ESE

Einführung in Software Engineering

Prof. O. Nierstrasz
Dr. Serge Demeyer

Wintersemester 2000/2001

Table Of Contents

1. ESE — Einführung in Software Engineering

Lecturer:	Matthias Rieger
	Schützenmattstr. 14/206, Tel.631.3547
	`rieger@iam.unibe.ch`
Secretary:	Frau Th. Schmid, Tel.631.4692
Assistants:	Michele Lanza, Tel. 631.4868
	David Jud, Michael Locher
WWW:	http://www.iam.unibe.ch/~scg/Teaching/ESE

Principle Texts:

- [Somm96a] *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996. *=> Sixth Edition out in summer 2000*
- [Pres97a] *Software Engineering — A Practitioner's Approach*, R. Pressman, Mc-Graw Hill, Fourth Edn., 1997.
- [Wirf90a] *Designing Object-Oriented Software*, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990.
- [Ghez91a] *Fundamentals of Software Engineering*, C. Ghezzi, M. Jazayeri, D. Mandroli, Prentice Hall, 1991.

Other Books

- *The Mythical Man-Month*, F. Brooks, Addison-Wesley, Anniversary Edition 1995.
- *Object Lessons — Lessons Learned in Object-Oriented Development Projects*, T. Love, SIGS Books, 1993
- *Object-Oriented Development — The Fusion Method*, D. Coleman, et al., Prentice Hall, 1994.
- *Succeeding with Objects: Decision Frameworks for Project Management*, A. Goldberg and K. Rubin, Addison-Wesley, 1995
- *A Discipline for Software Engineering*, W. Humphrey, Addison Wesley, 1995
- *Object-Oriented Software Construction*, B. Meyer, Prentice Hall, Second Edn., 1997.
- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999
- *UML Distilled,* M. Fowler with K. Scott, Addison Wesley, Second Edition, 2000
- *UML@Work*, M. Hitz, G. Kappel, DPunkt, 1999

Course Overview

1.	10-25-2000	Introduction — The Software Lifecycle
2.	11-01-2000	Project Management
3.	11-08-2000	Requirements Collection
4.	11-15-2000	Responsibility-Driven Design
5.	11-22-2000	Detailed Design
6.	12-29-2000	Modeling Objects and Classes
7.	12-06-2000	Modeling Behaviour
8.	12-13-2000	Software Architecture
9.	12-20-2000	User Interface Design
10.	01-10-2001	Software Validation
11.	01-17-2001	Software Quality
12.	01-24-2001	Software Metrics
13.	31-01-2001	Outlook: Heavy vs. Light Methods
	02-07-2001	*Final exam*

Why Software Engineering?

A naive view:

Problem Specification —*coding*→ Final Program

But ...

- Where did the specification come from?
- How do you know the specification correspond 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?

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

"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

"software engineering is different from other engineering disciplines" ([Somm96a])

- ❑ Not constrained by physical laws
  - ☞ limit = human mind
- ❑ It is constrained by political forces
  - ☞ balancing stake-holders

Software Development Activities

Requirements Collection

- Establish customer's needs

Analysis

- Model and specify the requirements ("what")

Design

- Model and specify a solution ("how")

Implementation

- Construct a solution in software

Testing

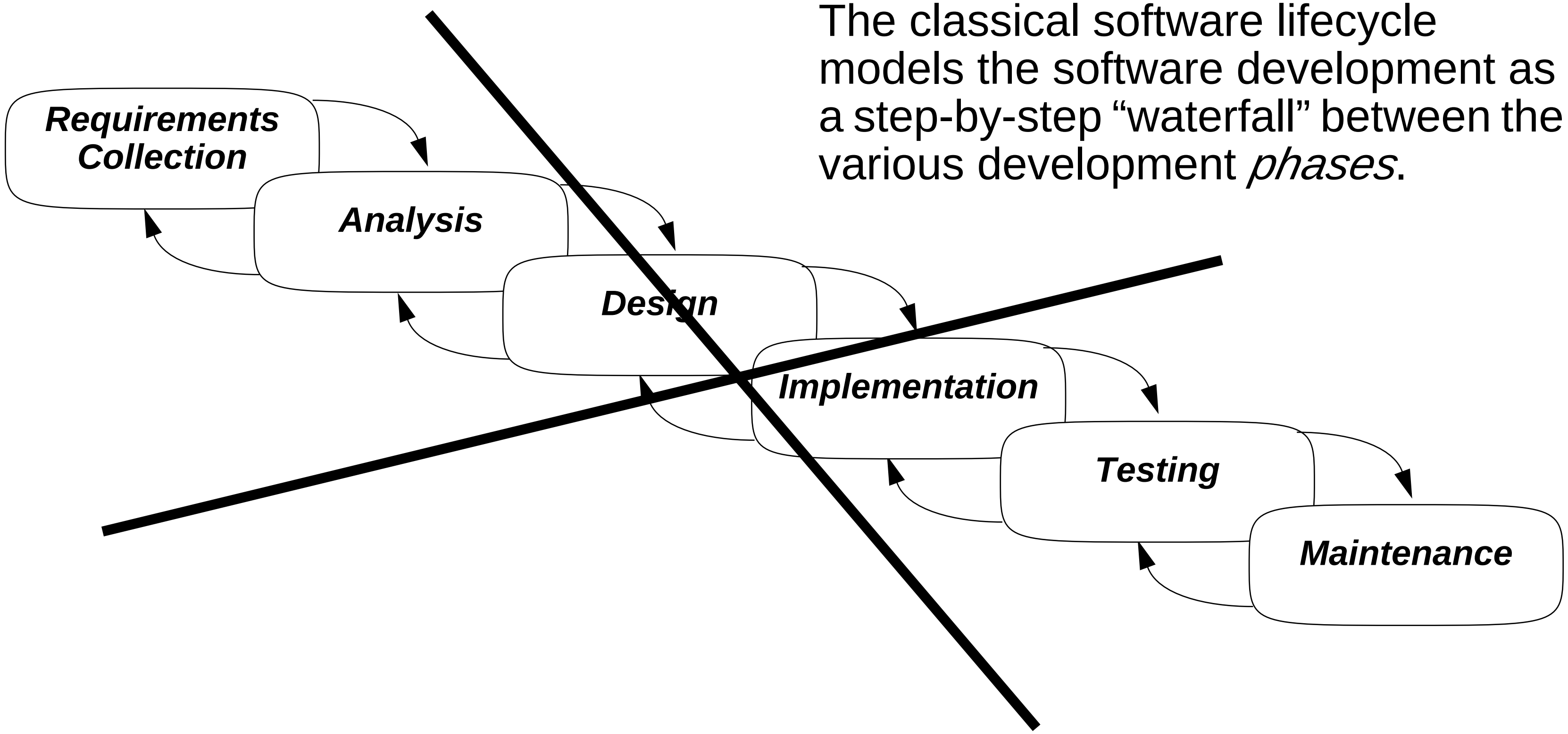
- Validate the solution against the requirements

