Einführung in Software Engineering

Prof. O. Nierstrasz

Wintersemester 2002/2003
# Table of Contents

## 1. ESE — Einführung in Software Engineering

- Principle Texts
- Recommended Literature
- Schedule
- Why Software Engineering?
- What is Software Engineering? (I)
- What is Software Engineering? (II)
- What is Software Engineering? (III)
- Software Development Activities
- The Classical Software Lifecycle
- Problems with the Software Lifecycle
- Iterative Development
- Iterative and Incremental Development
- The Unified Process
- Boehm’s Spiral Lifecycle
- Requirements Collection
- Changing requirements
- Requirements Analysis and Specification
- Object-Oriented Analysis
- Prototyping (I)
- Prototyping (II)
- Design
- Conway’s Law
- Implementation and Testing
- Design, Implementation and Testing
- Maintenance
- Maintenance activities
- Maintenance costs
- Methods and Methodologies
- Object-Oriented Methods: a brief history
- What you should know!

## 2. Requirements Collection

- Can you answer these questions?
  - The Requirements Engineering Process
  - Requirements Engineering Activities
  - Requirements Analysis
  - Problems of Requirements Analysis
  - Impedance Mismatches
  - Requirements evolution
  - The Requirements Analysis Process
  - Use Cases and Viewpoints
  - Use Cases and Viewpoints ...
  - Unified Modeling Language
  - Writing Requirements Definitions
  - Functional and Non-functional Requirements
  - Non-functional Requirements
  - Types of Non-functional Requirements
  - Examples of Non-functional Requirements
  - Requirements Verifiability
  - Precise Requirements Measures
  - Prototyping Objectives
  - Evolutionary Prototyping
  - Throw-away Prototyping
  - Requirements Checking
  - Requirements Reviews
  - Review checks
  - Traceability
  - Traceability ...
  - What you should know!
  - Can you answer the following questions?

## 3. The Planning Game

- Extreme Programming
Parameterized Classes 136
Interfaces 137
Utilities 138
Objects 139
Associations 140
Aggregation and Navigability 141
Association Classes 142
Qualified Associations 143
Inheritance 144
What is Inheritance For? 145
Inheritance supports ... 146
Design Patterns as Collaborations 147
Instantiating Design Patterns 148
Constraints 149
Specifying Constraints 150
Design by Contract in UML 151
Using the Notation 152
Using the Notation ... 153
What you should know! 154
Can you answer the following questions? 155

6. Modeling Behaviour 156
Use Case Diagrams 157
Scenarios 158
Sequence Diagrams 159
UML Message Flow Notation 160
Collaboration Diagrams 161
Message Labels 162
Message Labels ... 163
State Diagrams 164
State Diagram Notation 165
State Diagram Notation ... 166
State Box with Regions 167
Transitions 168
Operations and Activities 169

Composite States 170
Sending Events between Objects 171
Concurrent Substates 172
Branching and Merging 173
Branching and Merging ... 174
History Indicator 175
Creating and Destroying Objects 176
Using the Notations 177
What you should know! 178
Can you answer the following questions? 179

7. User Interface Design 180
Interface Design Models 181
User Interface Design Principles 182
GUI Characteristics 184
GUI advantages 186
GUI (dis) advantages ... 187
Direct Manipulation 188
Direct Manipulation ... 189
Interface Models 190
Menu Systems 191
Menu Systems ... 192
Menu Structuring 193
Command Interfaces 194
Command Interfaces ... 195
Information Presentation Factors 196
Analogue vs. Digital Presentation 197
Colour Use Guidelines 198
User Guidance 199
Design Factors in Message Wording 200
Error Message Guidelines 202
Good and Bad Error Messages 203
Help System Design 204
Help system use 205
User Interface Evaluation 206
Usability attributes 207
What you should know! 208
Can you answer the following questions?

8. Software Validation 210
Software Reliability, Failures and Faults
Kinds of failures 211
Programming for Reliability 212
Fault Avoidance 213
Common Sources of Software Faults 214
Common Sources of Software Faults ...
Fault Tolerance 215
Approaches to Fault Tolerance ...
Approaches to Fault Tolerance ...
Defensive Programming 216
Defensive Programming ...
Verification and Validation 217
Verification and Validation ...
The Testing Process 218
The Testing Process ...
Regression testing 219
Test Planning 220
Top-down Testing 221
Bottom-up Testing 222
Defect Testing 223
Defect Testing ...
Functional (black box) testing 224
Coverage Criteria 225
Equivalence partitioning 226
Test Cases and Test Data 227
Structural (white box) Testing 228
Coverage criteria 229
Binary Search Method 230
Path Testing 231
Basis Path Testing: The Technique 232

9. Project Management 252
Recommended Reading 253
Why Project Management? 254
What is Project Management? 255
Risk Management 256
Risk Management ...
Risk Management Techniques 257
Focus on Scope 258
Myth: Scope and Objectives 259
Scope and Objectives 260
Estimation Strategies 261
Estimation Techniques 262
Measurement-based Estimation 263
Estimation and Commitment 264
Planning and Scheduling 265
Planning and Scheduling ...
Myth: Deliverables and Milestones 266
Deliverables and Milestones 267
Example: Task Durations and Dependencies 268
Pert Chart: Activity Network 269
Gantt Chart: Activity Timeline 270
Gantt Chart: Staff Allocation 271
Myth: Delays 272
Scheduling problems 273

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>277</td>
<td>Planning under uncertainty</td>
</tr>
<tr>
<td>278</td>
<td>Dealing with Delays</td>
</tr>
<tr>
<td>279</td>
<td>Dealing with Delays...</td>
</tr>
<tr>
<td>280</td>
<td>Gantt Chart: Slip Line</td>
</tr>
<tr>
<td>281</td>
<td>Timeline Chart</td>
</tr>
<tr>
<td>282</td>
<td>Slip Line vs. Timeline</td>
</tr>
<tr>
<td>283</td>
<td>Software Teams</td>
</tr>
<tr>
<td>284</td>
<td>Chief Programmer Teams</td>
</tr>
<tr>
<td>285</td>
<td>Chief Programmer Teams...</td>
</tr>
<tr>
<td>286</td>
<td>Directing Teams</td>
</tr>
<tr>
<td>287</td>
<td>Directing Teams...</td>
</tr>
<tr>
<td>288</td>
<td>What you should know!</td>
</tr>
<tr>
<td>289</td>
<td>Can you answer these questions?</td>
</tr>
<tr>
<td>290</td>
<td>10. Software Architecture</td>
</tr>
<tr>
<td>291</td>
<td>Sources:</td>
</tr>
<tr>
<td>292</td>
<td>What is Software Architecture?</td>
</tr>
<tr>
<td>293</td>
<td>What is Software Architecture?</td>
</tr>
<tr>
<td>294</td>
<td>How Architecture Drives Implementation</td>
</tr>
<tr>
<td>295</td>
<td>How Architecture Drives Implementation...</td>
</tr>
<tr>
<td>296</td>
<td>Sub-systems, Modules and Components</td>
</tr>
<tr>
<td>297</td>
<td>Cohesion</td>
</tr>
<tr>
<td>298</td>
<td>Coupling</td>
</tr>
<tr>
<td>299</td>
<td>Tight Coupling</td>
</tr>
<tr>
<td>300</td>
<td>Loose Coupling</td>
</tr>
<tr>
<td>301</td>
<td>Architectural Parallels</td>
</tr>
<tr>
<td>302</td>
<td>Architectural Styles</td>
</tr>
<tr>
<td>303</td>
<td>Layered Architectures</td>
</tr>
<tr>
<td>304</td>
<td>Abstract Machine Model</td>
</tr>
<tr>
<td>305</td>
<td>OSI Reference Model</td>
</tr>
<tr>
<td>306</td>
<td>Client-Server Architectures</td>
</tr>
<tr>
<td>307</td>
<td>Client-Server Architectures...</td>
</tr>
<tr>
<td>308</td>
<td>Client-Server Architectures...</td>
</tr>
<tr>
<td>309</td>
<td>Four-Tier Architectures</td>
</tr>
<tr>
<td>310</td>
<td>Blackboard Architectures...</td>
</tr>
<tr>
<td>311</td>
<td>11. Software Quality</td>
</tr>
<tr>
<td>312</td>
<td>What is Quality?</td>
</tr>
<tr>
<td>313</td>
<td>Problems with Software Quality</td>
</tr>
<tr>
<td>314</td>
<td>Hierarchical Quality Model</td>
</tr>
<tr>
<td>315</td>
<td>Quality Attributes</td>
</tr>
<tr>
<td>316</td>
<td>Quality Attributes...</td>
</tr>
<tr>
<td>317</td>
<td>Correctness, Reliability, Robustness</td>
</tr>
<tr>
<td>318</td>
<td>Correctness, Reliability, Robustness...</td>
</tr>
<tr>
<td>319</td>
<td>Efficiency, Usability</td>
</tr>
<tr>
<td>320</td>
<td>Efficiency, Usability...</td>
</tr>
<tr>
<td>321</td>
<td>Maintainability</td>
</tr>
<tr>
<td>322</td>
<td>Maintainability...</td>
</tr>
<tr>
<td>323</td>
<td>Verifiability, Understandability</td>
</tr>
<tr>
<td>324</td>
<td>Productivity, Timeliness, Visibility</td>
</tr>
<tr>
<td>325</td>
<td>Productivity, Timeliness, Visibility...</td>
</tr>
<tr>
<td>326</td>
<td>Quality Control Assumption</td>
</tr>
<tr>
<td>327</td>
<td>The Quality Plan</td>
</tr>
<tr>
<td>328</td>
<td>The Quality Plan...</td>
</tr>
<tr>
<td>329</td>
<td>Types of Quality Reviews</td>
</tr>
</tbody>
</table>
Review Meetings 346
Review Minutes 347
Review Guidelines 348
Sample Review Checklists (I) 349
Sample Review Checklists (II) 350
Sample Review Checklists (III) 351
Sample Review Checklists (IV) 352
Sample Review Checklists (V) 353
Review Results 354
Product and Process Standards 355
Potential Problems with Standards 356
Sample Java Code Conventions 357
Sample Java Code Conventions ... 358
Quality System 359
ISO 9000 360
Capability Maturity Model (CMM) 361
What you should know! 362
Can you answer the following questions? 363

12. Software Metrics 364
Why Metrics? 365
Why Measure Software? 366
What are Software Metrics? 367
(Measures vs Metrics) 368
Direct and Indirect Measures 369
Measurement Mapping 370
Preciseness 371
Possible Problems 372
GQM 373
Cost estimation objectives 374
Estimation techniques 375
Algorithmic cost modelling 376
Measurement-based estimation 377
Lines of code 378
Function points 379

Function points 380
Programmer productivity 381
The COCOMO model 383
Basic COCOMO Formula 384
COCOMO Project classes 385
COCOMO assumptions and problems 386
COCOMO assumptions and problems ... 387
Quantitative Quality Model 388
"Define your own" Quality Model 389
Sample Size (and Inheritance) Metrics 390
Sample Coupling & Cohesion Metrics 391
Coupling & Cohesion Metrics 392
Sample Quality Metrics (I) 393
Sample Quality Metrics (II) 394
Sample Quality Metrics (III) 395
Sample Quality Metrics (IV) 396
What you should know! 397
Can you answer the following questions? 398

13. TBA ... 399
# 1. ESE — Einführung in Software Engineering

| Lecturer      | Prof. Oscar Nierstrasz  
|               | Oscar.Nierstrasz@iam.unibe.ch  
|               | Schützenmattstr. 14/103, Tel.631.4618 |
| Assistants    | Michele Lanza, Tel. 631.4868  
|               | Thomas Buehler, Markus Kobel,  
|               | Michael Locher, Mauricio Seeberger |
| Lectures      | ExWi B7, Wednesdays @ 14h15-16h00 |
| WWW           | www.iam.unibe.ch/~scg/Teaching/ESE/ |

Selected material courtesy of Prof. Serge Demeyer
Principle Texts


Recommended Literature


# Schedule

1. **23-Oct**  Introduction — The Software Lifecycle  
2. **30-Oct**  Requirements Collection  
3. **6-Nov**  The Planning Game  
4. **13-Nov**  Responsibility-Driven Design  
5. **20-Nov**  Modeling Objects and Classes  
6. **27-Nov**  Modeling Behaviour  
7. **4-Dec**  User Interface Design  
8. **11-Dec**  Software Validation  
9. **18-Dec**  Project Management  
10. **8-Jan**  Software Architecture  
11. **15-Jan**  Software Quality  
12. **22-Jan**  Software Metrics  
13. **29-Jan**  TBA ...  
14. **5-Feb**  Final Exam  

© O. Nierstrasz — U. Berne
Why Software Engineering?

A naive view: Problem Specification → coding → Final Program

But ...

- Where did the specification come from?
- How do you know the specification corresponds to the user’s needs?
- How did you decide how to structure your program?
- How do you know the program actually meets the specification?
- How do you know your program will always work correctly?
- What do you do if the users’ needs change?
- How do you divide tasks up if you have more than a one-person team?
What is Software Engineering? (I)

Some Definitions and Issues

“state of the art of developing quality software on time and within budget”

- Trade-off between perfection and physical constraints
  - SE has to deal with real-world issues

- State of the art!
  - Community decides on “best practice” + life-long education
What is Software Engineering? (II)

“multi-person construction of multi-version software”
— Parnas

- Team-work
  - Scale issue (“program well” is not enough) + Communication Issue

- Successful software systems must evolve or perish
  - Change is the norm, not the exception
What is Software Engineering? (III)

“software engineering is different from other engineering disciplines” — Sommerville

- Not constrained by physical laws
  - limit = human mind

- It is constrained by political forces
  - balancing stake-holders
# Software Development Activities

<table>
<thead>
<tr>
<th>Requirements Collection</th>
<th>Establish customer's needs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analysis</strong></td>
<td>Model and specify the requirements (“what”)</td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td>Model and specify a solution (“how”)</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>Construct a solution in software</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td>Validate the solution against the requirements</td>
</tr>
<tr>
<td><strong>Maintenance</strong></td>
<td>Repair defects and adapt the solution to new requirements</td>
</tr>
</tbody>
</table>

NB: these are ongoing activities, not sequential phases!
The Classical Software Lifecycle

The classical software lifecycle models the software development as a step-by-step “waterfall” between the various development phases.

- Requirements Collection
- Analysis
- Design
- Implementation
- Testing
- Maintenance

The waterfall model is unrealistic for many reasons, especially:

- requirements must be “frozen” too early in the life-cycle
- requirements are validated too late
Problems with the Software Lifecycle

1. “Real projects rarely follow the sequential flow that the model proposes. Iteration always occurs and creates problems in the application of the paradigm”

2. “It is often difficult for the customer to state all requirements explicitly. The classic life cycle requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.”

3. “The customer must have patience. A working version of the program(s) will not be available until late in the project timespan. A major blunder, if undetected until the working program is reviewed, can be disastrous.”

— Pressman, SE, p. 26
Iterative Development

In practice, development is always iterative, and all activities progress in parallel.

If the waterfall model is pure fiction, why is it still the standard software process?
Iterative and Incremental Development

Plan to *iterate* your analysis, design and implementation.

You won’t get it right the first time, so *integrate*, *validate* and *test* as frequently as possible.

The later in the lifecycle errors are discovered, the more expensive they are to fix!
Iterative and Incremental Development

Plan to *incrementally* develop (i.e., prototype) the system.

- If possible, *always have a running version* of the system, even if most functionality is yet to be implemented.
- *Integrate* new functionality as soon as possible.
- *Validate* incremental versions against user requirements.
The Unified Process

Inception | Elaboration | Construction | Transition

Requirements

Analysis

Design

Implementation

Test

Iter. #1 Iter. #2 ... ... ... ...

Iter. #n-1 Iter. #n

How do you plan the number of iterations?
How do you decide on completion?
Boehm’s Spiral Lifecycle

**Planning** = determination of objectives, alternatives and constraints

**Risk Analysis** = Analysis of alternatives and identification/resolution of risks

**Risk** = something that will delay project or increase its cost

**Engineering** = Development of the “next level” product

**Customer Evaluation** = Assessment of the results of engineering

**Initial requirements**

**Completion**

**Alpha demo**

**Evolving system**

**Go, no-go decision**

**First prototype**
Requirements Collection

User requirements are often expressed *informally*:
- features
- usage scenarios

Although requirements may be documented in written form, they may be *incomplete, ambiguous, or even incorrect.*
Changing requirements

Requirements will change!

- inadequately captured or expressed in the first place
- user and business needs may change during the project

Validation is needed throughout the software lifecycle, not only when the “final system” is delivered!

- build constant feedback into your project plan
- plan for change
- early prototyping [e.g., UI] can help clarify requirements
**Requirements Analysis and Specification**

*Analysis* is the process of specifying *what* a system will do.

The intention is to provide a clear understanding of what the system is about and what its underlying concepts are.

The result of analysis is a *specification document*.

*Does the requirements specification correspond to the users’ actual needs?*
Object-Oriented Analysis

An object-oriented analysis results in models of the system which describe:

- **classes** of objects that exist in the system
- **responsibilities** of those classes
- **relationships** between those classes
- **use cases** and **scenarios** describing
  - **operations** that can be performed on the system
  - allowable **sequences** of those operations
A *prototype* is a software program developed to test, explore or validate a hypothesis, i.e. *to reduce risks*.

An *exploratory prototype*, also known as a *throwaway prototype*, is intended to *validate requirements* or *explore design choices*.

- UI prototype — validate user requirements
- rapid prototype — validate functional requirements
- experimental prototype — validate technical feasibility
Prototyping (II)

An evolutionary prototype is intended to evolve in steps into a finished product.

- iteratively “grow” the application, redesigning and refactoring along the way

✔ First do it, then do it right, then do it fast.
Design

*Design* is the process of specifying how the specified system behaviour will be realized from software components. The results are *architecture* and *detailed design documents*.

