. nested)
- C6 Operators allowed? (gang of three)
- C7 Documentation standards applied?

Implementation Inspection Checklist (2)

Class Data Members

- A1 Necessary?
- A2 Could it be **static**?
- A3 Could it be **const**?
- A4 Naming conventions applied?
- A5 As private as possible?
- A6 Attributes are orthogonal?
- A7 Initialized?

Implementation Inspection Checklist (3)

Class Constructors

- O1 Necessary?
- O2 Would a factory method be better?
- O3 Maximal use of initialization list?
- O4 Private as possible?
- O5 Complete? (all data members)

Implementation Inspection Checklist (4)

Function Declarations

- F1 Appropriate name? consistent with requirements, design? sufficiently general/specialized?
- F2 As private as possible?
- F3 Should it be **static**?
- F4 Maximal use of **const**?
- F5 Purpose described?
- F6 Header references requirements or design element(s)?
- F7 Pre- postconditions, invariants stated?
- F8 Documentation standards?
- F9 Parameter types as tight as possible for correct functioning?

Implementation Inspection Checklist (5)

Function Bodies

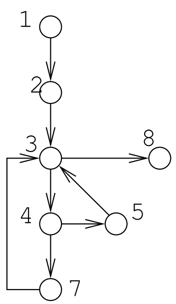
- B1 Algorithm consistent with SDD?
- B2 Code assumes no more than preconditions?
- B3 Code realizes all postconditions?
- B4 Code maintains invariant?
- B5 Each loop terminates?
- B6 Coding standards observed?
- B7 Each line of code necessary & has a clear purpose?
- B8 Check for illegal parameter values?
- B9 Appropriate comments that fit code?

Source Code Metrics

KLOC Need standard for counting comments, white space. Detail is not important but keep constant for comparison.

Cyclomatic Complexity . Based on number of loops in block of code: $C = E - N + 1$ where N, E are numbers of nodes and edges in graph. In example: $C = 2$. High complexity code needs more thorough inspection.

```
int x(x1;
int y(y1);
while (x!=y)
  if (x>y)
    x = x-y;
  else
    y = y-x;
cout << x;
```



Defect Types (1)

Logic. Forgotten case, extreme condition neglected, unnecessary functions, misinterpretation, missing test, wrong check, incorrect iteration, ...

Computational. Loss of precision, wrong equation, ...

Interface. Misunderstanding.

Data handling. Incorrect initialization, incorrect access or assignment, incorrect scaling or dimension, ...

Data. Embedded or external data incorrect or missing, output data incorrect or missing, input data incorrect or missing, ...

Defect Types (2)

Documentation. Mismatch with code, incorrect, missing, ...

Document quality. Standards not followed.

Failure caused by previous fix.

Interoperability. with other software component.

Standards conformance error.

Other . . . .

Implementation

34 Preparation and Execution

35 Hints

36 Quality

37 Personal Software Documentation

38 Updating The Project

Personal Software Documentation

Source code.

Defect log.

- Defect type
- Personal phase (residual design, personal inspection, personal unit test) during which injected/removed.

Time log: time spent on residual design, coding, testing.

Engineering notebook. Status, notes, ...

Bring to exam!

Implementation

34 Preparation and Execution

35 Hints

36 Quality

37 Personal Software Documentation

38 Updating The Project

Updating Project

SQAP
- Coding standards.
- Process metrics data; e.g. from inspections, personal software documentation.

SCMP Location of implementation CI's.

Part VII

Integration and Testing

Integration and Testing

39 Introduction

40 Unit Testing

41 Integration and System Testing

Integration and Testing

39 Introduction

40 Unit Testing

41 Integration and System Testing

Testing

Goal of testing: maximize number and severity of errors found with given budget.

Limit of testing:
- testing can only determine the presence of defects, not their absence.
- Inspections are more (HP: $\times 10$) efficient than testing.

Hierarchy of tests:

Unit tests: of function (members), classes, modules.
Integration tests: of use cases (combination of modules).
System tests: of system.

Integration and Testing

39 Introduction

40 Unit Testing

41 Integration and System Testing

Unit Testing Road Map

1. Based on requirements (& associated code) and detailed design (extra classes): determine which items will be tested in what order $\Rightarrow$ Unit Test Plan.
2. Get input and output data for each test. These may come from previous iterations $\Rightarrow$ Test Set.
3. Execute tests.

Unit Test Types

Black Box: based on requirements/specifications only, without considering design.

White Box: based on detailed design; attempts code coverage and looks at weak spots in design.