Maintenance

- Repair defects and adapt the solution to new requirements

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*.

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”

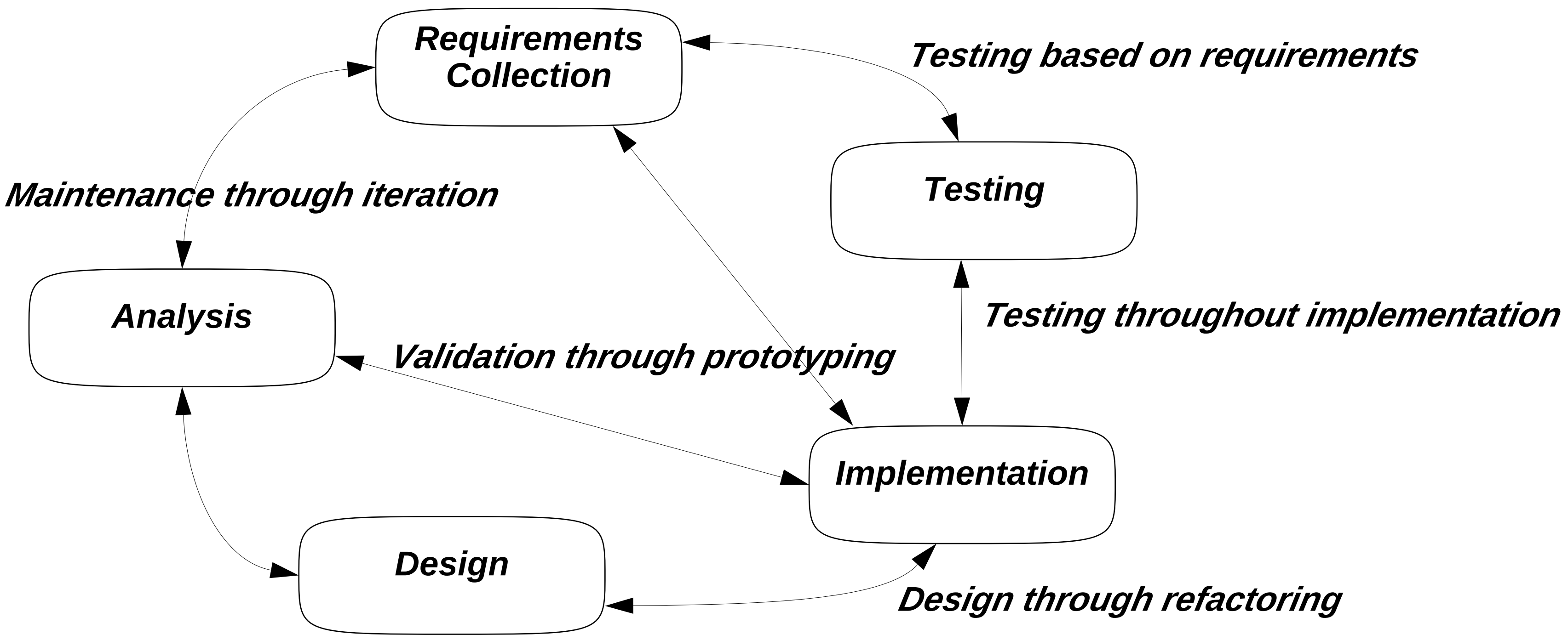
2. “It is often *difficult* for the customer *to state all requirements* explicitly. The classic life cycle requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.”

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.