**Object-oriented design** delivers models that describe:
- how system operations are implemented by *interacting objects*
- how classes *refer* to one another and how they are related by *inheritance*
- *attributes* and *operations* associated to classes

*Design is an iterative process, proceeding in parallel with implementation!*
Conway’s Law

“Organizations that design systems are constrained to produce designs that are copies of the communication structures of these organizations”
Implementation and Testing

**Implementation** is the activity of *constructing* a software solution to the customer’s requirements.

**Testing** is the process of *validating* that the solution meets the requirements.

The result of implementation and testing is a **fully documented** and **validated** solution.
Design, Implementation and Testing

Design, implementation and testing are iterative activities

- The implementation does not “implement the design”, but rather the design document documents the implementation!

- System tests reflect the requirements specification
- Testing and implementation go hand-in-hand
- Ideally, test case specification precedes design and implementation
Maintenance

Maintenance is the process of changing a system after it has been deployed.

- **Corrective maintenance**: identifying and repairing defects
- **Adaptive maintenance**: adapting the existing solution to new platforms
- **Perfective maintenance**: implementing new requirements

In a spiral lifecycle, everything after the delivery and deployment of the first prototype can be considered “maintenance”!
Maintenance activities

“Maintenance” entails:

- configuration and version management
- reengineering (redesigning and refactoring)
- updating all analysis, design and user documentation

Repeatable, automated tests enable evolution and refactoring
Breakdown of maintenance costs.
Source: Lientz 1979
Methods and Methodologies

**Principle** = general statement describing desirable properties

**Method** = general guidelines governing some activity

**Technique** = more technical and mechanical than method

**Methodology** = package of methods and techniques packaged

---

— Ghezzi et al. 1991
Object-Oriented Methods: a brief history

First generation:
- Adaptation of existing notations (ER diagrams, state diagrams ...): Booch, OMT, Shlaer and Mellor, ...
- Specialized design techniques:
  - CRC cards; responsibility-driven design; design by contract

Second generation:
- Fusion: Booch + OMT + CRC + formal methods

Third generation:
- Unified Modeling Language:
  - uniform notation: Booch + OMT + Use Cases + ...
  - various UML-based methods (e.g. Catalysis)
What you should know!

✎ How does Software Engineering differ from programming?
✎ Why is the “waterfall” model unrealistic?
✎ What is the difference between analysis and design?
✎ Why plan to iterate? Why develop incrementally?
✎ Why is programming only a small part of the cost of a “real” software project?
✎ What are the key advantages and disadvantages of object-oriented methods?
Can you answer these questions?

✎ What is the appeal of the “waterfall” model?
✎ Why do requirements change?
✎ How can you validate that an analysis model captures users’ real needs?
✎ When does analysis stop and design start?
✎ When can implementation start?
✎ What are good examples of Conway’s Law in action?
2. Requirements Collection

Overview:
- The Requirements Engineering Process
- Use cases and scenarios
- Functional and non-functional requirements
- Evolutionary and throw-away prototyping
- Requirements checking and reviews

Sources:
- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999
The Requirements Engineering Process

- Feasibility study
  - Feasibility report
- Requirements analysis
- Requirements definition
- Requirements specification
  - System models
  - Definition of requirements
  - Specifications of requirements
  - Requirements document

© Ian Sommerville 1995

© O. Nierstrasz — U. Berne
# Requirements Engineering Activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feasibility study</td>
<td>Determine if the user needs can be satisfied with the available technology and budget.</td>
</tr>
<tr>
<td>Requirements analysis</td>
<td>Find out what system stakeholders require from the system.</td>
</tr>
<tr>
<td>Requirements definition</td>
<td>Define the requirements in a form understandable to the customer.</td>
</tr>
<tr>
<td>Requirements specification</td>
<td>Define the requirements in detail. (Written as a contract between client and contractor.)</td>
</tr>
</tbody>
</table>

“Requirements are for users; specifications are for analysts and developers.”
Requirements Analysis

Sometimes called *requirements elicitation* or *requirements discovery*

Technical staff work with customers to determine

- the application *domain*,
- the *services* that the system should provide and
- the system’s operational *constraints*.

Involves various *stakeholders*:

- e.g., end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc.
Various problems typically arise:

- Stakeholders *don’t know* what they really want
- Stakeholders *express* requirements in their *own terms*
- Different stakeholders may have *conflicting* requirements
- *Organisational* and *political* factors may influence the system requirements
- The requirements *change* during the analysis process. New stakeholders may emerge.
Impedance Mismatches

1. As Management requested it
2. As the Project Leader defined it
3. As Systems designed it
4. As Programming developed it
5. As Operations installed it
6. What the User wanted
Requirements evolution

- Requirements *always evolve* as a better understanding of user needs is developed and as the organisation’s objectives change

- It is essential to *plan for change* in the requirements as the system is being developed and used
The Requirements Analysis Process

- Process entry
- Domain understanding
  - Requirements collection
  - Classification
- Prioritization
- Requirements validation
- Conflict resolution
- Requirements definition and specification

© Ian Sommerville 1995

© O. Nierstrasz — U. Berne

Requirements Collection
Use Cases and Viewpoints

A use case is the specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system”.

- e.g., buy a DVD through the internet

A scenario is a particular trace of action occurrences, starting from a known initial state.

- e.g., connect to myDVD.com, go to the “search” page ...

Use Cases and Viewpoints ...

**Stakeholders** represent different problem viewpoints.

- Interview as many **different** kinds of stakeholders as possible/necessary
- Translate requirements into **use cases** or “stories” about the desired system involving a fixed set of **actors** (users and system objects)
- For each use case, capture **both typical and exceptional usage scenarios**

**Users** tend to think about systems in terms of “features”.

- You must get them to tell you **stories** involving those features.
- Use cases and scenarios can tell you if the requirements are **complete and consistent!**
# Unified Modeling Language

UML is an industry standard for documenting OO models.

<table>
<thead>
<tr>
<th>Diagram Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class Diagrams</strong></td>
<td>visualize <em>logical structure</em> of system in terms of classes, objects and relationships</td>
</tr>
<tr>
<td><strong>Use Case Diagrams</strong></td>
<td>show external <em>actors</em> and <em>use cases</em> they participate in</td>
</tr>
<tr>
<td><strong>Sequence Diagrams</strong></td>
<td>visualize <em>temporal message ordering</em> of a <em>concrete</em> scenario of a use case</td>
</tr>
<tr>
<td><strong>Collaboration Diagrams</strong></td>
<td>visualize <em>relationships</em> of objects exchanging messages in a <em>concrete scenario</em></td>
</tr>
<tr>
<td><strong>State Diagrams</strong></td>
<td>specify the <em>abstract states</em> of an object and the <em>transitions</em> between the states</td>
</tr>
</tbody>
</table>
Writing Requirements Definitions

Requirements definitions usually consist of natural language, supplemented by (e.g., UML) diagrams and tables.

Three types of problem can arise:

Lack of clarity: It is hard to write documents that are both precise and easy-to-read.

Requirements confusion: Functional and non-functional requirements tend to be intertwined.

Requirements amalgamation: Several different requirements may be expressed together.
**Functional and Non-functional Requirements**

*Functional requirements* describe system *services* or *functions*
- Compute sales tax on a purchase
- Update the database on the server ...

*Non-functional requirements* are *constraints* on the system or the development process

Non-functional requirements may be more critical than functional requirements. 
*If these are not met, the system is useless!*
## Non-functional Requirements

<table>
<thead>
<tr>
<th><strong>Product requirements:</strong></th>
<th>specify that the delivered product <strong>must behave</strong> in a particular way e.g. execution <strong>speed</strong>, <strong>reliability</strong>, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Organisational requirements:</strong></td>
<td>are a consequence of <strong>organisational policies</strong> and procedures e.g. <strong>process standards</strong> used, implementation requirements, etc.</td>
</tr>
<tr>
<td><strong>External requirements:</strong></td>
<td>arise from factors which are external to the system and its development process e.g. <strong>interoperability</strong> requirements, <strong>legislative requirements</strong>, etc.</td>
</tr>
</tbody>
</table>
Types of Non-functional Requirements

- Non-functional requirements
  - Product requirements
    - Efficiency requirements
    - Usability requirements
      - Performance requirements
  - Organizational requirements
    - Reliability requirements
    - Delivery requirements
    - Implementation requirements
  - External requirements
    - Portability requirements
    - Standards requirements
    - Legislatively requirements
    - Interoperability requirements
    - Privacy requirements
      - Safety requirements
    - Ethical requirements
## Examples of Non-functional Requirements

<table>
<thead>
<tr>
<th><strong>Product requirement</strong></th>
<th>It shall be possible for all necessary communication between the APSE and the user to be expressed in the <em>standard Ada character set</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Organisational requirement</strong></td>
<td>The <em>system development process</em> and deliverable documents shall conform to the process and deliverables defined in <em>XYZCo-SP-STAN-95</em>.</td>
</tr>
<tr>
<td><strong>External requirement</strong></td>
<td>The system shall provide facilities that allow any user to check if personal data is maintained on the system. <em>A procedure must be defined and supported</em> in the software that will <em>allow users to inspect personal data</em> and to correct any errors in that data.</td>
</tr>
</tbody>
</table>
Requirements Verifiability

Requirements must be written so that they can be objectively verified.

Imprecise: The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.

Terms like “easy to use” and “errors shall be minimised” are useless as specifications.

Verifiable: Experienced controllers should be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users should not exceed two per day.
## Precise Requirements Measures

<table>
<thead>
<tr>
<th>Property</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>Processed transactions/second  User/Event response time  Screen refresh time</td>
</tr>
<tr>
<td>Size</td>
<td>K Bytes; Number of RAM chips</td>
</tr>
<tr>
<td>Ease of use</td>
<td>Training time  Rate of errors made by trained users  Number of help frames</td>
</tr>
<tr>
<td>Reliability</td>
<td>Mean time to failure  Probability of unavailability  Rate of failure occurrence</td>
</tr>
<tr>
<td>Property</td>
<td>Measure</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>Robustness</td>
<td>Time to restart after failure</td>
</tr>
<tr>
<td></td>
<td>Percentage of events causing failure</td>
</tr>
<tr>
<td></td>
<td>Probability of data corruption on failure</td>
</tr>
<tr>
<td>Portability</td>
<td>Percentage of target dependent statements</td>
</tr>
<tr>
<td></td>
<td>Number of target systems</td>
</tr>
</tbody>
</table>
Prototyping Objectives

The objective of **evolutionary prototyping** is to deliver a **working system** to end-users.

- Development starts with the requirements that are **best understood**.

The objective of **throw-away prototyping** is to **validate or derive the system requirements**.

- Prototyping starts with that requirements that are **poorly understood**.
Evolutionary Prototyping

- Must be used for systems where the specification cannot be developed in advance.
  - e.g., AI systems and user interface systems

- Based on techniques which allow rapid system iterations.
  - e.g., executable specification languages, VHL languages, 4GLs, component toolkits

- Verification is impossible as there is no specification.
  - Validation means demonstrating the adequacy of the system.
Throw-away Prototyping

- Used to reduce requirements risk

- The prototype is developed from an initial specification, delivered for experiment then discarded

- The throw-away prototype should not be considered as a final system
  - Some system characteristics may have been left out (e.g., platform requirements may be ignored)
  - There is no specification for long-term maintenance
  - The system will be poorly structured and difficult to maintain
## Requirements Checking

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Validity:</strong></td>
<td>Does the system provide the functions <em>which best support</em> the customer’s needs?</td>
</tr>
<tr>
<td><strong>Consistency:</strong></td>
<td>Are there any requirements <em>conflicts</em>?</td>
</tr>
<tr>
<td><strong>Completeness:</strong></td>
<td>Are <em>all functions</em> required by the customer included?</td>
</tr>
<tr>
<td><strong>Realism:</strong></td>
<td>Can the requirements be implemented given <em>available budget</em> and <em>technology</em>?</td>
</tr>
</tbody>
</table>
Requirements Reviews

- **Regular reviews** should be held while the requirements definition is being formulated.
- Both *client* and *contractor staff* should be involved in reviews.
- Reviews may be *formal* (with completed documents) or *informal*.

*Good communications* between developers, customers and users can resolve problems at an *early stage*. 
## Review checks

<table>
<thead>
<tr>
<th><strong>Verifiability</strong></th>
<th>Is the requirement realistically <em>testable</em>?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comprehensibility</strong></td>
<td>Is the requirement properly <em>understood</em>?</td>
</tr>
<tr>
<td><strong>Traceability</strong></td>
<td>Is the <em>origin</em> of the requirement clearly stated?</td>
</tr>
<tr>
<td><strong>Adaptability</strong></td>
<td>Can the requirement be <em>changed</em> without a large <em>impact</em> on other requirements?</td>
</tr>
</tbody>
</table>
Traceability

To protect against changes you should be able to trace back from every system component to the original requirement that caused its presence.

<table>
<thead>
<tr>
<th>Comp 1</th>
<th>Comp 2</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>Comp m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 1</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Req 2</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Req n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Traceability ...

- A *software process* should help you keeping this virtual table up-to-date
- *Simple techniques* may be quite valuable (naming conventions, ...)

© O. Nierstrasz — U. Berne

Requirements Collection
What you should know!

✎ What is the difference between requirements analysis and specification?
✎ Why is it hard to define and specify requirements?
✎ What are use cases and scenarios?
✎ What is the difference between functional and non-functional requirements?
✎ What’s wrong with a requirement that says a product should be “user-friendly”?
✎ What’s the difference between evolutionary and throw-away prototyping?
Can you answer the following questions?

✎ Why isn’t it enough to specify requirements as a set of desired features?

✎ Which is better for specifying requirements: natural language or diagrams?

✎ How would you prototype a user interface for a web-based ordering system?

✎ Would it be an evolutionary or throw-away prototype?

✎ What would you expect to gain from the prototype?

✎ How would you check a requirement for “adaptability“?
3. The Planning Game

Overview:

- XP — coping with change and uncertainty
- Customers and Developers — why do we plan?
- User stories
- Estimation

Source:


Based on a presentation by Matthias Rieger.
Extreme Programming

XP is a set of mutually supportive practices for developing quality software

- Planning Game
- Metaphor
- Simple design
- Refactoring
- Continuous integration
- Testing
- Pair programming
- Coding standards
- Collective code ownership
- Short releases
Driving Metaphor

Driving a car is not about pointing the car in one direction and holding to it; driving is about making *lots of little course corrections*.

“Do the simplest thing that could possibly work”
Why we plan

We want to ensure that

- we are always working on the most important things
- we are coordinated with other people
- when unexpected events occur, we understand the consequences on priorities and coordination

Plans must be

- easy to make and update
- understandable by everyone that uses them
The Planning Trap

- Plans project a *likely* course of events
- Plans must try to create *visibility*: where is the project

**But:** A plan does not mean you are in control of things
- Events happen
- Plans become invalid

Having a plan isn't everything, *planning* is.

- Keep plans honest and expect them to always change
Customer-Developer Relationships

A well-known experience in Software Development:
The customer and the developer sit in a small boat in the ocean and are afraid of each other.

<table>
<thead>
<tr>
<th>Customer fears</th>
<th>Developer fears</th>
</tr>
</thead>
<tbody>
<tr>
<td>They won't get what they asked for</td>
<td>They won't be given clear definitions of what needs to be done</td>
</tr>
<tr>
<td>They must surrender the control of their careers to techies who don't care</td>
<td>They will be given responsibility without authority</td>
</tr>
<tr>
<td>They'll pay too much for too little</td>
<td>They will be told to do things that don't make sense</td>
</tr>
<tr>
<td>They won't know what is going on (the plans they see will be fairy tales)</td>
<td>They'll have to sacrifice quality for deadlines</td>
</tr>
</tbody>
</table>

Result: A lot of energy goes into protective measures and politics instead of success
## The Customer Bill of Rights

<table>
<thead>
<tr>
<th>You have the right to an overall plan</th>
<th>To steer a project, you need to know what can be accomplished within time and budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>You have the right to get the most possible value out of every programming week</td>
<td>The most valuable things are worked on first.</td>
</tr>
<tr>
<td>You have the right to see progress in a running system.</td>
<td>Only a running system can give exact information about project state</td>
</tr>
<tr>
<td>You have the right to change your mind, to substitute functionality and to change priorities without exorbitant costs.</td>
<td>Market and business requirements change. We have to allow change.</td>
</tr>
<tr>
<td>You have the right to be informed about schedule changes, in time to choose how to reduce the scope to restore the original date.</td>
<td>XP works to be sure everyone knows just what is really happening.</td>
</tr>
</tbody>
</table>
# The Developer Bill of Rights

<table>
<thead>
<tr>
<th>Right</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>You have the right to know what is needed, with clear declarations of</td>
<td>Tight communication with the customer. Customer directs by value.</td>
</tr>
<tr>
<td>priority.</td>
<td>Unit Tests and Refactoring help to keep the code clean</td>
</tr>
<tr>
<td>You have the right to produce quality work all the time.</td>
<td>No one can ever refuse help to a team member</td>
</tr>
<tr>
<td>You have the right to ask for and receive help from peers, managers,</td>
<td>Programmers know best how long it is going to take them</td>
</tr>
<tr>
<td>and customers</td>
<td></td>
</tr>
<tr>
<td>You have the right to make and update your own estimates.</td>
<td></td>
</tr>
<tr>
<td>You have the right to accept your responsibilities instead having</td>
<td>We work most effectively when we have accepted our responsibilities</td>
</tr>
<tr>
<td>them assigned to you</td>
<td>instead of having them thrust upon us</td>
</tr>
</tbody>
</table>
Separation of Roles

- Customer makes *business* decisions
- Developers make *technical* decisions

<table>
<thead>
<tr>
<th>Business Decisions</th>
<th>Technical Decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Estimates</td>
</tr>
<tr>
<td>Dates of the releases</td>
<td>Dates within an iteration</td>
</tr>
<tr>
<td>Priority</td>
<td>Team velocity</td>
</tr>
<tr>
<td></td>
<td>Warnings about technical risks</td>
</tr>
</tbody>
</table>

The Customer owns “what you get” while the Developers own “what it costs”. 
The Planning Game

A game with a set of rules that ensures that Customer and Developers don’t become mortal enemies

Goal:
Maximize the value of the software produced by Developers.