Black Box Testing

The space of test data can be divided into classes of data that should be processed in an equivalent way: select test cases from each of the classes.
Example: search value in an array
Input classes:

Array	Element
single value	present
single value	not present
$>$ 1 value	first in array
$>$ 1 value	last in array
$>$ 1 value	middle in array
$>$ 1 value	not in array

White Box Testing

- Use knowledge of code to derive test data (e.g. further classes): path testing ensures that test cases cover each branch in the flow graph.
- Insert assertions to verify (at run time) predicates that should hold at that point. (E.g. **assert** macro in C, C++).

Planning Unit Tests

.

1. Policy: Responsibility of author? By project team or external QA team? Reviewed by?
2. Documentation (see next slide): Incorporate in STD? How to incorporate in other types of testing? Tools?
3. Determine extent of tests. Prioritize tests: tests that are likely to uncover errors first.
4. Decide how and where to get test input.
5. Estimate required resources (e.g. based on historic data).
6. Arrange to track metrics: time, defect count & type & source.

Unit Test Documentation

Typical:

- Test procedures (source code and scripts):
  - An example using program `test-class.C` for each class and a "check" target in the `Makefile`.
  - **Autotools** automatically generates a check target based on a Make variable `check_PROGRAMS`: An example.
- Test (input and output) data.

(Member) Function Unit Tests

- Verify with normal parameters (black box).
- Verify with limit parameters (black box).
- Verify with illegal parameters (black box).
- Ensure code coverage (white box).
- Check termination of all loops (white box) – can also be done using formal proof.
- Check termination of all recursive calls (white box) – can also be done using formal proof.
- Check the handling of error conditions.

See book pp. 408–412.

Class Unit Test

- Exercise member functions in combination:
  - Use most common sequences first.
  - Include sequences likely to cause defects.
  - Verify with expected result.
- Focus unit tests on usage of each data member.
- Verify class invariant is not changed (assert).
- Verify state diagram is followed.

See book pp. 415–417.

Integration and Testing

39 Introduction

40 Unit Testing

41 Integration and System Testing

Integration and System Testing

Integration: Building a (partial) system out of the different modules. Integration proceeds by iterations.

Builds: A build is a partial system made during integration. An iteration may involve several builds.

Associated tests:

- Interface tests.
- Regression tests.
- Integration tests.
- System tests.
- Usability tests.
- Acceptance test.

Planning Integration

1. Identify parts of architecture that will be integrated in each iteration:
   - Try to build bottom-up (no stubs for lower levels).
   - Document requirements and use cases supported by iteration.
   - Retire risks as early as possible.
2. Plan inspection, testing and review process.
3. Make schedule.

Testing during integration

Retest functions, modules in the context of the system (e.g. using no or higher level stubs).

Interface testing of integration.

Regression tests ensures that we did not break anything that worked in the previous build.

Integration tests exercise the combination of modules, verifying the architecture (and the requirements).

System tests test the whole system against the architecture and the requirements.

Usability testing validates the acceptability for the end user.

Acceptance testing is done by the customer to validate the acceptability of the product.

Integration Test Road Map

1. Plan integration.
2. For each iteration:
   1. For each build:
      1. Perform regression tests from previous build.
      2. Retest functions, classes, modules.
      3. Test interfaces.
      4. Perform integration tests.
   2. Perform iteration system and usability tests.
3. Perform installation test.
4. Perform acceptance test.

Integration Testing

1. Decide how and where to store, reuse, code the integration tests (show in project schedule).
2. Execute unit tests in context of the build.
3. Execute regression tests.
4. Ensure build requirements and (partial) use cases are known.
5. Test against these requirements and use cases.
6. Execute system tests supported by this build.

Interface Testing

When testing integrated components or modules: look for errors that misuse, or misunderstand the interface of a component:

- Passing parameters that do not conform to the interface specification, e.g. unsorted array where sorted array expected.
- Misunderstanding of error behavior, e.g. no check on overflow or misinterpretation of return value.

System Testing

A test (script) for each requirement/use case. In addition, do tests for:

- High volume of data.
- Performance.
- Compatibility.
- Reliability and availability (uptime).
- Security.
- Resource usage.
- Installability.
- Recoverability.
- Load/Stress resistance.

Usability Testing

- Against requirements.
- Typically measured by having a sample of users giving a score to various usability criteria.
- Usability criteria should have been specified in advance in the SRS.

The Integration and Testing Process

SCMP Specify iterations and builds (example on p. 466–468)

STD (example on p. 470 – 478, yours can be simpler) Mainly description of tests associated with iterations, builds, system.

- Requirements to be tested.
- Responsible.
- Resources and schedule.
- CI's that will be produced: e.g. for each test:
  - Test script/program.
  - Test data.
  - Test log.
  - Test incidence report.