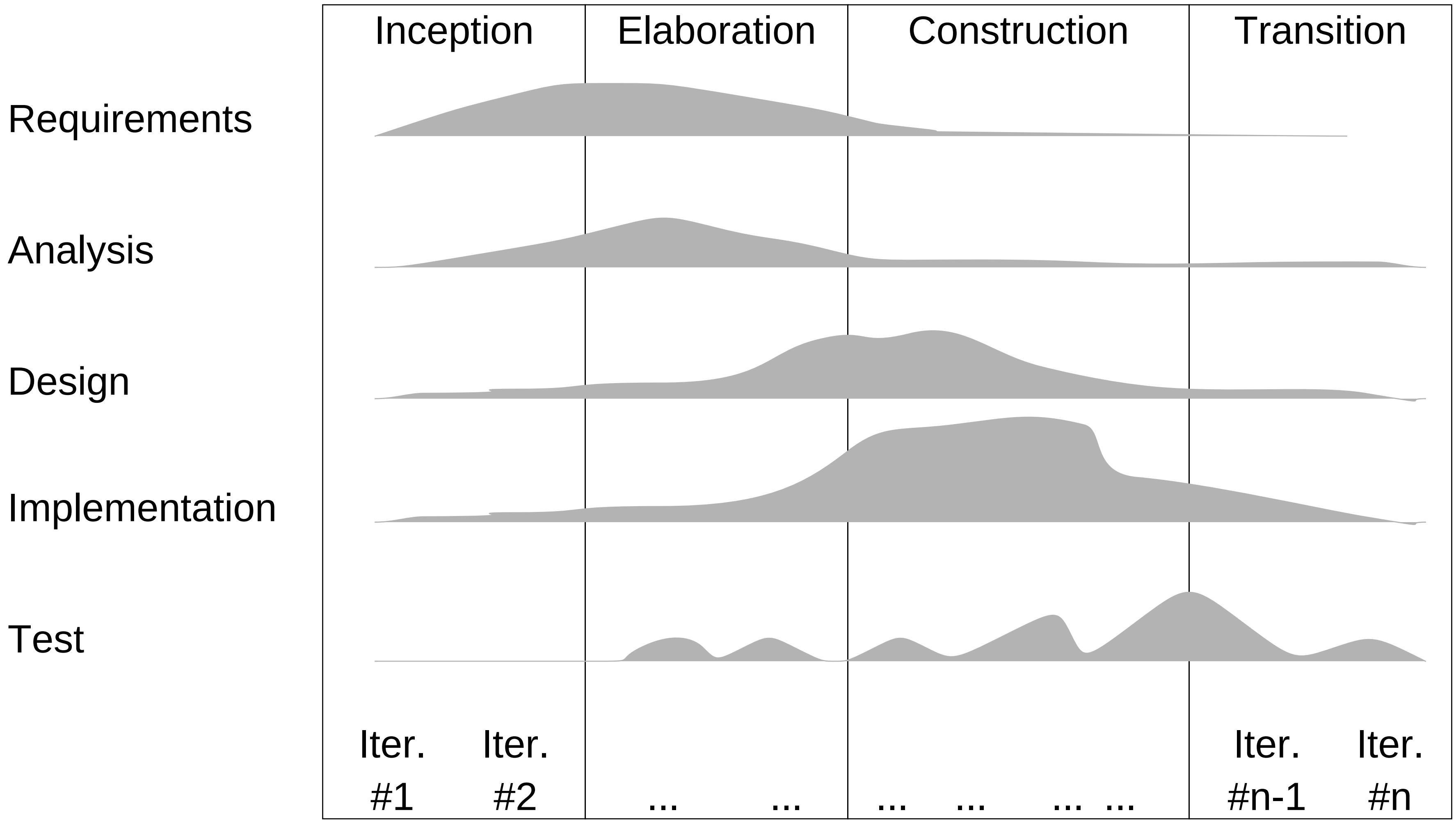
- ☞ You won't get it right the first time, so integrate, validate and test as frequently as possible.

The later in the lifecycle errors are discovered, the more expensive they are to fix!

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

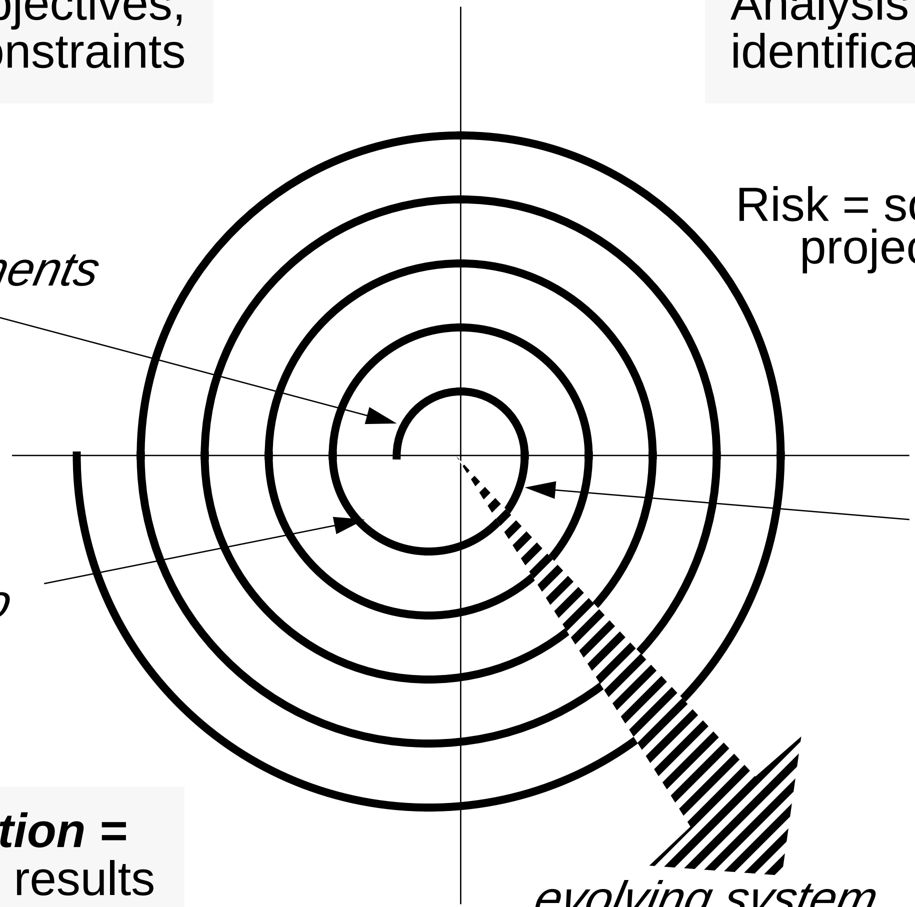


How do you plan the number of iterations? How do you decide on completion?

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

Customer Evaluation =
Assessment of the results of engineering

Engineering =
Development of the "next level" product

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.

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*.

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

- ❑ *classes* of objects that exist in the system
- ❑ *relationships* between those classes
- ❑ *use cases* and *scenarios* describing
  - ☞ *operations* that can be performed on the system
  - ☞ allowable *sequences* of those operations

Does the requirements specification correspond to the users' actual needs?

Prototyping

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

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 of, and operations, on classes

Design is an iterative process, proceeding in parallel with implementation!

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 are iterative activities
  - ☞ The implementation does not "implement the design", but rather the design document *documents the implementation*!

- System tests reflect the requirements specification
- Ideally, test case specification *precedes* design and implementation
  - ☞ Repeatable, automated tests *enable evolution* and *refactoring*

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"!