Overview:
1. Release Planning: Customer selects the scope of the next release

2. Iteration Planning: Developers decide on what to do and in which order
# The Release Planning Game

<table>
<thead>
<tr>
<th></th>
<th>Customer</th>
<th>Developers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exploration Phase</strong></td>
<td>Write Story</td>
<td>Estimate Story</td>
</tr>
<tr>
<td></td>
<td>Split Story</td>
<td></td>
</tr>
<tr>
<td><strong>Commitment Phase</strong></td>
<td>Sort Stories by Value</td>
<td>Sort Stories by Risk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set Velocity</td>
</tr>
<tr>
<td></td>
<td>Choose Scope</td>
<td></td>
</tr>
<tr>
<td><strong>Steering Phase</strong></td>
<td>Iteration</td>
<td>Recovery</td>
</tr>
<tr>
<td></td>
<td>New Story</td>
<td>Reestimate</td>
</tr>
</tbody>
</table>
Planning Game: Exploration Phase

Purpose:
Get an appreciation for what the system should eventually do.

The Moves:
- **Customer**: *Write a story*. Discuss it until everybody understands it.
- **Developers**: *Estimate* a story in terms of effort.
- **Customer**: *Split* a story, if Developers don’t understand or can’t estimate it.
- **Developers**: Do a *spike solution* to enable estimation.
- **Customer**: *Toss* stories that are no longer wanted or are covered by a split story.
User Stories

Principles of good stories:

- **Customers** write stories. Developers do *not* write stories.
- Stories must be *understandable* to the customer.
- The **shorter** the better. No detailed specification!
  - Write stories on index cards
- Each story must provide something of *value* to the customer.
- A story must be *testable*
  - then we can know when it is done

Writing stories is an iterative process, requiring interaction between Customer and Developers.
Stories

A story contains:

- a name
- the story itself
- an estimate

Example:

- When the GPS has contact with two or fewer satellites for more than 60 seconds, it should display the message “Poor satellite contact”, and wait for confirmation from the user. If contact improves before confirmation, clear the message automatically.
Splitting Stories

Developers ask the Customer to split a story if

- They cannot estimate a story because of its complexity
- Their estimate is longer than two or three weeks of effort

Why?

- Estimates get fuzzy for bigger stories
- The smaller the story, the better the control (tight feedback loop)
Initial Estimation of Stories

With no history, the first plan is the hardest and least accurate (fortunately, you only have to do it once)

How to start estimating:

❑ Begin with the stories that you feel the most comfortable estimating.
❑ Intuitively imagine how long it will take you.
❑ Base other estimates on the comparison with those first stories.

Spike Solutions:
Do a quick implementation of the whole story.
  — Do not look for the perfect solution!
  — Just try to find out how long something takes
Estimating Stories

Keys to effective story estimation:
- Keep it simple
- Use what happened in the past ("Yesterday's weather")
- Learn from experience

Comparative story estimation:
- One story is often an elaboration of a closely related one
- Look for stories that have already been implemented
- Compare difficulties, not implementation time
  - “twice as difficult”, “half as difficult”
- Discuss estimates in the team. Try to find an agreement.
- “Optimism wins”: Choose the more optimistic of two disagreeing estimates.
Planning Game: Commitment Phase

Purpose:

Customer: to choose scope and date of next delivery
Developers: to confidently commit to deliver the next release

The Moves:

- Customer: Sort by stories by value
  - (1) Stories without which the system will not function
  - (2) Less essential stories, but still providing significant business value
  - (3) Nice-to-have stories
  - Customer wants the release to be as valuable as possible
- **Developers:** Sort stories by risk
  1. Stories that can be estimated precisely *(low risk)*
  2. Stories that can be estimated reasonably well
  3. Stories that cannot be estimated *(high risk)*

  Developers want to tackle *high-risk first*, or at least make risk visible.

- **Developers:** Set team velocity
  How much ideal engineering time per calendar month/week can the team offer?
  
  this is the budget that is available to Customer.

- **Customer:** Choose *scope* of the release, by either
  - fixing the date and choosing stories based on estimates and velocity
  - fixing the stories and calculating the delivery date
Planning Game: Steering Phase

Purpose: Update the plan based on what is learned.

The Moves:

- **Iteration**: Customer picks one iteration worth of the most valuable stories.
  - see Iteration Planning
- **Get stories done**: Customer should only accept stories that are 100% done.
- **Recovery**: Developers realize velocity is wrong
  - Developers re-estimate velocity.
  - Customer can defer (or split) stories to maintain release date.

...
Planning Game: Steering Phase...

- **New Story**: Customer identifies *new, more valuable stories*
  - Developers estimate story
  - Customer removes estimated points from incomplete part of existing plan, and inserts the new story.

- **Reestimate**: Developers feel that *plan is no longer accurate*
  - Developers re-estimate velocity and all stories.
  - Customer sets new scope plan.
Iteration Planning

Read Story Cards
Write Task Cards
Unclaimed Tasks
Select and Estimate Tasks

Accepted Tasks: Programmer 1, Programmer 2, Programmer 3, Programmer 4

“too big” or “too busy”

Phase 1
Phase 2

Copyright 2000 by William C. Wake
Iteration Planning

- **Customer selects** stories to be implemented in this iteration.

- **Customer explains** the stories in detail to the Developers
  - Resolve ambiguities and unclear parts in discussion

...
Iteration Planning...

- **Developers** brainstorm **engineering tasks**
  - A task is small enough that everybody fully understands it and can estimate it.
  - Use short CRC or UML sessions to determine how a story is accomplished.

- Observing the design process builds common knowledge and confidence throughout the team.

- **Developers/pairs** sign up for work and estimates
  - Assignments are not forced upon anybody (Principle of Accepted Responsibility)
  - The person responsible for a task gets to do the estimate.
What you should know!

- Why is planning more important than having a plan?
- Why shouldn’t Customers make technical decisions? Why shouldn’t Developers make business decisions?
- Why should stories be written on index cards?
- Why should the Customer sort stories by value?
- Why should the Developer sort stories by risk?
- How do you assign stories to Developers?
Can you answer the following questions?

✎ What is “extreme” about XP?
✎ What is the differences between a User Story and a Use Case?
✎ Are Developers allowed to write stories?
✎ What is the ideal time period for one iteration?
✎ How can you improve your skill at estimating stories?
4. Responsibility-Driven Design

Overview:
- Finding Classes
- CRC sessions
- Identifying Responsibilities
- Finding Collaborations
- Structuring Inheritance Hierarchies

Source:
Why Responsibility-driven Design?

Functional Decomposition

- Decompose according to the functions a system is supposed to perform.

Functional Decomposition

- Good in a “waterfall” approach: stable requirements and one monolithic function

However

- Naive: Modern systems perform more than one function
- Maintainability: system functions evolve ⇒ redesign affect whole system
- Interoperability: interfacing with other system is difficult
Why Responsibility-driven Design? …

Object-Oriented Decomposition

Decompose according to the objects a system is supposed to manipulate.

Object-Oriented Decomposition

- Better for complex and evolving systems

However

- How to find the objects?
Iteration in Object-Oriented Design

- The result of the design process is not a final product:
  - design decisions may be revisited, even after implementation
  - design is not linear but iterative

- The design process is not algorithmic:
  - a design method provides guidelines, not fixed rules
  - “a good sense of style often helps produce clean, elegant designs — designs that make a lot of sense from the engineering standpoint”

✔ Responsibility-driven design is an (analysis and) design technique that works well in combination with various methods and notations.
The Initial Exploration

1. Find the *classes* in your system

2. Determine the *responsibilities* of each class

3. Determine how objects *collaborate* with each other to fulfil their responsibilities
The Detailed Analysis

1. **Factor** common responsibilities to build class hierarchies
2. **Streamline** collaborations between objects
   - Is message traffic heavy in parts of the system?
   - Are there classes that collaborate with everybody?
   - Are there classes that collaborate with nobody?
   - Are there groups of classes that can be seen as subsystems?
3. Turn class responsibilities into fully specified signatures
Finding Classes

Start with requirements specification:

❑ What are the goals of the system being designed, its expected inputs and desired responses?

1. Look for noun phrases:
   - separate into obvious classes, uncertain candidates, and nonsense
Finding Classes …

2. Refine to a list of candidate classes. Some guidelines are:
   - Model physical objects — e.g. disks, printers
   - Model conceptual entities — e.g. windows, files
   - Choose one word for one concept — what does it mean within the system
   - Be wary of adjectives — is it really a separate class?
   - Be wary of missing or misleading subjects — rephrase in active voice
   - Model categories of classes — delay modelling of inheritance
   - Model interfaces to the system — e.g., user interface, program interfaces
   - Model attribute values, not attributes — e.g., Point vs. Centre
Drawing Editor Requirements Specification

The drawing editor is an interactive graphics editor. With it, users can create and edit drawings composed of lines, rectangles, ellipses and text.

Tools control the mode of operation of the editor. Exactly one tool is active at any given time.

Two kinds of tools exist: the selection tool and creation tools. When the selection tool is active, existing drawing elements can be selected with the cursor. One or more drawing elements can be selected and manipulated; if several drawing elements are selected, they can be manipulated as if they were a single element. Elements that have been selected in this way are referred to as the current selection. The current selection is indicated visually by displaying the control points for the element. Clicking on and dragging a control point modifies the element with which the control point is associated.

When a creation tool is active, the current selection is empty. The cursor changes in different ways according to the specific creation tool, and the user can create an element of the selected kind. After the element is created, the selection tool is made active and the newly created element becomes the current selection.

The text creation tool changes the shape of the cursor to that of an I-beam. The position of the first character of text is determined by where the user clicks the mouse button. The creation tool is no longer active when the user clicks the mouse button outside the text element. The control points for a text element are the four corners of the region within which the text is formatted. Dragging the control points changes this region. The other creation tools allow the creation of lines, rectangles and ellipses. They change the shape of the cursor to that of a crosshair. The appropriate element starts to be created when the mouse button is pressed, and is completed when the mouse button is released. These two events create the start point and the stop point.

The line creation tool creates a line from the start point to the stop point. These are the control points of a line. Dragging a control point changes the end point.

The rectangle creation tool creates a rectangle such that these points are diagonally opposite corners. These points and the other corners are the control points. Dragging a control point changes the associated corner.

The ellipse creation tool creates an ellipse fitting within the rectangle defined by the two points described above. The major radius is one half the width of the rectangle, and the minor radius is one half the height of the rectangle. The control points are at the corners of the bounding rectangle. Dragging control points changes the associated corner.
**Drawing Editor: noun phrases**

The drawing editor is an interactive graphics editor. With it, users can create and edit drawings composed of lines, rectangles, ellipses and text. Tools control the mode of operation of the editor. Exactly one tool is active at any given time.

Two kinds of tools exist: the selection tool and creation tools. When the selection tool is active, existing drawing elements can be selected with the cursor. One or more drawing elements can be selected and manipulated; if several drawing elements are selected, they can be manipulated as if they were a single element. Elements that have been selected in this way are referred to as the current selection. The current selection is indicated visually by displaying the control points for the element. Clicking on and dragging a control point modifies the element with which the control point is associated.

When a creation tool is active, the current selection is empty. The cursor changes in different ways according to the specific creation tool, and the user can create an element of the selected kind. After the element is created, the selection tool is made active and the newly created element becomes the current selection.

...
The text creation tool changes the shape of the cursor to that of an I-beam. The position of the first character of text is determined by where the user clicks the mouse button. The creation tool is no longer active when the user clicks the mouse button outside the text element. The control points for a text element are the four corners of the region within which the text is formatted. Dragging the control points changes this region. The other creation tools allow the creation of lines, rectangles and ellipses. They change the shape of the cursor to that of a crosshair. The appropriate element starts to be created when the mouse button is pressed, and is completed when the mouse button is released. These two events create the start point and the stop point.

The line creation tool creates a line from the start point to the stop point. These are the control points of a line. Dragging a control point changes the end point.

The rectangle creation tool creates a rectangle such that these points are diagonally opposite corners. These points and the other corners are the control points. Dragging a control point changes the associated corner.

The ellipse creation tool creates an ellipse fitting within the rectangle defined by the two points described above. The major radius is one half the width of the rectangle, and the minor radius is one half the height of the rectangle. The control points are at the corners of the bounding rectangle. Dragging control points changes the associated corner.
Class Selection Rationale

Model physical objects:
- mouse button [event or attribute]

Model conceptual entities:
- ellipse, line, rectangle
- Drawing, Drawing Element
- Tool, Creation Tool, Ellipse Creation Tool, Line Creation Tool, Rectangle Creation Tool, Selection Tool, Text Creation Tool
- text, Character
- Current Selection

...
Class Selection Rationale ...

Choose one word for one concept:

- Drawing Editor ⇒ editor, interactive graphics editor
- Drawing Element ⇒ element
- Text Element ⇒ text
- Ellipse Element, Line Element, Rectangle Element ⇒ ellipse, line, rectangle

...
Class Selection Rationale ...

Be wary of adjectives:

- Ellipse Creation Tool, Line Creation Tool, Rectangle Creation Tool, Selection Tool, Text Creation Tool — all have different requirements
- bounding rectangle, rectangle, region ⇒ Rectangle — common meaning, but different from Rectangle Element
- Point ⇒ end point, start point, stop point
- Control Point — more than just a coordinate
- corner ⇒ associated corner, diagonally opposite corner — no new behaviour

...
Class Selection Rationale ...

Be wary of sentences with missing or misleading subjects:

“The current selection is indicated visually by displaying the control points for the element.” — by what? Assume Drawing Editor ...

Model categories:

Tool, Creation Tool

Model interfaces to the system:

user — don’t need to model user explicitly
cursor — cursor motion handled by operating system

...
Class Selection Rationale ...

Model values of attributes, not attributes themselves:
- height of the rectangle, width of the rectangle
- major radius, minor radius
- position — of first text character; probably Point attribute
- mode of operation — attribute of Drawing Editor
- shape of the cursor, I-beam, crosshair — attributes of Cursor
- corner — attribute of Rectangle
- time — an implicit attribute of the system
Candidate Classes

Preliminary analysis yields the following candidates:

- Character
- Control Point
- Creation Tool
- Current Selection
- Drawing
- Drawing Editor
- Drawing Element
- Ellipse Creation Tool
- Ellipse Element
- Line Creation Tool
- Line Element
- Point
- Rectangle
- Rectangle Creation Tool
- Rectangle Element
- Selection Tool
- Text Creation Tool
- Text Element
- Tool

Expect the list to evolve as design progresses.
CRC Cards

Use CRC cards to record candidate classes:

<table>
<thead>
<tr>
<th>Text Creation Tool</th>
<th>subclass of Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editing Text</td>
<td>Text Element</td>
</tr>
</tbody>
</table>

Record the candidate **Class Name** and **superclass** (if known)
Record each **Responsibility** and the **Collaborating classes**

- compact, easy to manipulate, easy to modify or discard!
- easy to arrange, reorganize
- easy to retrieve discarded classes
CRC Sessions

CRC cards are not a specification of a design. They are a tool to explore possible designs.

- Prepare a CRC card for each candidate class
- Get a team of Developers to sit around a table and distribute the cards to the team
- The team walks through scenarios, playing the roles of the classes.

This exercise will uncover:
- unneeded classes and responsibilities
- missing classes and responsibilities
Responsibilities

What are responsibilities?

- the knowledge an object maintains and provides
- the actions it can perform

Responsibilities represent the public services an object may provide to clients (but not the way in which those services may be implemented)

- specify what an object does, not how it does it
- don’t describe the interface yet, only conceptual responsibilities
Identifying Responsibilities

- Study the requirements specification:
  - highlight *verbs* and determine which represent responsibilities
  - perform a *walk-through* of the system
    - explore as many scenarios as possible
    - identify actions resulting from input to the system

- Study the candidate classes:
  - class names ⇒ roles ⇒ responsibilities
  - recorded purposes on class cards ⇒ responsibilities
Assigning Responsibilities

- **Evenly distribute** system intelligence
  - avoid procedural centralization of responsibilities
  - keep responsibilities close to objects rather than their clients

- State responsibilities as *generally* as possible
  - “draw yourself” vs. “draw a line/rectangle etc.”

- Keep *behaviour* together with any *related information*
  - principle of encapsulation

...
Assigning Responsibilities ...

- Keep information about one thing in one place
  - if multiple objects need access to the same information
    (i) a new object may be introduced to manage the information, or
    (ii) one object may be an obvious candidate, or
    (iii) the multiple objects may need to be collapsed into a single one

- Share responsibilities among related objects
  - break down complex responsibilities
Relationships Between Classes

Additional responsibilities can be uncovered by examining relationships between classes, especially:

- The “Is-Kind-Of” Relationship:
  - classes sharing a common attribute often share a common superclass
  - common superclasses suggest common responsibilities
  
  e.g., to create a new Drawing Element, a Creation Tool must:

  1. accept user input
  2. determine location to place it
  3. instantiate the element

implemented in subclass

implemented in subclass
Relationships Between Classes …

- The “Is-Analogous-To” Relationship:
  - *similarities* between classes suggest as-yet undiscovered superclasses

- The “Is-Part-Of” Relationship:
  - *distinguish* (don’t share) responsibilities of *part* and of *whole*

Difficulties in assigning responsibilities suggest:
- *missing classes* in design, or
- *free choice* between multiple classes
Collaborations

What are collaborations?

- collaborations are client requests to servers needed to fulfil responsibilities
- collaborations reveal control and information flow and, ultimately, subsystems
- collaborations can uncover missing responsibilities
- analysis of communication patterns can reveal misassigned responsibilities
Finding Collaborations

For each responsibility:
1. Can the class *fulfil* the responsibility *by itself*?
2. If not, *what does it need*, and from what other class can it obtain what it needs?

For each class:
1. What does this class *know*?
2. What *other classes* need its information or results?
   Check for collaborations.
3. Classes that *do not interact* with others should be *discarded*. (Check carefully!)

...
Finding Abstract Classes

Abstract classes factor out common behaviour shared by other classes

- group related classes with common attributes
- introduce abstract superclasses to represent the group
- “categories” are good candidates for abstract classes

✔ Warning: beware of premature classification; your hierarchy will evolve
Sharing Responsibilities

Concrete classes may be both instantiated and inherited from.
Abstract classes may only be inherited from.

Note on class cards and on class diagram.

Venn Diagrams can be used to visualize shared responsibilities.
(Warning: not part of UML!)
Multiple Inheritance

Decide whether a class will be instantiated to determine if it is abstract or concrete.

Responsibilities of subclasses are larger than those of superclasses. Intersections represent common superclasses.
Building Good Hierarchies

Model a “kind-of” hierarchy:

- Subclasses should support all inherited responsibilities, and possibly more

Factor common responsibilities as high as possible:

- Classes that share common responsibilities should inherit from a common abstract superclass; introduce any that are missing
Building Good Hierarchies …

Make sure that abstract classes do not inherit from concrete classes:

- Eliminate by introducing common abstract superclass: abstract classes should support responsibilities in an implementation-independent way

Eliminate classes that do not add functionality:

- Classes should either add new responsibilities, or a particular way of implementing inherited ones
Building Kind-Of Hierarchies

Correctly Formed Subclass Responsibilities:

C assumes *all* the responsibilities of both A and B
Building Kind-Of Hierarchies ...

Incorrect Subclass/Superclass Relationships

$G$ assumes only some of the responsibilities inherited from $E$

Revised Inheritance Relationships

Introduce abstract superclasses to encapsulate common responsibilities
Refactoring Responsibilities

Lines, Ellipses and Rectangles are responsible for keeping track of the width and colour of the lines they are drawn with.
This suggests a common superclass.
Protocols

A *protocol* is a *set of signatures* (i.e., an *interface*) to which a class will respond.

- Generally, protocols are specified for *public responsibilities*
- Protocols for *private* responsibilities should be specified if they will be used or implemented by *subclasses*

1. Construct protocols for each class
2. Write a design specification for each class and subsystem
3. Write a design specification for each contract
What you should know!

- What criteria can you use to identify potential classes?
- How can CRC cards help during analysis and design?
- How can you identify abstract classes?
- What are class responsibilities, and how can you identify them?
- How can identification of responsibilities help in identifying classes?
- What are collaborations, and how do they relate to responsibilities?
- How can you identify abstract classes?
- What criteria can you use to design a good class hierarchy?
- How can refactoring responsibilities help to improve a class hierarchy?
Can you answer the following questions?

✎ When should an attribute be promoted to a class?
✎ Why is it useful to organize classes into a hierarchy?
✎ How can you tell if you have captured all the responsibilities and collaborations?
✎ What use is multiple inheritance during design if your programming language does not support it?
5. Modeling Objects and Classes

- Classes, attributes and operations
- Visibility of Features
- Parameterized Classes
- Objects, Associations, Inheritance
- Constraints

Sources

UML

What is UML?
- uniform notation: Booch + OMT + Use Cases (+ state charts)
  - UML is not a method or process
  - .. The Unified Development Process is

Why a Graphical Modeling Language?
- Software projects are carried out in team
- Team members need to communicate
  - ... sometimes even with the end users
- “One picture conveys a thousand words”
  - the question is only which words
  - Need for different views on the same software artifact
Why UML?

Why UML
- Represents de-facto standard
  - more tool support, more people understand your diagrams, less education
- Is reasonably well-defined
  - ... although there are interpretations and dialects
- Is open
  - stereotypes, tags and constraints to extend basic constructs
  - has a meta-meta-model for advanced extensions
UML History

- 1994: Grady Booch (Booch method) + James Rumbaugh (OMT) at Rational
- 1994: Ivar Jacobson (OOSE, use cases) joined Rational
  “The three amigos”
- 1996: Rational formed a consortium to support UML
- January, 1997: UML1.0 submitted to OMG by consortium
- November, 1997: UML 1.1 accepted as OMG standard
  However, OMG names it UML1.0
- December, 1998: UML task force cleans up standard in UML1.2
- June, 1999: UML task force cleans up standard in UML1.3
- ...: Major revision to UML2.0
Class Diagrams

“Class diagrams show generic descriptions of possible systems, and object diagrams show particular instantiations of systems and their behaviour.”

Class name, attributes and operations:

A collapsed class view:

Class with Package name:

Attributes and operations are also collectively called features.
Visibility and Scope of Features

**Stereotype**
(what “kind” of class is it?)

**underlined attributes** have class scope

+ = “public”
# = “protected”
− = “private”

An ellipsis signals that further entries are not shown

**User-defined properties**
(e.g., readonly, owner = “Pingu”)

**italic attributes** are abstract

```
Window
{ abstract }
+size: Area = (100, 100)
#visibility: Boolean = false
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindow*
+display ()
+hide ()
+create ()
-attachXWindow (xwin: Xwindow*)
...
```
Attributes and Operations

**Attributes** are specified as:

```plaintext
name: type = initialValue { property string }
```

**Operations** are specified as:

```plaintext
name (param: type = defaultValue, ...) : resultType
```
UML Lines and Arrows

- Constraint
  (usually annotated)

- Dependency
  e.g., «requires», «imports»...

- Realization
  e.g., class/template, class/interface

Association
  e.g., «uses»

Navigable
  association
  e.g., part-of

“Generalization”
  i.e., specialization (!)
  e.g., class/superclass,
  concrete/abstract class

Aggregation
  i.e., “consists of”

“Composition”
  i.e., containment
Parameterized Classes

Parameterized (aka “template” or “generic”) classes are depicted with their parameters shown in a dashed box. Parameters may be either types (just a name) or values (name: Type).

Instantiation of a class from a template can be shown by a dashed arrow (Realization).

NB: All forms of arrows (directed arcs) go from the client to the supplier!
Interfaces

Interfaces, equivalent to abstract classes with no attributes, are represented as classes with the stereotype «interface» or, alternatively, with the “Lollipop-Notation”:

<table>
<thead>
<tr>
<th>«interface»</th>
<th>Comparable</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEqual(String): Boolean</td>
<td></td>
</tr>
<tr>
<td>hash(): Integer</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>isEqual(String): Boolean</td>
</tr>
<tr>
<td>hash(): Integer</td>
</tr>
</tbody>
</table>

NB: Interfaces cannot have (navigable) associations!
Utilities

A utility is a grouping of global attributes and operations. It is represented as a class with the stereotype «utility». Utilities may be parameterized.

```
«utility»
MathPack

randomSeed : long = 0
pi : long = 3.14159265358979
sin (angle : double) : double
cos (angle : double) : double
random () : double
```

return \( \sin(\text{angle} + \pi/2.0) \);

NB: A utility’s attributes are already interpreted as being in class scope, so it is redundant to underline them.

A “note” is a text comment associated with a view, and represented as box with the top right corner folded over.
# Objects

*Objects* are shown as rectangles with their name and type underlined in one compartment, and attribute values, optionally, in a second compartment.

<table>
<thead>
<tr>
<th>triangle1: Polygon</th>
</tr>
</thead>
<tbody>
<tr>
<td>centre = (0, 0)</td>
</tr>
<tr>
<td>vertices = ((0,0), (4,0), (4,3))</td>
</tr>
<tr>
<td>borderColour = black</td>
</tr>
<tr>
<td>fillColour = white</td>
</tr>
</tbody>
</table>

At least one of the name or the type must be present.
Associations represent structural relationships between objects of different classes.

- usually binary (but may be ternary etc.)
- optional name and direction
- (unique) role names and multiplicities at end-points
- can traverse using navigation expressions
  e.g., Sandoz.employee[name = “Pingu“].boss
Aggregation and Navigability

Aggregation is denoted by a *diamond* and indicates a part-whole dependency:

A *hollow* diamond indicates a *reference*; a *solid* diamond an *implementation*.

If the link terminates with an arrowhead, then one can *navigate* from the whole to the part.

If the multiplicity of a role is > 1, it may be marked as *{ordered}* or as *{sorted}*.
Association Classes

An association may be an instance of an *association class*:

In many cases the association class only stores attributes, and its name can be left out.
A **qualified association** uses a special **qualifier value** to identify the object at the other end of the association.

“The multiplicity attached to the target role denotes the possible cardinalities of the set of target objects selected by the pairing of a source object and a qualifier value.”

**NB:** Qualifiers are part of the association, not the class.
Inheritance

A subclass inherits the features of its superclasses:

- **Figure1dim**
  - { abstract }
  - colour
  - display ( )

- **Drawing**

- **Line**
  - endpoints
  - display ( )

- **Arc**
  - radius
  - start angle
  - arc angle
  - display ( )

- **Spline**
  - control points
  - display ( )
What is Inheritance For?

New software often builds on old software by imitation, refinement or combination.

Similarly, classes may be extensions, specializations or combinations of existing classes.
Inheritance supports ...

Conceptual hierarchy:
- conceptually related classes can be organized into a specialization hierarchy
  - people, employees, managers
  - geometric objects ...

Software reuse:
- related classes may share interfaces, data structures or behaviour
  - geometric objects ...

Polymorphism:
- objects of distinct, but related classes may be uniformly treated by clients
  - array of geometric objects
Design Patterns can be represented as "parameterized collaborations":

```
Client -> Target
  Request()

Adapter
  Request()

Adaptee
  SpecificRequest()

adaptee
  adaptee.SpecificRequest()
```
Instantiating Design Patterns

A Design Pattern in use (an *instantiation*) can be described with a *dashed oval*:

![Diagram showing the instantiation of a design pattern with a dashed oval representing the instantiator. The diagram includes classes like Shape, TextView, TextShape, and Adaptee, with arrows indicating inheritance and association relationships.]
**Constraints**

*Constraints* are *restrictions* on values attached to classes or associations.

- **Binary constraints** may be shown as dashed lines between elements.

- **Derived values** and associations can be marked with a “/”

```plaintext
{ age = currentDate - birthdate }
```
Specifying Constraints

Constraints are specified between *braces*, either free or within a note:

\[
\{ \text{Person.employer} = \text{Person.boss.employer} \}
\]
Design by Contract in UML

Combine constraints with stereotypes:

«invariant»
(isEmpty()) or (!isEmpty())

«precondition»
oldSize > 0

«postcondition»
(!isEmpty()) and (top() = char)

let
oldSize:Integer = self.size
in
pre: oldSize > 0
post: self.size = oldSize - 1

(OCL)

NB: «invariant», «precondition», and «postcondition» are predefined in UML.
Using the Notation

During Analysis:
- Capture classes visible to users
- Document attributes and responsibilities
- Identify associations and collaborations
- Identify conceptual hierarchies
- Capture all visible features

...
Using the Notation ...

During Design:

- Specify contracts and operations
- Decompose complex objects
- Factor out common interfaces and functionalities

The graphical notation is only part of the analysis or design document. For example, a data dictionary cataloguing and describing all names of classes, roles, associations, etc. must be maintained throughout the project.
What you should know!

✎ How do you represent classes, objects and associations?
✎ How do you specify the visibility of attributes and operations to clients?
✎ How is a utility different from a class? How is it similar?
✎ Why do we need both named associations and roles?
✎ Why is inheritance useful in analysis? In design?
✎ How are constraints specified?
Can you answer the following questions?

- Why would you want a feature to have *class scope*?
- Why *don’t* you need to show *operations* when depicting an *object*?
- Why aren’t *associations* drawn with *arrowheads*?
- How is *aggregation* different from any other kind of association?
- How are *associations* realized in an *implementation language*?
6. Modeling Behaviour

- Use Case Diagrams
- Sequence Diagrams
- Collaboration Diagrams
- State Diagrams

Sources:
A use case is a generic description of an entire transaction involving several actors.

A use case diagram presents a set of use cases (ellipses) and the external actors that interact with the system. Dependencies and associations between use cases may be indicated.
Scenarios

A **scenario** is an **instance** of a use case showing a **typical example** of its execution.

Scenarios can be presented in UML using either **sequence diagrams** or **collaboration diagrams**.

Note that a scenario only describes an **example** of a use case, so conditionality cannot be expressed!
Sequence Diagrams

A sequence diagram depicts a scenario by showing the interactions among a set of objects in temporal order.

Objects (not classes!) are shown as vertical bars. Events or message dispatches are shown as horizontal (or slanted) arrows from the sender to the receiver.

Temporal constraints between events may also be expressed.
UML Message Flow Notation

- Filled solid arrowhead
  *procedure call* or other nested control flow

- Stick arrowhead
  flat, *sequential* control flow

- Half-stick arrowhead
  *asynchronous* control flow between objects within a procedural sequence
Collaboration diagrams depict scenarios as flows of messages between objects:

1: displayPositions(window)

1.1*[i=1..n]: drawSegment(i)

1.1.1a: r0 := position()

1.1.1b: r1 := position()

1.1.2: line := create(r0, r1)

1.1.3: display(window)

add(self)
Message Labels

Messages from one object to another are labelled with text strings showing the direction of message flow and information indicating the message sequence.

1. **Prior messages** from other threads (e.g. “[A1.3, B6.7.1]”) only needed with concurrent flow of control

2. **Dot-separated list** of sequencing elements
   - sequencing integer (e.g., “3.1.2” is invoked by “3.1” and follows “3.1.1”)
   - letter indicating concurrent threads (e.g., “1.2a” and “1.2b”)
   - iteration indicator (e.g., “1.1*[i=1..n]”)
   - conditional indicator (e.g., “2.3 [#items = 0]”)

...
Message Labels …

3. *Return value* binding (e.g., “status :=“)
4. Message *name*
   - event or operation name
5. Argument list
State Diagrams

- **Idle**
  - caller hangs up / disconnect
  - lift receiver / get dial tone

- **DialTone**
  - do/play dial tone

- **Dialing**
  - dial digit(n)
  - after 15 sec.
  - dial digit(n) [invalid]
  - dial digit(n) [valid] / connect

- **Pinned**
  - callee answers
  - callee hangs up

- **Talking**
  - callee answers / enable speech

- **Active**
  - after 15 sec.
  - dial digit(n)

- **Connecting**
  - busy
  - connected

- **Ringing**
  - do/play ringing tone

- **Timeout**
  - do/play message
  - after 15 sec.
State Diagram Notation

A State Diagram describes the temporal evolution of an object of a given class in response to interactions with other objects inside or outside the system.

An event is a one-way (asynchronous) communication from one object to another:
- atomic (non-interruptible)
- includes events from hardware and real-world objects e.g., message receipt, input event, elapsed time, ...
- notation: eventName(parameter: type, ...)
- may cause object to make a transition between states
State Diagram Notation …

A *state* is a period of time during which an object is *waiting* for an event to occur:

- depicted as *rounded box* with (up to) three sections:
  - name — optional
  - state variables — name: type = value (valid only for that state)
  - triggered operations — internal transitions and ongoing operations
- may be *nested*
State Box with Regions

The **entry** event occurs whenever a transition is made into this state, and the **exit** operation is triggered when a transition is made out of this state.

The **help** and **character** events cause internal transitions with no change of state, so the entry and exit operations are not performed.

<table>
<thead>
<tr>
<th>Typing Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry / set echo invisible</td>
</tr>
<tr>
<td>exit / set echo normal</td>
</tr>
<tr>
<td>character / handle character</td>
</tr>
<tr>
<td>help / display help</td>
</tr>
</tbody>
</table>

**name**

**internal operations**
Transitions

A transition is an response to an external event received by an object in a given state

- May invoke an operation, and cause the object to change state
- May send an event to an external object
- Transition syntax (each part is optional):
  
  \[
  \text{event(arguments)} \ [\text{condition}] / ^\text{target}.\text{sendEvent} \ \text{operation(arguments)}
  \]

- External transitions label arcs between states
- Internal transitions are part of the triggered operations of a state
Operations and Activities

An **operation** is an *atomic action* invoked by a *transition*

- Entry and exit operations can be associated with states

An **activity** is an *ongoing operation* that takes place while object is in a given state

- Modelled as “internal transitions” labelled with the pseudo-event **do**
Composite States

Composite states may be depicted either as high-level or low-level views. “Stubbed transitions” indicate the presence of internal states:

Initial and terminal substates are shown as black spots and “bulls-eyes”:

- **Idle**
- **Dialing**
- **Connecting**

- **Start**
  - do / play dial tone

- **Partial Dial**
  - entry/number.append(n)
  - [number.isValid()]
  - ^dialedNumber(number)
Sending Events between Objects

VCR

Off

On

toggle Power

toggle Power

Remote Control

TV mode

VCR button

Power button

TV button

Power button

toggle Power

Television

Off

On

toggle Power

toggle Power
Concurrent Substates

- Lab1: lab done
- Lab2: lab done
- Term Project: project done
- Final Test: pass, fail

States: Incomplete, Passed, Failed

Taking Class
Branching and Merging

Entering concurrent states:
*Entering* a state with concurrent substates means that *each* of the substates is entered concurrently (one logical thread per substate).

Leaving concurrent states:
A *labelled transition* out of any of the substates *terminates all* of the substates.
An *unlabelled transition* out of the overall state *waits* for all substates to terminate.
Branching and Merging …

An alternative notation for explicit branching and merging uses a “synchronization bar”:

![Diagram](image-url)
A “history indicator” can be used to indicate that the current composite state should be remembered upon an external transition. To return to the saved state, a transition should point explicitly to the history icon:
Creating and Destroying Objects

Creation and destruction of objects can be depicted by using the start and terminal symbols as top-level states:

![Diagram showing object states and transitions]

- **Create**
- **Writeable**
- **ReadOnly**
- **Modify**
- **Unlock**
- **Lock**
- **Destroy**
Using the Notations

The diagrams introduced here complement class and object diagrams.

During Analysis:
- Use case, sequence and collaboration diagrams document use cases and their scenarios during requirements specification.

During Design:
- Sequence and collaboration diagrams can be used to document implementation scenarios or refine use case scenarios.
- State diagrams document internal behaviour of classes and must be validated against the specified use cases.
What you should know!