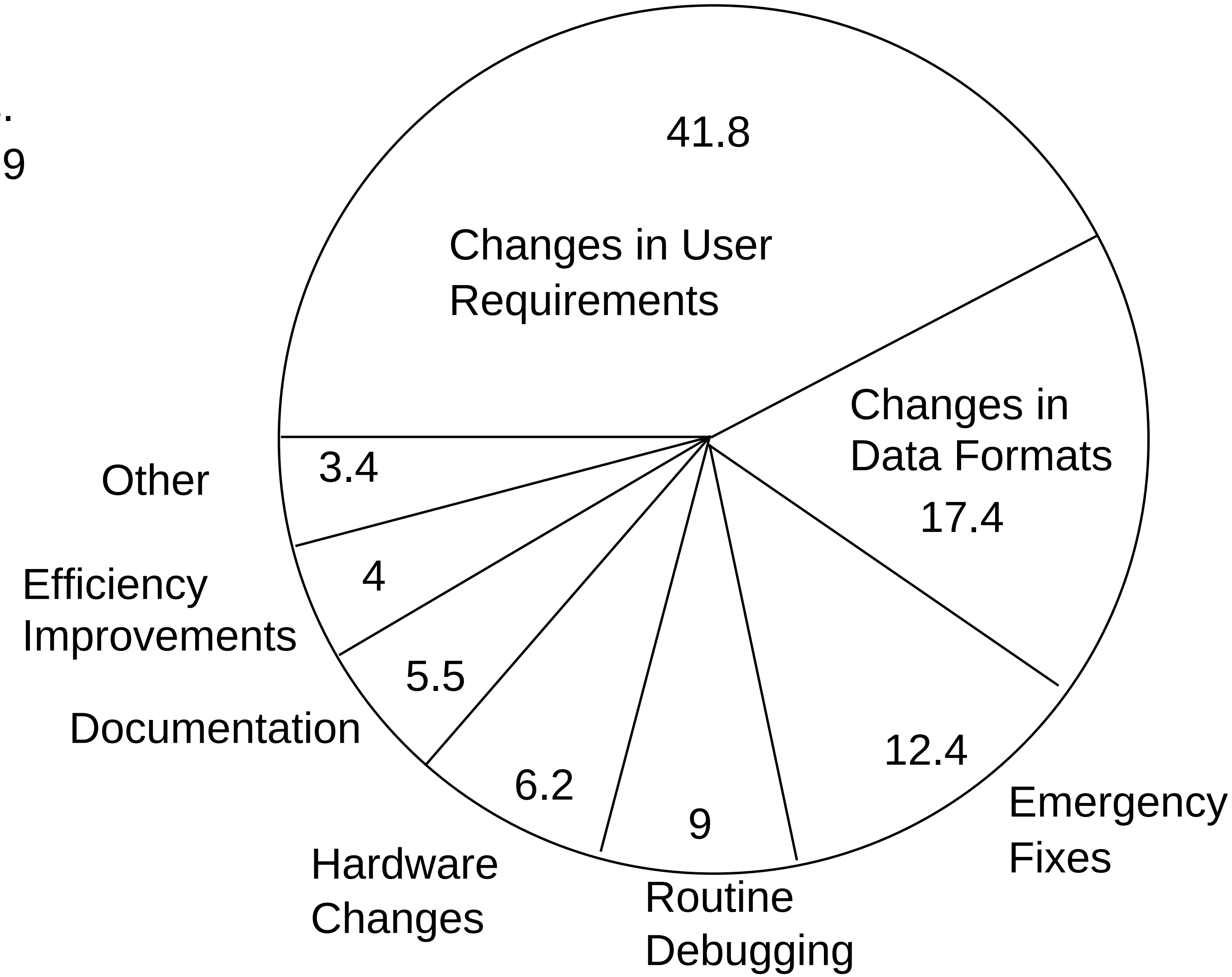
"Maintenance" entails:

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

Maintenance

Breakdown of maintenance costs.

Source: Lientz 1979



Methods and Methodologies

Principle

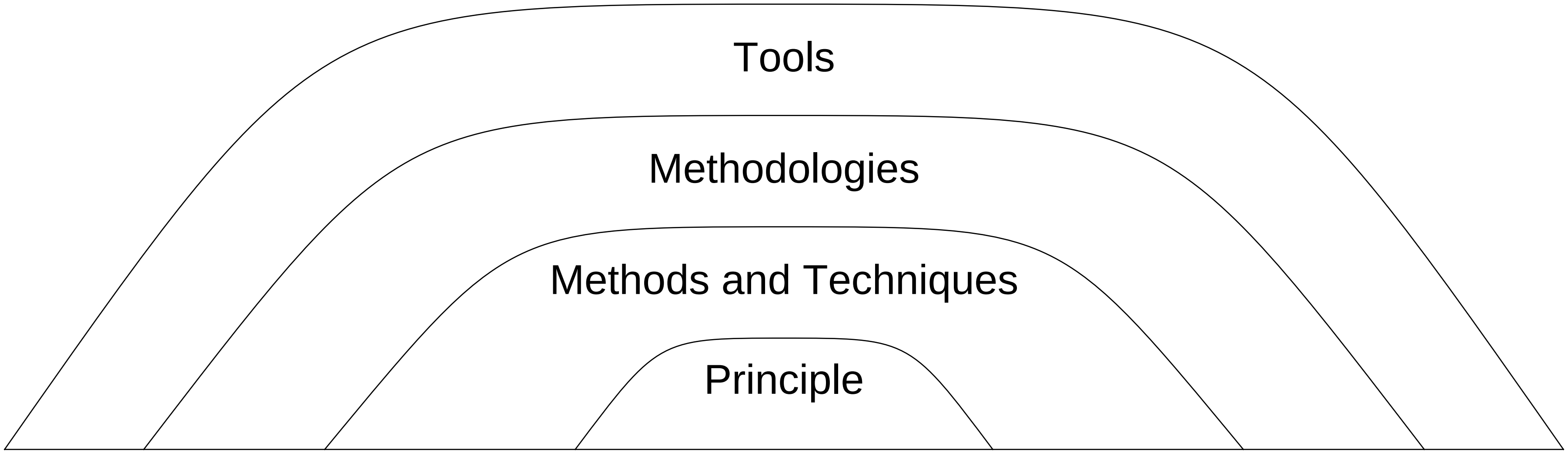
- = general and abstract statements describing desirable properties

Method and Technique

- method = general guidelines that govern the execution of some activity
- technique = more technical and mechanical than method

Methodology

- = set of methods and techniques packaged together



Relationship between principles, techniques, methodologies and tools [Ghez91a]

Why use a Method?