✎ What is the purpose of a use case diagram?
✎ Why do scenarios depict objects but not classes?
✎ How can timing constraints be expressed in scenarios?
✎ How do you specify and interpret message labels in a scenario?
✎ How do you use nested state diagrams to model object behaviour?
✎ What is the difference between “external” and “internal” transitions?
✎ How can you model interaction between state diagrams for several classes?
Can you answer the following questions?

 Projectile: 

- Can a sequence diagram always be translated to an collaboration diagram?
- Or vice versa?
- Why are arrows depicted with the message labels rather than with links?
- When should you use concurrent substates?
7. User Interface Design

Overview:
- Interface design models
- Design principles
- Information presentation
- User Guidance
- Evaluation

Sources:
Interface Design Models

Four different models occur in HCI design:

1. The design model expresses the software design.
2. The user model describes the profile of the end users. (i.e., novices vs. experts, cultural background, etc.)
3. The user’s model is the end users’ perception of the system.
4. The system image is the external manifestation of the system (look and feel + documentation etc.)
# User Interface Design Principles

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User familiarity</strong></td>
<td>Use terms and concepts <em>familiar to the user</em>.</td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
<td><em>Comparable</em> operations should be <em>activated in the same way</em>. Commands and menus should have the same format, etc.</td>
</tr>
<tr>
<td><strong>Minimal surprise</strong></td>
<td>If a command operates in a known way, the user <em>should be able to predict</em> the operation of comparable commands.</td>
</tr>
<tr>
<td><strong>Feedback</strong></td>
<td>Provide the user with visual and auditory feedback, maintaining <em>two-way communication</em>.</td>
</tr>
<tr>
<td>Principle</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Memory load</strong></td>
<td>Reduce the amount of information that must be remembered between actions. Minimize the memory load.</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>Seek efficiency in dialogue, motion and thought. Minimize keystrokes and mouse movements.</td>
</tr>
<tr>
<td><strong>Recoverability</strong></td>
<td>Allow users to recover from their errors. Include undo facilities, confirmation of destructive actions, 'soft' deletes, etc.</td>
</tr>
<tr>
<td><strong>User guidance</strong></td>
<td>Incorporate some form of context-sensitive user guidance and assistance.</td>
</tr>
</tbody>
</table>

© O. Nierstrasz — U. Berne  
User Interface Design
### GUI Characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Windows</strong></td>
<td>Multiple windows allow <em>different information</em> to be displayed <em>simultaneously</em> on the user’s screen.</td>
</tr>
<tr>
<td><strong>Icons</strong></td>
<td>Usually icons represent <em>files</em> (including folders and applications), but they may also stand for <em>processes</em> (e.g., printer drivers).</td>
</tr>
<tr>
<td><strong>Menus</strong></td>
<td>Menus bundle and organize <em>commands</em> (eliminating the need for a command language).</td>
</tr>
<tr>
<td>Characteristic</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Pointing</strong></td>
<td>A pointing device such as a mouse is used for <em>commands</em> choices from a menu or indicating items of interest in a window.</td>
</tr>
<tr>
<td><strong>Graphics</strong></td>
<td>Graphical elements can be <em>commands</em> on the same display.</td>
</tr>
</tbody>
</table>
GUI advantages

- They are *easy to learn* and use.
  - Users without experience can learn to use the system quickly.

- The user may *switch attention* between tasks and applications.
  - Information remains visible in its own window when attention is switched.

- *Fast, full-screen interaction* is possible with immediate access to the entire screen
GUI (dis) advantages ...

But

- A GUI is not automatically a good interface
  - Many software systems are never used due to poor UI design
  - A poorly designed UI can cause a user to make catastrophic errors

![Comic strip](image-url)
Direct Manipulation

A direct manipulation interface presents the user with a model of the information space which is modified by direct action.

Examples
- forms (direct entry)
- WYSIWYG document and graphics editors

...
Direct Manipulation ...

Advantages

- Users feel in control and are less likely to be intimidated by the system
- User learning time is relatively short
- Users get immediate feedback on their actions
  - mistakes can be quickly detected and corrected

Problems

- Finding the right user metaphor may be difficult
- It can be hard to navigate efficiently in a large information space.
- It can be complex to program and demanding to execute
Interface Models

Desktop metaphor.
- The model of an interface is a “desktop” with icons representing files, cabinets, etc.

Control panel metaphor.
- The model of an interface is a hardware control panel with interface entities including:
  - buttons, switches, menus, lights, displays, sliders etc.
Menu Systems

Menu systems allow users to make a selection from a list of possibilities presented to them by the system by pointing and clicking with a mouse, using cursor keys or by typing (part of) the name of the selection.
Menu Systems ... 

Advantages

- Users don’t need to remember command names
- Typing effort is minimal
- User errors are trapped by the interface
- Context-dependent help can be provided (based on the current menu selection)

Problems

- Actions involving logical conjunction (and) or disjunction (or) are awkward to represent
- If there are many choices, some menu structuring facility must be used
- Experienced users find menus slower than command language
Menu Structuring

Scrolling menus
- The menu can be scrolled to reveal additional choices
- Not practical if there is a very large number of choices

Hierarchical menus
- Selecting a menu item causes the menu to be replaced by a sub-menu

Walking menus
- A menu selection causes another menu to be revealed

Associated control panels
- When a menu item is selected, a control panel pops-up with further options
Command Interfaces

With a command language, the user types commands to give instructions to the system

- May be implemented using cheap terminals
- Easy to process using compiler techniques
- Commands of arbitrary complexity can be created by command combination
- Concise interfaces requiring minimal typing can be created
Command Interfaces ...

Advantages

❑ Allow experienced users to *interact quickly* with the system
❑ Commands can be *scripted* (!)

Problems

❑ Users have to *learn and remember* a command language
❑ Not suitable for *occasional* or inexperienced users
❑ An *error detection* and recovery system is required
❑ *Typing* ability is required
Information Presentation Factors

- Is the user interested in precise information or data relationships?
- How quickly do information values change? Must the change be indicated immediately?
- Must the user take some action in response to a change?
- Is there a direct manipulation interface?
- Is the information textual or numeric? Are relative values important?
Analogue vs. Digital Presentation

Digital presentation
- Compact — takes up little screen space
- Precise values can be communicated

Analogue presentation
- Easier to get an 'at a glance' *impression* of a value
- Possible to show *relative* values
- Easier to see *exceptional* data values

© Ian Sommerville 1995
Colour Use Guidelines

Colour can help the user understand complex information structures.

- Don’t use (only) colour to communicate meaning!
  - Open to misinterpretation (colour-blindness, cultural differences ...)
  - Design for monochrome then add colour

- Use colour coding to support user tasks
  - highlight exceptional events
  - allow users to control colour coding

- Use colour change to show status change

- Don’t use too many colours
  - Avoid colour pairings which clash

- Use colour coding consistently
User Guidance

The user guidance system is integrated with the user interface to help users when they need information about the system or when they make some kind of error.

User guidance covers:
- System messages, including error messages
- Documentation provided for users
- On-line help
## Design Factors in Message Wording

<table>
<thead>
<tr>
<th><strong>Context</strong></th>
<th>The user guidance system should be aware of what the user is doing and should <em>adjust the output message to the current context</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Experience</strong></td>
<td>The user guidance system should provide both longer, <em>explanatory messages for beginners</em>, and more <em>terse messages for experienced users</em>.</td>
</tr>
<tr>
<td><strong>Skill level</strong></td>
<td>Messages should be <em>tailored to the user’s skills</em> as well as their experience. I.e., depending on the <em>terminology</em> which is familiar to the reader.</td>
</tr>
<tr>
<td>Style</td>
<td>Messages should be positive rather than negative. They should never be insulting or try to be funny.</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Culture</td>
<td>Wherever possible, the designer of messages should be familiar with the culture of the country (or environment) where the system is used. (A suitable message for one culture might be unacceptable in another!)</td>
</tr>
</tbody>
</table>
Error Message Guidelines

- Speak the *user’s language*
- Give *constructive advice* for recovering from the error
- Indicate *negative consequences* of the error (e.g., possibly corrupted files)
- Give an audible or visual cue
- Don’t make the user feel guilty!
Good and Bad Error Messages

The application “Convert To GIF” has crashed (Unknown floating point instruction).

Norton CrashGuard recommends that you quit the application, but if you have unsaved data, try to fix the crash.

Try To Fix  Restart  Quit Application

Sorry, a system error occurred.
"Convert to GIF"
error type 10

Restart
Help System Design

Help? means “Please help. I want information.”
Help! means “HELP. I’m in trouble.”

Help information

- Should not simply be an on-line manual
  - Screens or windows don’t map well onto paper pages
- Dynamic characteristics of display can improve information presentation
  - but people are not so good at reading screens as they are text.
Help system use

- **Multiple entry points** should be provided
  - the user should be able to get help from different places
- The help system should indicate *where the user is positioned*
- **Navigation and traversal** facilities must be provided
User Interface Evaluation

User interface design should be evaluated to assess its suitability and usability.
# Usability attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learnability</td>
<td>How long does it take a new user to become <em>productive</em> with the system?</td>
</tr>
<tr>
<td>Speed of operation</td>
<td>How well does the system <em>response</em> match the user’s work <em>practice</em>?</td>
</tr>
<tr>
<td>Robustness</td>
<td>How <em>tolerant</em> is the system of user error?</td>
</tr>
<tr>
<td>Recoverability</td>
<td>How good is the system at <em>recovering</em> from user errors?</td>
</tr>
<tr>
<td>Adaptability</td>
<td>How closely is the system tied to a <em>single model</em> of work?</td>
</tr>
</tbody>
</table>
What you should know!

- What *models* are important to keep in mind in UI design?
- What is the principle of *minimal surprise*?
- What problems arise in designing a good *direct manipulation interface*?
- What are the trade-offs between *menu systems and command languages*?
- How can you use *colour* to improve a UI?
- In what way can a help system be *context sensitive*?
Can you answer the following questions?

- Why is it important to offer "keyboard short-cuts" for equivalent mouse actions?
- How would you present the current load on the system? Over time?
- What is the worst UI you ever used? Which design principles did it violate?
- What’s the worst web site you’ve used recently? How would you fix it?
- What’s good or bad about the MS-Word help system?
8. Software Validation

Overview:

- Reliability, Failures and Faults
- Fault Tolerance
- Software Testing: Black box and white box testing
- Static Verification

Source:

Software Reliability, Failures and Faults

The reliability of a software system is a measure of how well it provides the services expected by its users, expressed in terms of software failures.

- A software failure is an execution event where the software behaves in an unexpected or undesirable way.

- A software fault is an erroneous portion of a software system which may cause failures to occur if it is run in a particular state, or with particular inputs.
# Kinds of failures

<table>
<thead>
<tr>
<th>Failure class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transient</td>
<td>Occurs only with <em>certain inputs</em></td>
</tr>
<tr>
<td>Permanent</td>
<td>Occurs with <em>all inputs</em></td>
</tr>
<tr>
<td>Recoverable</td>
<td>System can <em>recover</em> without operator intervention</td>
</tr>
<tr>
<td>Unrecoverable</td>
<td>Operator <em>intervention</em> is needed to recover from failure</td>
</tr>
<tr>
<td>Non-corrupting</td>
<td>Failure does <em>not corrupt</em> data</td>
</tr>
<tr>
<td>Corrupting</td>
<td>Failure <em>corrupts</em> system data</td>
</tr>
</tbody>
</table>
Programming for Reliability

Fault avoidance:
- development techniques to reduce the number of faults in a system

Fault tolerance:
- developing programs that will operate despite the presence of faults
Fault Avoidance

Fault avoidance depends on:

1. A precise *system specification* (preferably formal)
2. Software design based on *information hiding* and *encapsulation*
3. Extensive validation *reviews* during the development process
4. An organizational *quality philosophy* to drive the software process
5. Planned *system testing* to expose faults and assess reliability
Common Sources of Software Faults

Several features of programming languages and systems are common sources of faults in software systems:

- **Goto statements** and other unstructured programming constructs make programs hard to understand, reason about and modify.
  - Use structured programming constructs

- **Floating point numbers** are inherently imprecise and may lead to invalid comparisons.
  - Fixed point numbers are safer for exact comparisons

- **Pointers** are dangerous because of aliasing, and the risk of corrupting memory
  - Pointer usage should be confined to abstract data type implementations

...
Common Sources of Software Faults . . .

- **Parallelism** is dangerous because timing differences can affect overall program behaviour in hard-to-predict ways.
  
  - Minimize inter-process dependencies

- **Recursion** can lead to convoluted logic, and may exhaust (stack) memory.
  
  - Use recursion in a disciplined way, within a controlled scope

- **Interrupts** force transfer of control independent of the current context, and may cause a critical operation to be terminated.
  
  - Minimize the use of interrupts; prefer disciplined exceptions
Fault Tolerance

A fault-tolerant system must carry out four activities:

1. Failure detection: detect that the system has reached a particular state or will result in a system failure
2. Damage assessment: detect which parts of the system state have been affected by the failure
3. Fault recovery: restore the state to a known, “safe” state (either by correcting the damaged state, or backing up to a previous, safe state)
4. Fault repair: modify the system so the fault does not recur (!)
Approaches to Fault Tolerance

N-version Programming:

*Multiple versions* of the software system are implemented *independently* by different teams.

The final system:

- runs all the versions in *parallel*,
- *compares* their results using a voting system, and
- *rejects* inconsistent outputs. (At least three versions should be available!)
Approaches to Fault Tolerance …

Recovery Blocks:

A finer-grained approach in which a program unit contains a test to check for failure, and alternative code to back up and try in case of failure.

- alternatives are executed in sequence, not in parallel
- the failure test is independent (not by voting)
Defensive Programming

Failure detection:

- Use the *type system* as much as possible to ensure that state variables do not get assigned invalid values.
- Use *assertions* to detect failures and raise exceptions. Explicitly state and check all invariants for abstract data types, and pre- and post-conditions of procedures as assertions. Use exception handlers to recover from failures.
- Use *damage assessment* procedures, where appropriate, to assess what parts of the state have been affected, before attempting to fix the damage.
Defensive Programming ...

Fault recovery:

- **Backward recovery**: backup to a previous, consistent state
- **Forward recovery**: make use of redundant information to reconstruct a consistent state from corrupted data
Verification and Validation

Verification:
- Are we building the product right?
  — i.e., does it conform to specs?

Validation:
- Are we building the right product?
  — i.e., does it meet expectations?
Verification and Validation ...

Static techniques include program inspection, analysis and formal verification.

Dynamic techniques include statistical testing and defect testing ...

The Testing Process

1. **Unit** testing:
   - Individual (stand-alone) *components* are tested to ensure that they operate correctly.

2. **Module** testing:
   - A collection of *related components* (a module) is tested as a group.

3. **Sub-system** testing:
   - The phase tests a *set of modules* integrated as a sub-system. Since the most common problems in large systems arise from sub-system interface mismatches, this phase focuses on testing these interfaces.

...
4. **System** testing:
   - This phase concentrates on (i) detecting errors resulting from *unexpected interactions* between subsystems, and (ii) validating that the complete systems fulfils *functional* and *non-functional* requirements.

5. **Acceptance** testing (alpha/beta testing):
   - The system is tested with *real* rather than simulated *data*.

*Testing is iterative!* **Regression testing** is performed when defects are repaired.
Regression testing

Regression testing means testing that everything that used to work *still works* after changes are made to the system!

- tests must be *deterministic* and *repeatable*
- should test “all” functionality
  - every interface
  - all boundary situations
  - every feature
  - every line of code
  - everything that can conceivably go wrong!

*It costs extra work to define tests up front, but they pay off in debugging & maintenance!*
Test Planning

The preparation of the test plan should begin \textit{when the system requirements are formulated}, and the plan should be developed \textit{in detail as the software is designed}. The plan should be \textit{revised regularly}, and tests should be \textit{repeated and extended} where the software process iterates.
Top-down Testing

- **Start with sub-systems**, where modules are represented by “stubs”
- Similarly test modules, representing functions as stubs
- **Coding** and **testing** are carried out as a single activity
- **Design errors** can be detected early on, avoiding expensive redesign
- Always have a **running** (if limited) system!

- **BUT**: may be impractical for stubs to simulate complex components
Bottom-up Testing

- Start by testing units and modules
- Test drivers must be written to exercise lower-level components
- Works well for reusable components to be shared with other projects

**BUT:** pure bottom-up testing will not uncover architectural faults till late in the software process

Typically a combination of top-down and bottom-up testing is best.
Defect Testing

Tests are designed to reveal the presence of defects in the system.

Testing should, in principle, be exhaustive, but in practice can only be representative.

Test data are inputs devised to test the system.

Test cases are input/output specifications for a particular function being tested.
Defect Testing …

Petschenik (1985) proposes:

1. “Testing a system’s capabilities is more important than testing its components.”
   - Choose test cases that will identify situations that may prevent users from doing their job.

2. “Testing old capabilities is more important than testing new capabilities.”
   - Always perform regression tests when the system is modified.

3. “Testing typical situations is more important than testing boundary value cases.”
   - If resources are limited, focus on typical usage patterns.
Functional (black box) testing

*Functional testing* treats a component as a “black box” whose behaviour can be determined only by studying its *inputs and outputs*.

- **Input set** $I_e$
- **Output set** $O_e$
- **Component**

- **Inputs causing anomalous behaviour**
- **Outputs revealing the presence of defects**
Coverage Criteria

Test cases are derived from the external specification of the component and should cover:

- all exceptions
- all data ranges (incl. invalid) generating different classes of output
- all boundary values

Test cases can be derived from a component’s interface, by assuming that the component will behave similarly for all members of an equivalence partition...
Equivalence partitioning

private int[] elements_;
public boolean find(int key) { ... }

Check input partitions:
- Do the inputs fulfil the pre-conditions?
  - is the array sorted, non-empty ...
- Is the key in the array?
  - leads to (at least) 2x2 equivalence classes

Check boundary conditions:
- Is the array of length 1?
- Is the key at the start or end of the array?
  - leads to further subdivisions (not all combinations make sense)
### Test Cases and Test Data

Generate test data that cover all meaningful equivalence partitions.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array length 0</td>
<td>key = 17, elements = {}</td>
</tr>
<tr>
<td>Array not sorted</td>
<td>key = 17, elements = { 33, 20, 17, 18 }</td>
</tr>
<tr>
<td>Array size 1, key in array</td>
<td>key = 17, elements = { 17 }</td>
</tr>
<tr>
<td>Array size 1, key not in array</td>
<td>key = 0, elements = { 17 }</td>
</tr>
<tr>
<td>Array size &gt; 1, key is first element</td>
<td>key = 17, elements = { 17, 18, 20, 33 }</td>
</tr>
<tr>
<td>Array size &gt; 1, key is last element</td>
<td>key = 33, elements = { 17, 18, 20, 33 }</td>
</tr>
<tr>
<td>Array size &gt; 1, key is in middle</td>
<td>key = 20, elements = { 17, 18, 20, 33 }</td>
</tr>
<tr>
<td>Array size &gt; 1, key not in array</td>
<td>key = 50, elements = { 17, 18, 20, 33 }</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Structural testing treats a component as a "white box" or "glass box" whose structure can be examined to generate test cases.

Derive test cases to maximize coverage of that structure, yet minimize number of test cases.
Coverage criteria

- every \textit{statement} at least once
- all \textit{portions of control flow} at least once
- all possible \textit{values of compound conditions} at least once
- all portions of \textit{data flow} at least once
- for all \textit{loops} \( L \), with \( n \) allowable passes:
  (i) skip the loop;
  (ii) 1 pass through the loop
  (iii) 2 passes
  (iv) \( m \) passes where \( 2 < m < n \)
  (v) \( n-1, n, n+1 \) passes

\textit{Path testing} is a white-box strategy which exercises every \textit{independent execution path} through a component.
Binary Search Method

```java
class BinarySearchMethod {
    public boolean find(int key) throws assertionViolation {
        // (1)
        assert(isSorted()); // pre-condition
        if (isEmpty()) { return false; }
        int bottom = 0;
        int top = elements_.length-1;
        int lastIndex = (bottom+top)/2;
        int mid;
        boolean found = key == elements_[lastIndex];

        while ((bottom <= top) && !found) { // (2) (3)
            assert(bottom <= top); // loop invariant
            mid = (bottom + top) / 2;
            found = key == elements_[mid];
        }
    }
}
```
if (found) {
    lastIndex = mid;
} else {
    if (elements_[mid] < key) {
        bottom = mid + 1;
    } else {
        top = mid - 1;
    }
} // loop variant decreases: top - bottom

assert((key == elements_[lastIndex]) || !found);
// post-condition
return found;
Path Testing

Test cases should be chosen to cover all independent paths through a routine:

```
while (bottom <= top)
    if (! found)
        if (key == elements_[mid])
            if (elements_[mid] < key)
                if (elements_[mid] > key)
```

e.g., \{1,2,12,13\}, \{1,2,3,4,12,13\}, \{1,2,3,5,6,11,2,12,13\},
\{1,2,3,5,7,8,10,11,2,12,13\}, \{1,2,3,5,7,9,10,11,2,12,13\} ...
Basis Path Testing: The Technique

See [Press92a]

1. Draw a control flow graph
   Nodes represent nonbranching statements; edges represent control flow.

   ![Control Flow Graphs]
   - if-then-else
   - while
   - case-of
   - and / or

2. Compute the Cyclomatic Complexity
   \[ \text{Cyclomatic Complexity} = \#(edges) - \#(nodes) + 2 = \text{number of conditions} + 1 \]

...
Basis Path Testing …

3. Determine a set of *independent paths*
   Several possibilities. Upper bound = Cyclomatic Complexity
4. Prepare *test cases* that force each of these paths
   Choose values for all variables that control the branches.
   Predict the result in terms of values and/or exceptions raised
5. Write *test driver* for each test case
For complex boolean expressions, Basis Path Testing is not enough! Input values \( \{x = 3, y=4\} \) and \( \{x = 4, y=3\} \) will exercise all paths, but consider \( \{x = 3, y=3\} \) ...

- **Condition Testing** exercises all logical conditions
- **Domain Testing**: for each occurrence of <, <=, =, >, >= 3 tests

```java
public int abs (int x, int y) throws assertionViolation {
    int result;
    if (x > y) {
        result = x - y;
    } else {
        result = y - x;
    }
    assert (result > 0); // post-condition
    return result;
}
```
Statistical Testing

The objective of statistical testing is to determine the reliability of the software, rather than to discover faults.

Reliability may be expressed as:

- **Probability** of failure on demand
  - i.e., for safety-critical systems
- **Rate** of failure occurrence
  - i.e., #failures/time unit
- **Mean time** to failure
  - i.e., for a stable system
- **Availability**
  - i.e., fraction of time, for e.g. telecom systems
Statistical Testing ...

Tests are designed to reflect the frequency of actual user inputs and, after running the tests, an estimate of the operational reliability of the system can be made:

1. **Determine usage patterns** of the system (classes of input and probabilities)
2. **Select or generate test data** corresponding to these patterns
3. **Apply the test cases**, recording execution time to failure
4. Based on a statistically significant number of test runs, **compute reliability**
Static Verification

Program Inspections:

- Small team systematically checks program code
- Inspection checklist often drives this activity
  - e.g., “Are all invariants, pre- and post-conditions checked?” ...

Static Program Analysers:

- Complements compiler to check for common errors
  - e.g., variable use before initialization

...
Static Verification …

Mathematically-based Verification:
- Use mathematical reasoning to demonstrate that program meets specification
  - e.g., that invariants are not violated, that loops terminate, etc.

Cleanroom Software Development:
- Systematically use:
  1. incremental development,
  2. formal specification,
  3. mathematical verification, and
  4. statistical testing
When to Stop?

When are we done testing? When do we have enough tests?

Cynical Answers (sad but true)

- You’re never done: each run of the system is a new test
  - Each bug-fix should be accompanied by a new regression test

- You’re done when you are out of time/money
  - Include testing in the project plan AND DO NOT GIVE IN TO PRESSURE
  - ... in the long run, tests save time

...
When to Stop? ...

Statistical Testing

- Test until you’ve reduced the failure rate to fall below the risk threshold
- Testing is like an insurance company calculating risks

![Graph showing the relationship between execution time and errors per test hour]

**Errors per test hour**

**Execution Time**
What you should know!

- What is the difference between a failure and a fault?
- What kinds of failure classes are important?
- How can a software system be made fault-tolerant?
- How do assertions help to make software more reliable?
- What are the goals of software validation and verification?
- What is the difference between test cases and test data?
- How can you develop test cases for your programs?
- What is the goal of path testing?
Can you answer the following questions?

✎ When would you combine top-down testing with bottom-up testing?
✎ When would you combine black-box testing with white-box testing?
✎ Is it acceptable to deliver a system that is not 100% reliable?
9. Project Management

Overview:
- Risk management
- Scoping and estimation, planning and scheduling
- Dealing with delays
- Staffing, directing, teamwork

Sources:
Recommended Reading

- The Mythical Man-Month, F. Brooks, Addison-Wesley, 1975
- Object Lessons, T. Love, SIGS Books, 1993
- Succeeding with Objects: Decision Frameworks for Project Management, A. Goldberg and K. Rubin, Addison-Wesley, 1995
Why Project Management?

Almost all software products are obtained via projects. (as opposed to manufactured products)

Project Concern = Deliver on time and within budget

Achieve Interdependent & Conflicting Goals

Limited Resources

The Project Team is the primary Resource!
What is Project Management?

Project Management = Plan the work and work the plan

Management Functions

- **Planning**: Estimate and schedule *resources*
- **Organization**: *Who does what*
- **Staffing**: Recruiting and motivating personnel
- **Directing**: Ensure team *acts as a whole*
- **Monitoring (Controlling)**: Detect plan *deviations* + corrective actions
Risk Management

If you don’t actively attack risks, they will actively attack you.

— Tom Gilb

Project risks
- budget, schedule, resources, size, personnel, morale
...

Technical risks
- implementation technology, verification, maintenance
...

Business risks
- market, sales, management, commitment ...

Risk Management

Management must:

- **identify** risks as early as possible
- **assess** whether risks are acceptable
- take appropriate action to **mitigate** and **manage** risks
  - e.g., training, prototyping, iteration, ...
- **monitor** risks throughout the project
## Risk Management Techniques

<table>
<thead>
<tr>
<th>Risk Items</th>
<th>Risk Management Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personnel <em>shortfalls</em></td>
<td>Staffing with top talent; <em>team building</em>; cross-training; pre-scheduling key people</td>
</tr>
<tr>
<td>Unrealistic schedules and budgets</td>
<td>Detailed multi-source cost &amp; schedule estimation; <em>incremental development</em>; reuse; re-scoping</td>
</tr>
<tr>
<td>Developing the <em>wrong</em> software functions</td>
<td>User-surveys; <em>prototyping</em>; early users’s manuals</td>
</tr>
<tr>
<td>Continuing stream of requirements changes</td>
<td>High change threshold; information hiding; <em>incremental development</em></td>
</tr>
<tr>
<td>Risk Items</td>
<td>Risk Management Techniques</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>Real time performance shortfalls</td>
<td>Simulation; benchmarking; modeling; prototyping; <em>instrumentation</em>; tuning</td>
</tr>
<tr>
<td>Straining computer science capabilities</td>
<td>Technical analysis; cost-benefit analysis; <em>prototyping</em>; reference checking</td>
</tr>
</tbody>
</table>
Focus on Scope

For decades, programmers have been whining, “The customers can’t tell us what they want. When we give them what they say they want, they don’t like it.” Get over it. This is an absolute truth of software development. The requirements are never clear at first. Customers can never tell you exactly what they want.

— Kent Beck
Myth: Scope and Objectives

Myth

“A general statement of objectives is enough to start coding.”

Reality

Poor up-front definition is the major cause of project failure.
Scope and Objectives

In order to plan, you must set clear scope & objectives

- **Objectives** identify the general goals of the project, *not how they will be achieved*.
- **Scope** identifies the primary functions that the software is to accomplish, and **bounds** these functions in a quantitative manner.

Goals must be **realistic and measurable**

*Constraints*, performance, reliability must be explicitly stated

*Customer* must set *priorities*
## Estimation Strategies

These strategies are simple but risky:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expert judgement</strong></td>
<td>Consult experts and compare estimates cheap, but unreliable</td>
</tr>
<tr>
<td><strong>Estimation by analogy</strong></td>
<td>Compare with other projects in the same application domain limited applicability</td>
</tr>
<tr>
<td><strong>Parkinson's Law</strong></td>
<td>Work expands to fill the time available pessimistic management strategy</td>
</tr>
<tr>
<td><strong>Pricing to win</strong></td>
<td>You do what you can with the budget available requires trust between parties</td>
</tr>
</tbody>
</table>
**Estimation Techniques**

“Decomposition” and “Algorithmic cost modeling” are used together

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>Estimate costs for <em>components + integration</em></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‣ top-down or bottom-up estimation</td>
</tr>
<tr>
<td>Algorithmic cost modeling</td>
<td>Exploit <em>database</em> of historical facts to map size on costs</td>
</tr>
<tr>
<td></td>
<td>‣ requires correlation data</td>
</tr>
</tbody>
</table>
Measurement-based Estimation

A. Measure
Develop a system model and measure its size

B. Estimate
Determine the effort with respect to an empirical database of measurements from similar projects

C. Interpret
Adapt the effort with respect to a specific development project plan
Estimation and Commitment

Example: The XP process

1. a. Customers write stories and
   b. Programmers estimate stories
      else ask the customers to split/rewrite stories

2. Programmers measure the team load factor, the ratio of ideal programming time to the calendar

3. Customers sort stories by priority

4. Programmers sort stories by risk

5. a. Customers pick date, programmers calculate budget, customers pick stories adding up to that number, or
   b. Customers pick stories, programmers calculate date
      (customers complain, programmers ask to reduce scope, customers complain some more but reduce scope anyway)
Planning and Scheduling

Good planning depends largely on project manager’s intuition and experience!

- **Split** project into *tasks*.
  - Tasks into subtasks etc.
- For each task, *estimate the time*.
  - Define tasks small enough for reliable estimation.
- Significant tasks should end with a *milestone*.
  - **Milestone** = A *verifiable* goal that must be met after task completion
  - Clear unambiguous milestones are a necessity! ("80% coding finished" is a meaningless statement)
  - *Monitor* progress via *milestones*

...
Planning and Scheduling …

- Define dependencies between project tasks
  - Total time depends on longest (= critical) path in activity graph
  - Minimize task dependencies to avoid delays
- Organize tasks concurrently to make optimal use of workforce

Planning is iterative
  ⇒ monitor and revise schedules during the project!
Myth: Deliverables and Milestones

Myth

“The only deliverable for a successful project is the working program.”

Reality

Documentation of all aspects of software development are needed to ensure maintainability.
Deliverables and Milestones

Project **deliverables** are results that are delivered to the customer.

- E.g.:
  - initial requirements document
  - UI prototype
  - architecture specification

- Milestones and deliverables help to **monitor progress**
  
  Should be scheduled roughly every 2-3 weeks

NB: Deliverables must evolve as the project progresses!
Example: Task Durations and Dependencies

<table>
<thead>
<tr>
<th>Task</th>
<th>Duration (days)</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>15</td>
<td>T1</td>
</tr>
<tr>
<td>T4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>10</td>
<td>T2, T4</td>
</tr>
<tr>
<td>T6</td>
<td>5</td>
<td>T1, T2</td>
</tr>
<tr>
<td>T7</td>
<td>20</td>
<td>T1</td>
</tr>
<tr>
<td>T8</td>
<td>25</td>
<td>T4</td>
</tr>
<tr>
<td>T9</td>
<td>15</td>
<td>T3, T6</td>
</tr>
<tr>
<td>T10</td>
<td>15</td>
<td>T5, T7</td>
</tr>
<tr>
<td>T11</td>
<td>7</td>
<td>T9</td>
</tr>
<tr>
<td>T12</td>
<td>10</td>
<td>T11</td>
</tr>
</tbody>
</table>

What is the minimum total duration of this project?
Pert Chart: Activity Network

- Start: 4/7/94
- M1: 25/7/94, 14/7/94
- T3: 4/8/94, 15 days
- T9: 25/8/94, 15 days
- M4: 11/8/94, 10 days
- T10: 5/9/94, 10 days
- T12: 19/9/94, Finish
- M5: 18/7/94
- T4: 10 days
- M2: 25/7/94
- M3: 20 days
- T2: 15 days
- T7: 10 days
- M6: 7 days
- T11: 5/9/94
- M8: 15 days
- T8: 25 days

© Ian Sommerville 1995

© O. Nierstrasz — U. Berne
Project Management
Gantt Chart: Activity Timeline

© Ian Sommerville 1995
## Gantt Chart: Staff Allocation

|          | J  | F  | M  | A  | M  | J  | J  | A  | S  | O  | N  | D  | J  | F  | M  | A  | M  | J  | J  | A  |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Marta    |    |    |    |    |    | 1  | 2  | Design | 3.3. Code Gen. |    | 4. Integrate & Test |    |    |    |    |    |    |    |    |    |
| Leo      |    |    |    |    |    |    |    | 3.3. Code Gen. |    | 4. Integrate & Test |    |    |    |    |    |    |    |    |    |
| Ryan     |    |    |    |    |    |    |    | 3.1 | 3.2. Parser |    | 4. Integrate & Test |    |    |    |    |    |    |    |    |    |
| Sylvia   |    |    |    |    |    |    |    | 3.1 | 3.2. Parser |    | 4. Integrate & Test |    |    |    |    |    |    |    |    |    |
| Laura    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 5. Manual |

(Overall tasks such as reviewing, reporting, ... are difficult to incorporate)
Myth: Delays

Myth

“If we get behind schedule, we can add more programmers and catch up.”

Reality

Adding more people typically **slows a project down**.
Scheduling problems

- *Estimating* the difficulty of problems and the cost of developing a solution *is hard*
- *Productivity is not proportional* to the number of people working on a task
- *Adding people to a late project makes it later* due to communication overhead
- *The unexpected always happens.* Always allow contingency in planning
- *Cutting back in testing and reviewing is a recipe for disaster*
- *Working overnight?* Only short term benefits!
Planning under uncertainty

- State clearly *what you know and don’t know*
- State clearly what you will do to *eliminate unknowns*
- Make sure that all *early milestones can be met*
- Plan to *replan*
Dealing with Delays

Spot potential delays as soon as possible
... then you have more time to recover

How to spot?