Requirements checking:

- ❑ System Modeling helps uncover omissions and ambiguities in requirements

Clearer concepts:

- ❑ Domain analysis models can be reused/adapted when requirements change

Less design rework:

- ❑ Analysis and design models allow alternatives to be studied before implementation starts

Better refactoring of design work:

- ❑ Analysis and design helps to decompose large systems into manageable parts

Improved communications between developers:

- ❑ Standard notations provide a common vocabulary for analysis and design

Less effort needed on maintenance:

- ❑ Analysis and design documents help maintainers understand complex systems

Object-Oriented Methods: A 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 + ...
  - ☞ complete lifecycle support (to be defined!)

Object-oriented methods are still maturing. Notations are converging, but:

- ☞ *transition* is still risky
- ☞ few methods deal seriously with software *reuse*.

Summary

You should know the answers to these questions:

- ❑ 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 the following 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?*

2. Project Management

Overview:

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

Sources:

- *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.
- *Software Engineering — A Practitioner's Approach*, R. Pressman, Mc-Graw Hill, Third Edn., 1994.

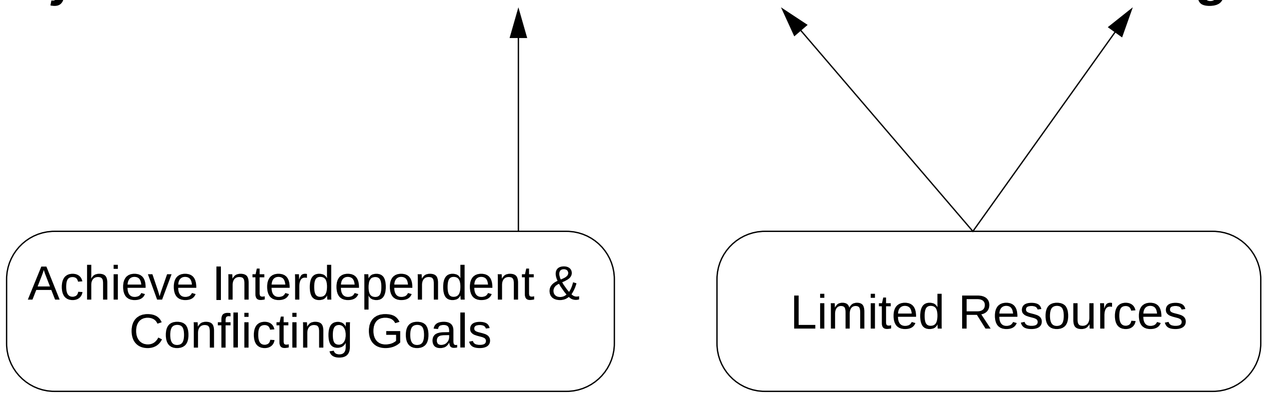
Recommended Reading:

- *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
- *Extreme Programming Explained: Embrace Change*, Kent Beck, Addison Wesley, 1999

Why Project Management?

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

Project Concern= Deliver on time and within budget



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 ...

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

Risk Items	*Risk Management Techniques*
Personnel shortfalls	Staffing with top talent; team building; cross-training; pre-scheduling key people
Unrealistic schedules and budgets	Detailed multisource cost & schedule estimation; incremental development; reuse; re-scoping
Developing the wrong software functions	User-surveys; prototyping; early users's manuals
Continuing stream of requirements changes	High change threshold; information hiding; incremental development
Real time performance shortfalls	Simulation; benchmarking; Modeling; prototyping; instrumentation; tuning
Straining computer science capabilities	Technical analysis; cost-benefit analysis; prototyping; reference checking

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

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.

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*

Effort Estimation

Estimation Strategies

- ❑ Expert judgement — cheap, but unreliable
  - ☞ Consult experts and compare estimates
- ❑ Estimation by analogy — limited applicability
  - ☞ Compare with other projects in the same application domain
- ❑ Parkinson's Law — pessimistic management strategy
  - ☞ Work expands to fill the time available
- ❑ Pricing to win — requires trust between parties
  - ☞ You do what you can with the budget available

Estimation Techniques

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

- ❑ Decomposition — top-down or bottom-up estimation
  - ☞ Estimate costs for components + integrating costs ...
- ❑ Algorithmic cost modeling — requires correlation data
  - ☞ Exploit database of historical facts to map size on costs

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*

   ☞ if they can’t, they ask the customers to split/merge/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 suggest customers reduce scope, customers complain some more but reduce scope anyway)

Planning and Scheduling

Good planning depends a lot 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
- 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!

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.

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

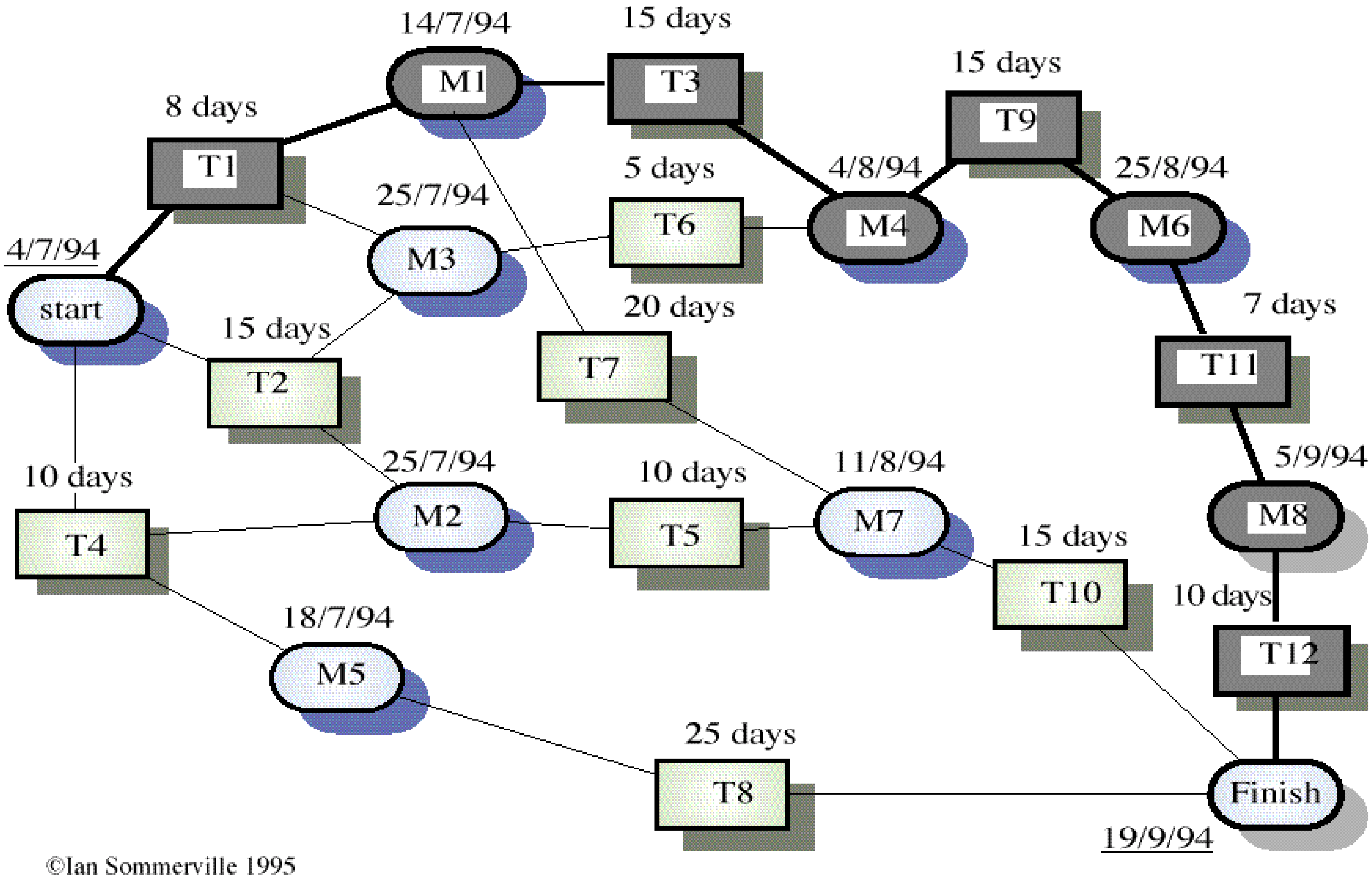
NB: Deliverables must evolve as the project progresses!

Example: Task Durations and Dependencies

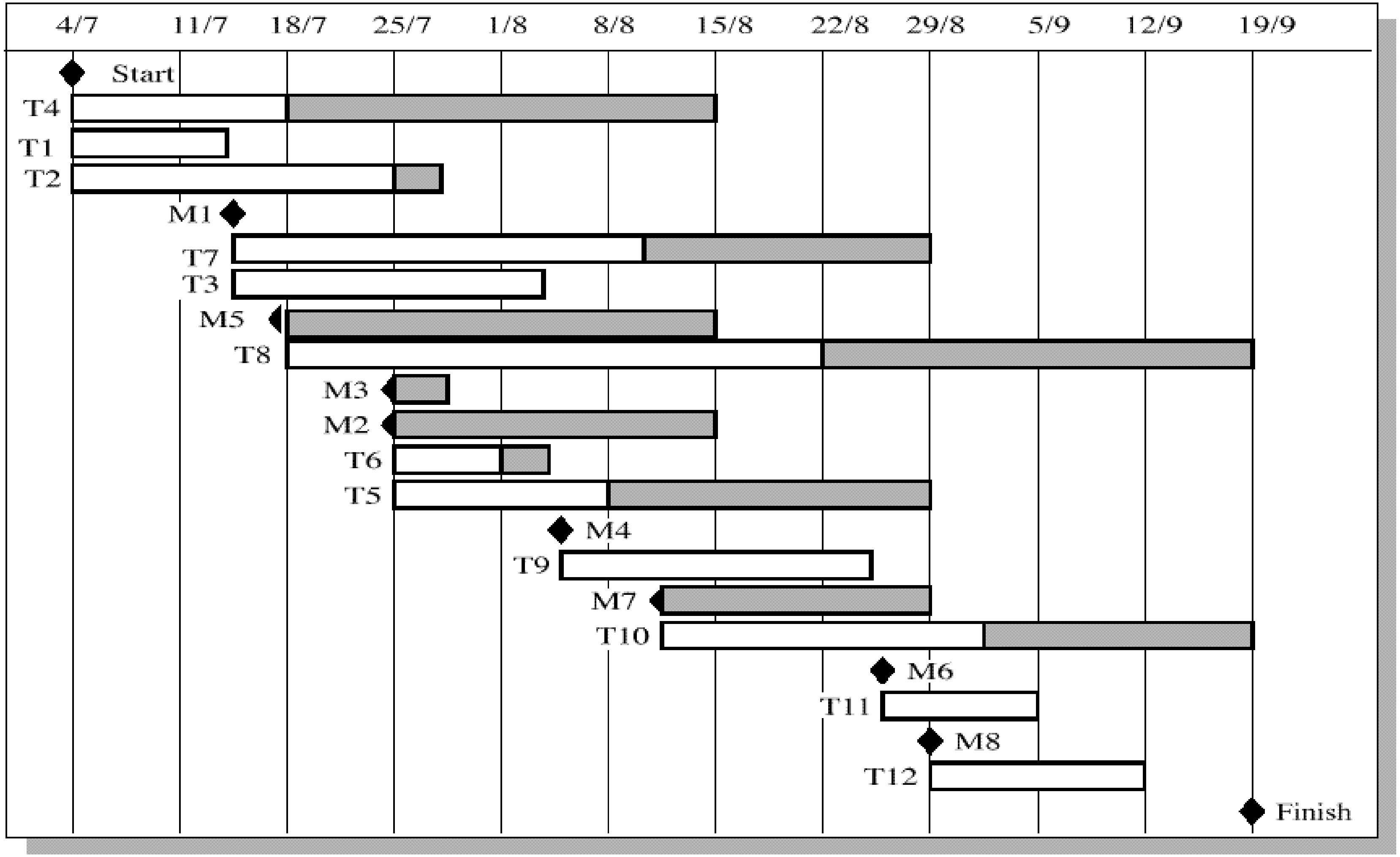
Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1
T4	10	
T5	10	T2, T4
T6	5	T1, T2
T7	20	T1
T8	25	T4
T9	15	T3, T6
T10	15	T5, T7
T11	7	T9
T12	10	T11