❑ Earned value analysis
  ➢ planned time is the project budget
  ➢ time of a completed task is *credited* to the project budget
Dealing with Delays …

How to recover?
A combination of following 3 actions

❑ Adding senior staff for well-specified tasks
  ➫ outside critical path to avoid communication overhead

❑ Prioritize requirements and deliver incrementally
  ➫ deliver most important functionality on time
  ➫ testing remains a priority (even if customer disagrees)

❑ Extend the deadline
Gantt Chart: Slip Line

Visualize slippage

- Shade time line = portion of task completed
- Draw a *slip line* at current date, connecting endpoints of the shaded areas
  - bending to the right = *ahead* of schedule
  - to the left = *behind* schedule

| J | F | M | A | M | J | J | A | S | O | N | D | J | F | M | A | M | J | J | A |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

© O. Nierstrasz — U. Berne
Timeline Chart

Visualise slippage evolution

- downward lines represent planned completion time as they vary in current time
- bullets at the end of a line represent completed tasks

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Actual Time

- 3.1 scanner on time
- 3.2 parser ahead of schedule
- 3.3 code generator behind schedule

Today

Planned Time

- on time
- ahead of schedule
- behind schedule

© O. Nierstrasz — U. Berne

Project Management
## Slip Line vs. Timeline

| Slip Line | Monitors *current slip status* of project tasks  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- <em>many</em> tasks</td>
</tr>
<tr>
<td></td>
<td>- only for <em>1 point in time</em></td>
</tr>
<tr>
<td></td>
<td>- include a few slip lines from the past to illustrate evolution</td>
</tr>
</tbody>
</table>
| Timeline  | Monitors how the slip status of project tasks evolves  
|           | - *few* tasks                                 |
|           | - crossing lines quickly clutter the figure   |
|           | - colours can be used to show more tasks      |
|           | - *complete* time scale                       |
Software Teams

Team organisation

- Teams should be relatively small (< 8 members)
  - minimize communication overhead
  - team quality standard can be developed
  - members can work closely together
  - programs are regarded as team property ("egoless programming")
  - continuity can be maintained if members leave
- Break big projects down into multiple smaller projects
- Small teams may be organised in an informal, democratic way
- Chief programmer teams try to make the most effective use of skills and experience
Chief Programmer Teams

- Consist of a kernel of specialists helped by others as required
- **chief programmer** takes full responsibility for design, programming, testing and installation of system
- **backup programmer** keeps track of CP's work and develops test cases
- **librarian** manages all information
- Others may include: project administrator, toolsmith, documentation editor, language/system expert, tester, and support programmers

...
Chief Programmer Teams …

- Reportedly successful but problems are:
  - **Difficult** to find talented chief programmers
  - **Disrupting** to normal organisational structures
  - **De-motivating** for those who are not chief programmers
Directing Teams

Managers serve their team

- Managers ensure that team has the necessary information and resources

"The manager’s function is not to make people work, it is to make it possible for people to work"

— Tom DeMarco

Responsibility demands authority

- Managers must delegate
  
  Trust your own people and they will trust you.
Directing Teams ...

Managers manage
- Managers *cannot* perform tasks on the *critical path*
  - Especially difficult for technical managers

Developers control deadlines
- A manager cannot meet a deadline to which the *developers* have not *agreed*
What you should know!

- How can prototyping help to reduce risk in a project?
- What are milestones, and why are they important?
- What can you learn from an activity network? An activity timeline?
- What’s the difference between the 0/100; the 50/50 and the milestone technique for calculating the earned value.
- Why should programming teams have no more than about 8 members?
Can you answer these questions?

✎ What will happen if the developers, not the customers, set the project priorities?
✎ What is a good way to measure the size of a project (based on requirements alone)?
✎ When should you sign a contract with the customer?
✎ Would you consider bending slip lines as a good sign or a bad sign? Why?
✎ How would you select and organize the perfect software development team?
10. Software Architecture

Overview:

- What is Software Architecture?
- Coupling and Cohesion
- Architectural styles:
  - Layered, Client-Server, Blackboard, Dataflow, ...
- UML diagrams for architectures
Sources:

- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999
What is Software Architecture?

A neat-looking drawing of some boxes, circles, and lines, laid out nicely in Powerpoint or Word, does not constitute an architecture.

— D’Souza & Wills
What is Software Architecture?

The **architecture** of a system consists of:

- the **structure(s) of its parts**
  - including design-time, test-time, and run-time
    hardware and software parts
- the **externally visible properties** of those parts
  - modules with interfaces, hardware units, objects
- the **relationships and constraints** between them

**in other words:**

- The set of **design decisions** about any system (or subsystem) that keeps its implementors and maintainers from exercising “**needless creativity**”.
How Architecture Drives Implementation

- Use a 3-tier client-server architecture: all business logic must be in the middle tier, presentation and dialogue on the client, and data services on the server; that way you can scale the application server processing independently of persistent store.

- Use Corba for all distribution, using Corba event channels for notification and the Corba relationship service; do not use the Corba messaging service as it is not yet mature.
How Architecture Drives Implementation ...

- Use Collection Galore’s *collections* for representing any collections; by default use their List class, or document your reason otherwise.

- Use *Model-View-Controller* with an explicit *ApplicationModel* object to connect any UI to the business logic and objects.
Sub-systems, Modules and Components

- **A sub-system** is a system in its own right whose operation is independent of the services provided by other sub-systems.

- **A module** is a system component that provides services to other components but would not normally be considered as a separate system.

- **A component** is an independently deliverable unit of software that encapsulates its design and implementation and offers interfaces to the out-side, by which it may be composed with other components to form a larger whole.
Cohesion

Cohesion is a measure of how well the parts of a component “belong together”.

- Cohesion is **weak** if elements are bundled simply because they perform similar or related functions (e.g., java.lang.Math).
- Cohesion is **strong** if all parts are needed for the functioning of other parts (e.g. java.lang.String).
- Strong cohesion promotes maintainability and adaptability by limiting the scope of changes to small numbers of components.

There are many definitions and interpretations of cohesion. **Most attempts to formally define it are inadequate!**
Coupling

**Coupling** is a measure of the *strength of the interconnections* between system components.

- Coupling is *tight* between components if they depend heavily on one another, (e.g., there is a lot of communication between them).
- Coupling is *loose* if there are few dependencies between components.
- Loose coupling *promotes maintainability and adaptability* since *changes in one component are less likely to affect others.*
Tight Coupling

Module A

Module B

Module C

Module D

Shared data area

© Ian Sommerville 1995
Loose Coupling

- Module A
  - A’s data

- Module B
  - B’s data

- Module C
  - C’s data

- Module D
  - D’s data

© Ian Sommerville 1995
Architectural Parallels

- Architects are the *technical interface* between the customer and the contractor building the system.

- A *bad architectural design* for a building *cannot be rescued* by good construction — the same is true for software.

- There are *specialized types* of building and software architects.

- There are *schools* or *styles* of building and software architecture.
**Architectural Styles**

An architectural style defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined.

— Shaw and Garlan
Layered Architectures

A **layered architecture** organises a system into a set of layers each of which *provide a set of services to the layer “above”*.

- Normally layers are *constrained* so elements only see
  - other elements in the same layer, or
  - elements of the layer below
- **Callbacks** may be used to communicate to higher layers
- Supports the *incremental* development of sub-systems in different layers.
  - When a layer interface *changes*, only the adjacent layer is affected
Abstract Machine Model

Version management

Object management

Database system

Operating system

© Ian Sommerville 1995
OSI Reference Model

© Ian Sommerville 1995
Client-Server Architectures

A **client-server architecture** distributes **application logic** and **services** respectively to a number of client and server subsystems, each potentially running on a different machine and communicating through the **network** (e.g., by RPC).

**Advantages**

- **Distribution** of data is straightforward
- Makes effective use of **networked** systems. May require cheaper hardware
- Easy to **add** new servers or **upgrade** existing servers

...
Client-Server Architectures …

Disadvantages

❑ **No shared data model** so sub-systems use different data organisation.

Data *interchange* may be *inefficient*

❑ **Redundant** management in each server

❑ **May require a central registry** of names and services — it may be hard to find out what servers and services are available
Client-Server Architectures

Client 1

Client 2

Client 3

Client 4

Wide-bandwidth network

Catalogue server

Catalogue

Video server

Film clip files

Picture server

Digitized photographs

Hypertext server

Hypertext web

© Ian Sommerville 1995
Four-Tier Architectures

- Clients
  - Browser
    - Non-browser GUI
  - Web server
    - Servlet
- Business objects and components server
  - EJB Container
    - Enterprise JavaBean
  - Legacy System
    - Legacy Adaptor
Blackboard Architectures

A **blackboard architecture** distributes application logic to a number of independent sub-systems, but manages all data in a **single, shared repository** (or “blackboard”).

**Advantages**
- Efficient way to share large amounts of data
- Sub-systems need not be concerned with how data is produced, backed up etc.
- Sharing model is published as the repository schema
Blackboard Architectures ...

Disadvantages

- Sub-systems must *agree* on a repository data model
- Data evolution is *difficult* and *expensive*
- No scope for *specific management policies*
- Difficult to *distribute efficiently*
Repository Model

Design editor

Code generator

Design translator

Project repository

Program editor

Design analyser

Report generator

© Ian Sommerville 1995
Event-driven Systems

In an event-driven architecture components perform services in reaction to external events generated by other components.

- In broadcast models an event is broadcast to all subsystems. Any sub-system which can handle the event may do so.

- In interrupt-driven models real-time interrupts are detected by an interrupt handler and passed to some other component for processing.
**Broadcast model**

- Effective in *integrating* sub-systems on different computers in a network
- Can be implemented using a *publisher-subscriber* pattern:
  - Sub-systems *register* an interest in specific events
  - When these occur, control is transferred to the *subscribed* sub-systems
- Control *policy is not embedded* in the event and message handler. Sub-systems decide on events of interest to them
- However, sub-systems don’t know if or when an event will be handled
Selective Broadcasting

Sub-system 1

Sub-system 2

Sub-system 3

Sub-system 4

Event and message handler

©Ian Sommerville 1995
Dataflow Models

In a dataflow architecture each component performs functional transformations on its inputs to produce outputs.

- Highly effective for reducing latency in parallel or distributed systems
  - No call/reply overhead
  - But, fast processes must wait for slower ones

- Not really suitable for interactive systems
  - Dataflows should be free of cycles

...
Dataflow Models …

Examples:

- The single-input, single-output variant is known as pipes and filters
  - e.g., UNIX (Bourne) shell

<table>
<thead>
<tr>
<th>Data source</th>
<th>Filter</th>
<th>Data sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>tar cf - .</td>
<td>gzip -9</td>
<td>rsh picasso dd</td>
</tr>
</tbody>
</table>

- e.g., CGI Scripts for interactive Web-content

<table>
<thead>
<tr>
<th>Data source</th>
<th>Filter</th>
<th>Data sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML Form</td>
<td>CGI Script</td>
<td>generated HTML page</td>
</tr>
</tbody>
</table>
Invoice Processing System

Read issued invoices → Identify payments

Invoices → Payments

Find payments due → Issue receipts

Issue receipts → Receipts

Issue payment reminder → Reminders

© Ian Sommerville 1995
Compilers as Dataflow Architectures

- Lexical analysis
- Syntactic analysis
- Semantic analysis
- Code generation

Symbol table

© Ian Sommerville 1995
Compilers as Blackboard Architectures

- Lexical analyser
- Syntax analyser
- Semantic analyser
- Abstract syntax tree
- Grammar definition
- Symbol table
- Output definition
- Repository
- Pretty-printer
- Editor
- Optimizer
- Code generator

© Ian Sommerville 1995
UML support: Package Diagram

Decompose system into packages (containing any other UML element, incl. packages)

Application Layer

Processing Orders

Customer Management

Domain Layer

Order

Customer

Database Layer

RDB Interface

query()
**UML support: Deployment Diagram**

*Physical layout* of run-time components on hardware nodes.

- **myMac**: Mac
  - Netscape
- **aPC**: PC
  - IExplorer
- **UnixHost**: WebServer
- **UnixHost**: Database
- «internet»
- «internet»
- «ethernet»
What you should know!

✎ How does software architecture constrain a system?
✎ How does choosing an architecture simplify design?
✎ What are coupling and cohesion?
✎ What is an architectural style?
✎ Why shouldn’t elements in a software layer “see” the layer above?
✎ What kinds of applications are suited to event-driven architectures?
Can you answer the following questions?

✎ What is meant by a “fat client” or a “thin client” in a 4-tier architecture?
✎ What kind of architectural styles are supported by the Java AWT? by RMI?
✎ How do callbacks reduce coupling between software layers?
✎ How would you implement a dataflow architecture in Java?
✎ Is it easier to understand a dataflow architecture or an event-driven one?
✎ What are the coupling and cohesion characteristics of each architectural style?
11. Software Quality

Overview:

- What is quality?
- Quality Attributes
- Quality Assurance: Planning and Reviewing
- Quality System and Standards

Sources:

What is Quality?

**Software Quality** is conformance to:

- explicitly stated *functional and performance requirements*,
- explicitly documented *development standards*,
- *implicit characteristics* that are expected of all professionally developed software.
Problems with Software Quality

- Software specifications are usually *incomplete* and often *inconsistent*.
- There is *tension* between:
  - *customer* quality requirements (efficiency, reliability, etc.)
  - *developer* quality requirements (maintainability, reusability, etc.)
- Some quality requirements are *hard to specify* in an unambiguous way:
  - *directly* measurable qualities (e.g., errors/KLOC),
  - *indirectly* measurable qualities (e.g., usability).

*Quality management is not just about reducing defects!*
Hierarchical Quality Model

Define quality via hierarchical quality model, i.e. number of quality attributes (a.k.a. quality factors, quality aspects, ...)

Choose quality attributes (and weights) depending on the project context.
Quality Attributes

Quality attributes apply both to the product and the process.

- **product**: delivered to the customer
- **process**: produces the software product
- **resources**:
  (both the product and the process require resources)
- Underlying assumption: a quality process leads to a quality product (cf. metaphor of manufacturing lines)
Quality Attributes ...

Quality attributes can be external or internal.

- **External:** Derived from the relationship between the environment and the system (or the process). (To derive, the system or process must run)
  - e.g. Reliability, Robustness

- **Internal:** Derived immediately from the product or process description (To derive, it is sufficient to have the description)
  - Underlying assumption: internal quality leads to external quality (cfr. metaphor manufacturing lines)
  - e.g. Efficiency
Correctness, Reliability, Robustness

Correctness

- A system is **correct** if it behaves according to its specification
  - An **absolute** property (i.e., a system cannot be “almost correct”)
  - ... in theory and practice **undecidable**

Reliability

- The user may rely on the system behaving properly
- **Reliability** is the **probability** that the system will operate as expected over a specified interval
  - A **relative** property (a system has a mean time between failure of 3 weeks)
Correctness, Reliability, Robustness ...

Robustness

- A system is **robust** if it behaves reasonably *even in circumstances that were not specified*
  
  A **vague** property (once you specify the abnormal circumstances they become part of the requirements)
Efficiency, Usability

Efficiency. (Performance)

- **Use of resources** such as computing time, memory
  - Affects user-friendliness and scalability
  - Hardware technology changes fast!
  - (Remember: First do it, then do it right, then do it fast)

- For process, resources are manpower, time and money
  - relates to the “productivity” of a process
Efficiency, Usability …

Usability. (User Friendliness, Human Factors)

- The degree to which the human users find the system (process) both “easy to use” and useful
  - Depends a lot on the target audience (novices vs. experts)
  - Often a system has various kinds of users (end-users, operators, installers)
  - Typically expressed in “amount of time to learn the system”
Maintainability

*external product* attributes (evolvability also applies to process)

**Maintainability**

- How easy it is to *change* a system after its initial release
  - *software entropy* ⇒ maintainability gradually decreases over time
Maintainability ...

Is often refined into ... 

Repairability 
- How much work is needed to *correct* a defect

Evolvability (Adaptability) 
- How much work is needed to *adapt* to changing requirements (both system and process)

Portability 
- How much work is needed to *port* to new environment or platforms
Verifiability, Understandability

internal (and external) product attribute

Verifiability

❑ How easy it is to verify whether desired attributes are there?
  ➣ internally: e.g., verify requirements, code inspections
  ➣ externally: e.g., testing, efficiency

Understandability

❑ How easy it is to understand the system
  ➣ internally: contributes to maintainability
  ➣ externally: contributes to usability
Productivity, Timeliness, Visibility

*external* process attribute (visibility also internal)

**Productivity**

- Amount of product produced by a process for a given number of resources
  - productivity among individuals varies a lot
  - often: productivity \( (\sum \text{individuals}) < \sum \text{productivity (individuals)} \)
Productivity, Timeliness, Visibility ...

Timeliness

- Ability to *deliver the product on time*
  - important for marketing (“short time to market”)
  - often a reason to sacrifice other quality attributes
  - incremental development may provide an answer
Productivity, Timeliness, Visibility ...

Visibility. (Transparency, Glasnost)

- Current process *steps* and project *status* are accessible
  - important for management
  - also deal with staff turn-over
Quality Control Assumption

Project Concern = Deliver on time and within budget

External (and Internal) Product Attributes  Process Attributes

Assumptions:

Internal quality $\Rightarrow$ External quality
Process quality $\Rightarrow$ Product quality

Control during project $\Rightarrow$ Obtain after project

Otherwise, quality is mere coincidence!
The Quality Plan

Plan Time → Schedule
Plan Money → Budget
Plan Quality → Quality Plan

Project Plan
The Quality Plan …

A quality plan should:

❑ set out desired product qualities and how these are assessed
  ➢ define the most significant quality attributes

❑ define the quality assessment process
  ➢ i.e., the controls used to ensure quality

❑ set out which organisational standards should be applied
  ➢ may define new standards, i.e., if new tools or methods are used

NB: Quality Management should be separate from project management to ensure independence
Types of Quality Reviews

A quality review is carried out by a group of people who carefully examine part or all of a software system and its associated documentation.

<table>
<thead>
<tr>
<th>Review type</th>
<th>Principal purpose</th>
</tr>
</thead>
</table>
| **Formal Technical Reviews** (a.k.a. design or program inspections)** | Driven by checklist  
  - detect detailed errors in any product  
  - mismatches between requirements and product  
  - check whether standards have been followed. |
<table>
<thead>
<tr>
<th>Review type</th>
<th>Principal purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progress reviews</td>
<td>Driven by budgets, plans and schedules</td>
</tr>
<tr>
<td></td>
<td>- check whether project runs according to plan</td>
</tr>
<tr>
<td></td>
<td>- requires precise milestones</td>
</tr>
<tr>
<td></td>
<td>- both a process and a product review</td>
</tr>
</tbody>
</table>

- Reviews should be recorded and records maintained
- Software or documents may be “signed off” at a review
- Progress to the next development stage is thereby approved
Review Meetings

Review meetings should:

- typically involve 3-5 people
- require a maximum of 2 hours advance preparation
- last less than 2 hours
Review Minutes

The review report should summarize:

1. **What** was reviewed
2. **Who** reviewed it?
3. **What** were the **findings** and **conclusions**?