✎ *What is the minimum total duration of this project?*

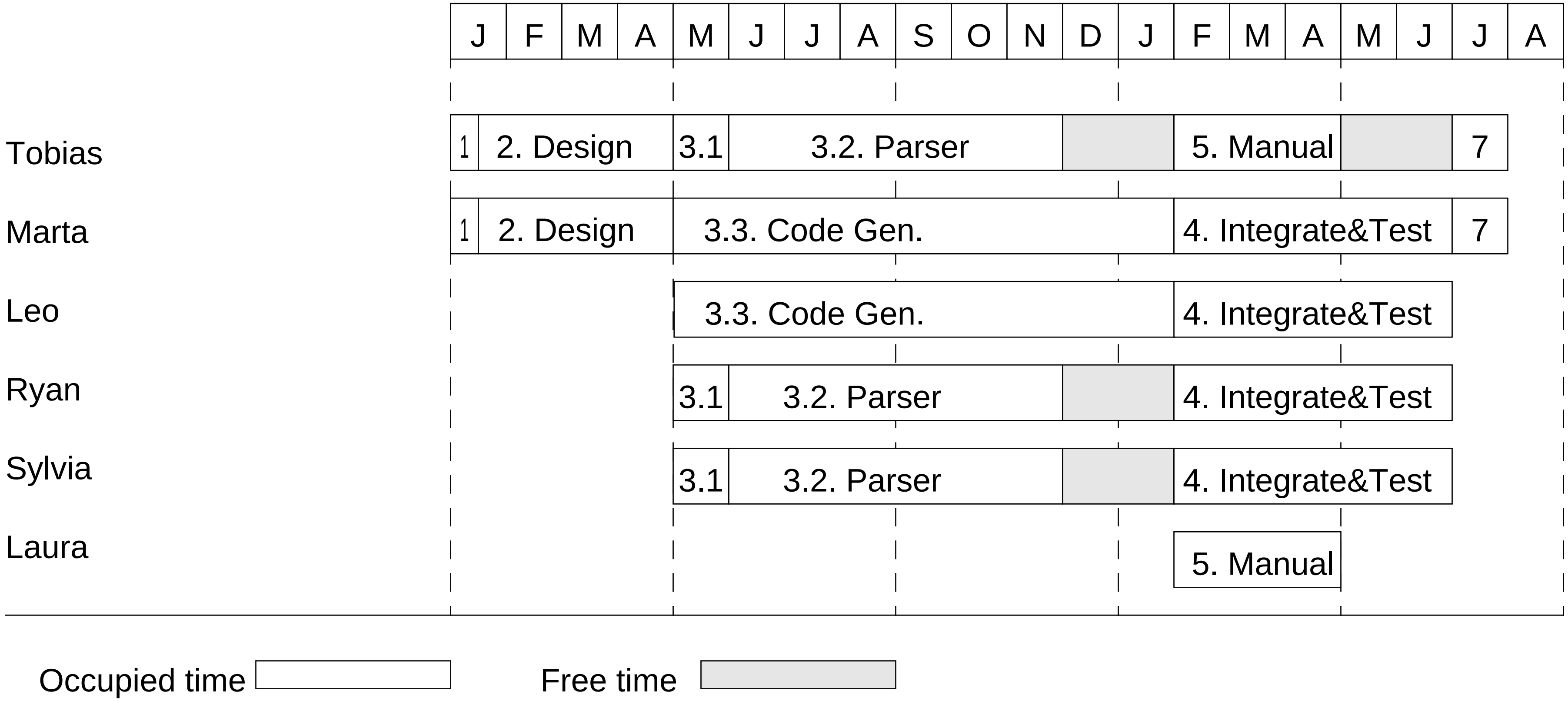
Pert Chart: Activity Network



Gantt Chart: Activity Timeline



Gantt Chart: Staff Allocation



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

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

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

Earned Value: Tasks Completed

The 0/100 Technique

- ❑ earned value := 0% when task not completed
- ❑ earned value := 100% when task completed
  - ☞ tasks should be rather small
  - ☞ gives a pessimistic impression