The review should conclude whether the product is:

1. **Accepted** without modification
2. **Provisionally accepted**, subject to corrections (no follow-up review)
3. **Rejected**, subject to corrections and follow-up review
Review Guidelines

1. Review the *product*, not the producer
2. Set an *agenda* and maintain it
3. *Limit debate* and rebuttal
4. *Identify problem areas*, but don’t attempt to solve every problem noted
5. Take *written notes*
6. *Limit* the number of participants and insist upon advance preparation
7. Develop a *checklist* for each product that is likely to be reviewed
8. *Allocate resources* and time schedule for reviews
9. Conduct meaningful *training* for all reviewers
10. *Review your early reviews*
Sample Review Checklists (I)

Software Project Planning

1. Is software *scope* unambiguously defined and bounded?
2. Are *resources* adequate for scope?
3. Have *risks* in all important categories been defined?
4. Are *tasks* properly defined and sequenced?
5. Is the basis for *cost estimation* reasonable?
6. Have historical *productivity* and *quality data* been used?
7. Is the *schedule* consistent?

...
Sample Review Checklists (II)

Requirements Analysis

1. Is information *domain analysis* complete, consistent and accurate?
2. Does the *data model* properly reflect data objects, attributes and relationships?
3. Are all *requirements traceable* to system level?
4. Has *prototyping* been conducted for the user/customer?
5. Are requirements *consistent* with schedule, resources and budget?

...
Sample Review Checklists (III)

Design

1. Has **modularity** been achieved?
2. Are **interfaces** defined for modules and external system elements?
3. Are the **data structures consistent** with the **information domain**?
4. Are the **data structures consistent** with the **requirements**?
5. Has **maintainability** been considered?

...
Sample Review Checklists (IV)

Code
1. Does the code reflect the *design* documentation?
2. Has proper use of *language conventions* been made?
3. Have *coding standards* been observed?
4. Are there incorrect or ambiguous *comments*?

...
Sample Review Checklists (V)

Testing
1. Have test resources and tools been identified and acquired?
2. Have both white and black box tests been specified?
3. Have all the independent logic paths been tested?
4. Have test cases been identified and listed with expected results?
5. Are timing and performance to be tested?
Review Results

Comments made during the review should be classified.

☐ No action.
  ➔ No change to the software or documentation is required.

☐ Refer for repair.
  ➔ Designer or programmer should correct an identified fault.

☐ Reconsider overall design.
  ➔ The problem identified in the review impacts other parts of the design.

Requirements and specification errors may have to be referred to the client.
Product and Process Standards

Product standards define characteristics that all components should exhibit.

Process standards define how the software process should be enacted.

<table>
<thead>
<tr>
<th>Product standards</th>
<th>Process standards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design review form</td>
<td>Design review conduct</td>
</tr>
<tr>
<td>Document naming standards</td>
<td>Submission of documents</td>
</tr>
<tr>
<td>Procedure header format</td>
<td>Version release process</td>
</tr>
<tr>
<td>Java conventions</td>
<td>Project plan approval process</td>
</tr>
<tr>
<td>Project plan format</td>
<td>Change control process</td>
</tr>
<tr>
<td>Change request form</td>
<td>Test recording process</td>
</tr>
</tbody>
</table>
Potential Problems with Standards

- Not always seen as *relevant* and up-to-date by software engineers

- May involve too much *bureaucratic* form filling

- May require *tedious* manual work if unsupported by software tools
Sample Java Code Conventions

4.2 Wrapping Lines
When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead.
Sample Java Code Conventions ...

10.3 Constants
Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.
A Quality Plan should be an instance of an organization’s quality system.

- Project Plan
- Quality plan

**Certification**
- Quality Standards (ISO 9001, CMM)

**External Body**
- Accreditation Body

**Audit**
- Feedback & improve
- Instantiates

**Quality Assurance**
- Quality System
- Quality Manual
- Standards & Procedures

Customers may require an externally reviewed quality system.
ISO 9000

ISO 9000 is an international set of standards for quality management applicable to a range of organisations from manufacturing to service industries.

ISO 9001 is a generic model of the quality process, applicable to organisations whose business processes range all the way from design and development, to production, installation and servicing;

- ISO 9001 must be instantiated for each organisation
- ISO 9000-3 interprets ISO 9001 for the software developer

ISO = International Organisation for Standardization

- ISO main site: http://www.iso.ch/
- ISO 9000 main site: http://www.tc176.org/
Capability Maturity Model (CMM)

The SEI process maturity model classifies how well contractors *manage software processes*

- **Level 1: Initial (Ad Hoc)**
  - No effective QA procedures, quality is luck

- **Level 2: Repeatable**
  - Formal QA procedures in place

- **Level 3: Defined**
  - QA process is defined and institutionalized

- **Level 4: Managed**
  - QA Process + quantitative data collection

- **Level 5: Optimizing**
  - Improvement is fed back into QA process

*Quantitative data are necessary for improvement!*

*Quality depends on individual project managers!*
What you should know!

- Can a correctly functioning piece of software still have poor quality?
- What’s the difference between an external and an internal quality attribute?
- And between a product and a process attribute?
- Why should quality management be separate from project management?
- How should you organize and run a review meeting?
- What information should be recorded in the review minutes?
Can you answer the following questions?

- Why does a project need a quality plan?
- Why are coding standards important?
- What would you include in a documentation review checklist?
- How often should reviews be scheduled?
- Would you trust software developed by an ISO 9000 certified company?
- And if it were CMM level 5?
12. Software Metrics

Overview:
- What are metrics? Why do we need them?
- Metrics for cost estimation
- Metrics for software quality evaluation

Sources:
Why Metrics?

When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science.

— Lord Kelvin
# Why Measure Software?

<table>
<thead>
<tr>
<th>Why Measure Software?</th>
<th>Measure focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimate cost and effort</td>
<td>measure correlation between specifications and final product</td>
</tr>
<tr>
<td>Improve productivity</td>
<td>measure value and cost of software</td>
</tr>
<tr>
<td>Improve software quality</td>
<td>measure usability, efficiency, maintainability ...</td>
</tr>
<tr>
<td>Improve reliability</td>
<td>measure mean time to failure, etc.</td>
</tr>
<tr>
<td>Evaluate methods and tools</td>
<td>measure productivity, quality, reliability ...</td>
</tr>
</tbody>
</table>

“You cannot control what you cannot measure” — De Marco, 1982

“What is not measurable, make measurable” — Galileo
What are Software Metrics?

Software metrics

- Any type of measurement which relates to a software system, process or related documentation
- Lines of code in a program
- the Fog index (calculates readability of a piece of documentation)
  \[ 0.4 \times \left( \frac{\text{# words}}{\text{# sentences}} \right) + \left( \frac{\text{percentage of words } \geq 3 \text{ syllables}}{} \right) \]
- number of person-days required to implement a use-case

NB: “Software metrics” are not mathematical metrics, but rather measures
(Measures vs Metrics)

Mathematically, a **metric** is a function $m$ measuring the **distance** between two objects such that:

1. $\forall x, m(x,x) = 0$
2. $\forall x, y, m(x,y) = m(y,x)$
3. $\forall x, y, z, m(x,z) \leq m(x,y) + m(y,z)$

So, technically “software metrics” is an abuse of terminology, and we should instead talk about “software measures”.
Direct and Indirect Measures

Direct Measures
- *Measured* directly in terms of the observed attribute (usually by counting)
  - Length of source-code, Duration of process, Number of defects discovered

Indirect Measures
- *Calculated* from other direct and indirect measures
  - Module Defect Density = Number of defects discovered / Length of source
  - Temperature is usually derived from the length of a liquid column
Measurement Mapping

Measure & Measurement

A measure is a function mapping
- an attribute of a real world entity (= the domain)
onto
- a symbol in a set with known mathematical relations (= the range).

A measurement is then the symbol assigned to the real world attribute by the measure.

Purpose: Manipulate symbol(s) in the range to draw conclusions about attribute(s) in the domain.

Example: measure mapping "height" attribute of person on a number representing "height in meters".
Preciseness

To be precise, the definition of the measure must specify:

- **domain**: do we measure people’s height or width?
- **range**: do we measure height in centimetres or inches?
- **mapping rules**: do we allow shoes to be worn?
### Possible Problems

**Compare productivity in lines of code per time unit.**

<table>
<thead>
<tr>
<th>Question</th>
<th>Considerations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do we use the <strong>same units</strong> to compare?</td>
<td>What is a “line of code”? What is the “time unit”?</td>
</tr>
<tr>
<td>Is the <strong>context</strong> the same?</td>
<td>Were programmers familiar with the language?</td>
</tr>
<tr>
<td>Is “code size” really what we want to produce?</td>
<td>What about code <strong>quality</strong>?</td>
</tr>
<tr>
<td>How do we want to <strong>interpret</strong> results?</td>
<td>Average productivity of a programmer? Programmer X is twice as productive as Y?</td>
</tr>
<tr>
<td>What do we want to <strong>do with</strong> the results?</td>
<td>Do you reward “productive” programmers? Do you compare productivity of software processes?</td>
</tr>
</tbody>
</table>
GQM

Goal - Question - Metrics approach. [Basili et al. 1984]

- Define Goal
  - e.g., “How effective is the coding standard XYZ?”

- Break down into Questions
  - “Who is using XYZ?”
  - “What is productivity/quality with/without XYZ?”

- Pick suitable Metrics
  - Proportion of developers using XYZ
  - Their experience with XYZ ...
  - Resulting code size, complexity, robustness ...
Cost estimation objectives

Cost estimation and planning/scheduling are closely related activities

Goals
- To establish a budget for a software project
- To provide a means of controlling project costs
- To monitor progress against the budget
  - comparing planned with estimated costs
- To establish a cost database for future estimation
### Estimation techniques

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expert judgement</td>
<td>cheap, but risky!</td>
</tr>
<tr>
<td>Estimation by analogy</td>
<td>limited applicability</td>
</tr>
<tr>
<td>Parkinson's Law</td>
<td>unlimited risk!</td>
</tr>
<tr>
<td>Pricing to win</td>
<td>i.e., you do what you can with the money</td>
</tr>
<tr>
<td>Top-down estimation</td>
<td>may miss low-level problems</td>
</tr>
<tr>
<td>Bottom-up estimation</td>
<td>may underestimate integration costs</td>
</tr>
<tr>
<td>Algorithmic cost modelling</td>
<td>requires correlation data</td>
</tr>
</tbody>
</table>

Each method has strengths and weaknesses!

Estimation should be based on *several* methods
Algorithmic cost modelling

- Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers.

- The function is derived from a study of historical costing data.

- Most commonly used product attribute for cost estimation is LOC (code size).

- Most models are basically similar but with different attribute values.
Measurement-based estimation

A. Measure
Develop a system model and measure its size

B. Estimate
Determine the effort with respect to an empirical database of measurements from similar projects

C. Interpret
Adapt the effort with respect to a specific development project plan
Lines of code

Lines of Code as a measure of system size?

- *Easy to measure; but not well-defined* for modern languages
  - What's a line of code?

- *A poor indicator* of productivity
  - *Ignores software reuse*, code duplication, benefits of redesign
  - The lower level the language, the more productive the programmer!
  - The more verbose the programmer, the higher the productivity!
Function points

Function Points (Albrecht, 1979)

- Based on a combination of program characteristics:
  - external inputs and outputs
  - user interactions
  - external interfaces
  - files used by the system
- A weight is associated with each of these
- The function point count is computed by multiplying each raw count by the weight and summing all values
- Function point count modified by complexity of the project
Function points

Good points, bad points

❑ Can be measured already after design
❑ FPs can be used to estimate LOC depending on the average number of LOC per FP for a given language
❑ LOC can vary wildly in relation to FP
❑ FPs are very subjective — depend on the estimator. They cannot be counted automatically
Programmer productivity

A measure of the rate at which individual engineers involved in software development produce software and associated documentation

Productivity metrics

- Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.
- Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure

Productivity estimates

- Real-time embedded systems, 40-160 LOC/P-month
- Systems programs, 150-400 LOC/P-month
- Commercial applications, 200-800 LOC/P-month
Quality and productivity

- All metrics based on volume/unit time are flawed because they do not take quality into account
- Productivity may generally be increased at the cost of quality
- It is not clear how productivity/quality metrics are related
The COCOMO model

- Developed at TRW, a US defence contractor
- Based on a cost database of more than 60 different projects
- Exists in three stages
  - Basic — Gives a 'ball-park' estimate based on product attributes
  - Intermediate — modifies basic estimate using project and process attributes
  - Advanced — Estimates project phases and parts separately
Basic COCOMO Formula

\[ \text{Effort} = C \times \text{PM}^S \times M \]

- C is a complexity factor
- PM is a product metric (size or functionality)
- Exponent S is close to 1, but increasing for large projects
- M is a multiplier based on process, product and development attributes (~1)
# COCOMO Project classes

**Organic mode:** small teams, familiar environment, well-understood applications, no difficult non-functional requirements (EASY)

\[
\text{Effort} = 2.4 \times (\text{KDSI})^{1.05} \times M
\]

**Semi-detached mode:** Project team may have experience mixture, system may have more significant non-functional constraints, organization may have less familiarity with application (HARDER)

\[
\text{Effort} = 3 \times (\text{KDSI})^{1.12} \times M
\]

**Embedded:** Hardware/software systems, tight constraints, unusual for team to have deep application experience (HARD)

\[
\text{Effort} = 3.6 \times (\text{KDSI})^{1.2} \times M
\]

KDSI = Kilo Delivered Source Instructions
COCOMO assumptions and problems

- Implicit productivity estimate
  - Organic mode = 16 LOC/day
  - Embedded mode = 4 LOC/day
- Time required is a function of total effort not team size
- Not clear how to adapt model to personnel availability

Copyright © 2002 United Feature Syndicate, Inc.
**COCOMO assumptions and problems ...**

- *Staff required* can’t be computed by dividing the development time by the required schedule.
- The *number of people* working on a project varies depending on the phase of the project.
- The *more people* who work on the project, the *more total effort* is usually required (!).
- Very *rapid build-up* of people often correlates with schedule slippage.
Quantitative Quality Model

Quality according to ISO 9126 standard

- Divide-and-conquer approach via “hierarchical quality model”
- Leaves are simple metrics, measuring basic attributes

```
<table>
<thead>
<tr>
<th>ISO 9126</th>
<th>Factor</th>
<th>Characteristic</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software Quality</td>
<td>Functionality</td>
<td>Error tolerance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reliability</td>
<td>Accuracy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Efficiency</td>
<td>Consistency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Usability</td>
<td>Simplicity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Maintainability</td>
<td>Modularity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Portability</td>
<td>defect density</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>correction time</td>
</tr>
</tbody>
</table>
```
"Define your own" Quality Model

Define the quality model with the development team

- Team chooses the characteristics, design principles, metrics ... and the thresholds

<table>
<thead>
<tr>
<th>Factor</th>
<th>Characteristic</th>
<th>Design Principle</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>Modularity</td>
<td>design class as an abstract data-type</td>
<td>number of private attributes ]2, 10[</td>
</tr>
<tr>
<td></td>
<td></td>
<td>encapsulate all attributes</td>
<td>number of public attributes ]0, 0[</td>
</tr>
<tr>
<td></td>
<td></td>
<td>avoid complex interfaces</td>
<td>number of public methods ]5, 30[</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>average number of arguments [0, 4[</td>
</tr>
</tbody>
</table>
Sample Size (and Inheritance) Metrics

**Inheritance Metrics**
- hierarchy nesting level (HNL)
- # immediate children (NOC)
- # inherited methods, unmodified (NMI)
- # overridden methods (NMO)

**Class Size Metrics**
- # methods (NOM)
- # attributes, instance/class (NIA, NCA)
- # $\Sigma$ of method size (WMC)

**Method Size Metrics**
- # invocations (NOI)
- # statements (NOS)
- # lines of code (LOC)
- # arguments (NOA)
Sample Coupling & Cohesion Metrics

Following definitions stem from [Chid91a], later republished as [Chid94a]

**Coupling Between Objects (CBO)**
CBO = number of other class to which given class is coupled
Interpret as “number of other classes a class requires to compile”

**Lack of Cohesion in Methods (LCOM)**
LCOM = number of disjoint sets (= not accessing same attribute) of local methods
Coupling & Cohesion Metrics

Beware!
Researchers disagree whether coupling/cohesion methods are valid

- Classes that are observed to be cohesive may have a high LCOM value
  - due to accessor methods
- Classes that are not much coupled may have high CBO value
  - no distinction between data, method or inheritance coupling
Sample Quality Metrics (I)

Productivity (Process Metric)

- functionality / time
- functionality in LOC or FP; time in hours, weeks, months
  - be careful to compare: the same unit does not always represent the same
- Does not take into account the quality of the functionality!
Sample Quality Metrics (II)

Reliability (Product Metric)

- **mean time to failure** = mean of probability density function PDF
  - for software one must take into account the fact that repairs will influence the rest of the function \(\Rightarrow\) quite complicated formulas

- **average time between failures** = \# failures / time
  - time in execution time or calendar time
  - necessary to calibrate the probability density function

- **mean time between failure** = MTTF + mean time to repair
  - to know when your system will be available, take into account repair
Sample Quality Metrics (III)

Correctness (Product Metric)

- “a system is correct or not, so one cannot measure correctness”
- *defect density* = # known defects / product size
  - product size in LOC or FP
  - # known defects is a time based count!
- do *not* compare across projects unless your data collection is sound!
Sample Quality Metrics (IV)

Maintainability (Product Metric)

- #time to repair certain categories of changes
- “mean time to repair” vs. “average time to repair”
  - similar to “mean time to failure” and “average time between failures”
- beware of the units
  - “categories of changes” is subjective
  - time =?
- problem recognition time + administrative delay time + problem analysis time + change time + testing & reviewing time
What you should know!

- What is a measure? What is a metric?
- What is GQM?
- What are the three phases of algorithmic cost modelling?
- What problems arise when using LOC as a software metric?
- What are the key ideas behind COCOMO?
- What’s the difference between “Mean time to failure” and “Average time between failures”? Why is the difference important?
Can you answer the following questions?

- During which **phases** in a software project would you use metrics?
- Is the Fog index a “good” metric?
- How would you measure your own software **productivity**?
- Why are **coupling/cohesion** metrics important? Why then are they so rarely used?
13. TBA ...