The 50/50 Technique

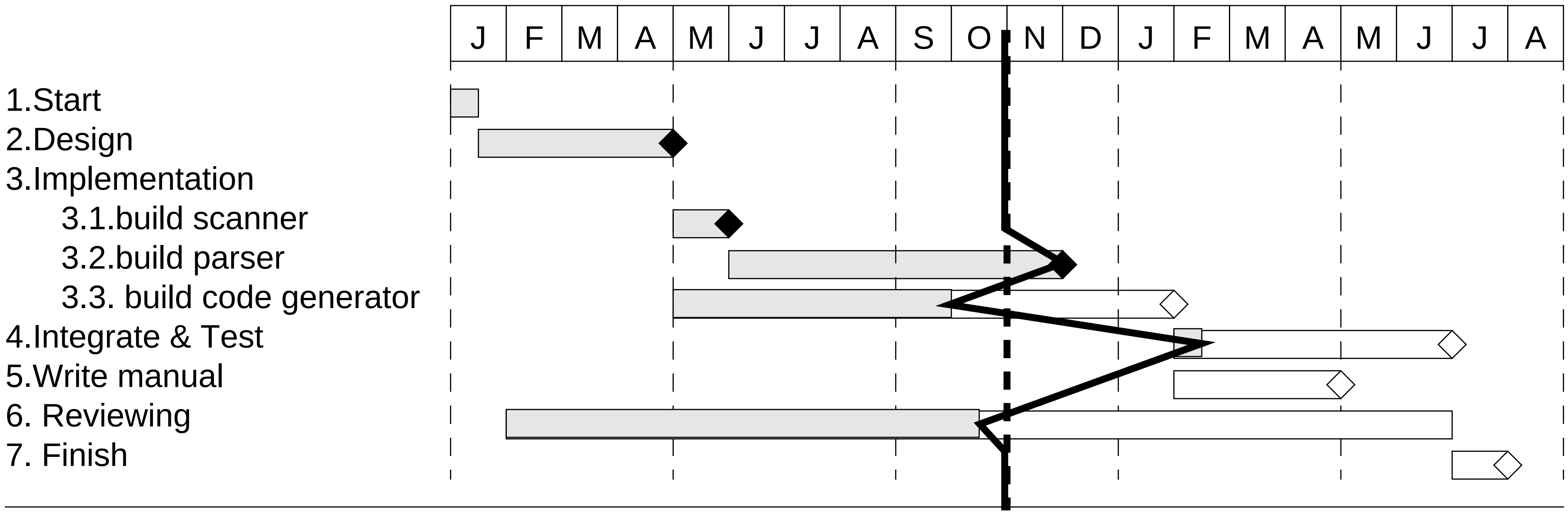
- ❑ earned value := 50% when task started
- ❑ earned value := 100% when task completed
  - ☞ tasks are rather large
  - ☞ may give an optimistic impression
- ❑ variant with 20/80

The Milestone Technique

- ❑ earned value := number of milestones completed / total number of milestones
  - ☞ tasks should be large with lots of intermediate milestones
  - ☞ better to split task in several subtasks and fall back on 0/100

Gantt Chart: Slip Line

- Visualise percentage of task completed via shading
  - ☞ 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



Interpretation (end of october):

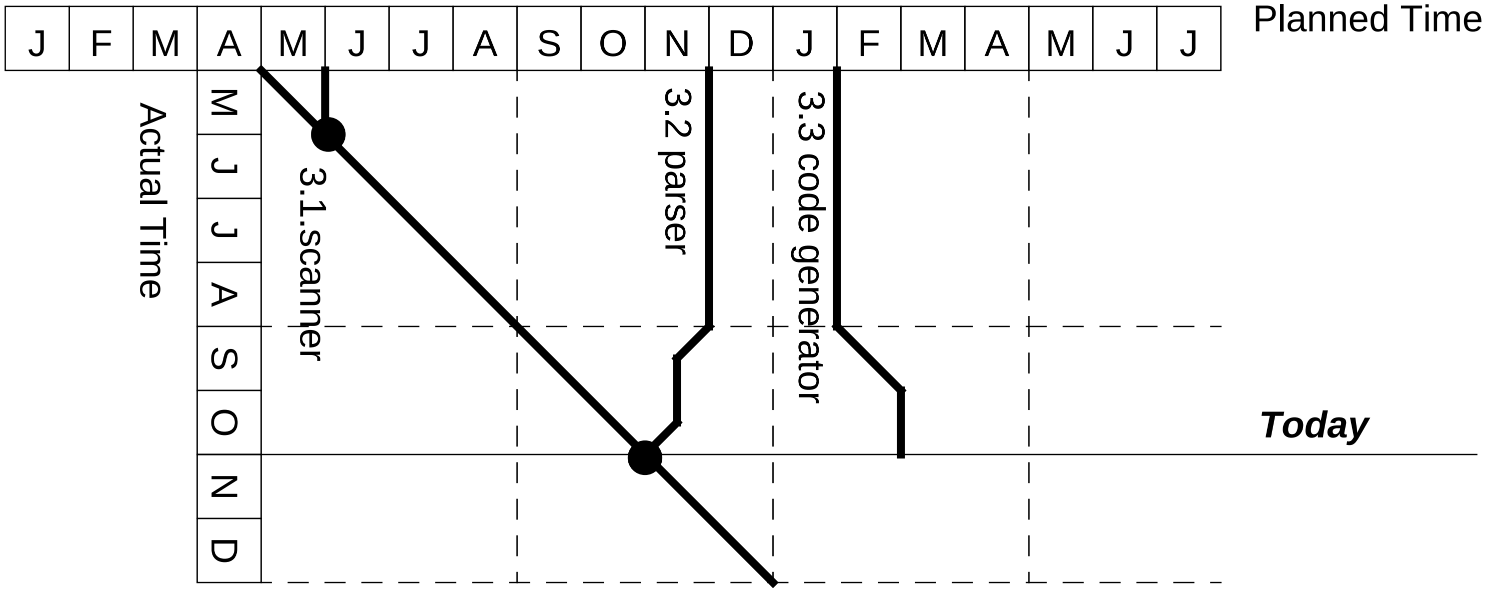
Task 3.2 is finished ahead of schedule and task 4 is started ahead of schedule

Tasks 3.3 and 6 seem to be behind schedule (i.e., less completed than planned)

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



Interpretation (end of october):

Task 3.1: completed on time.

Task 3.2: rescheduled 2 weeks earlier end of August, finished 2 weeks ahead of time.

Task 3.3: rescheduled with one month delay at the end of August

Slip Line vs. Timeline

Slip Line

Monitors current slip status of project tasks

- ❑ many tasks
- ❑ only for 1 point in time
  - ☞ include a few slip lines from the past to illustrate evolution

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*

- 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

Directing a team = the whole becomes more then the sum of its parts

Managers serve their team

- ❑ Managers ensure that team has the necessary information and resources

  “The managers 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.

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

Conway's Law

"Organizations that design systems are constrained to produce designs that are copies of the communication structures of these organizations"

Summary

You should know the answers to these questions:

- ❑ 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 the following 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?*
- ✎ *What are good examples of Conway's Law in action?*

3. Requirements Collection

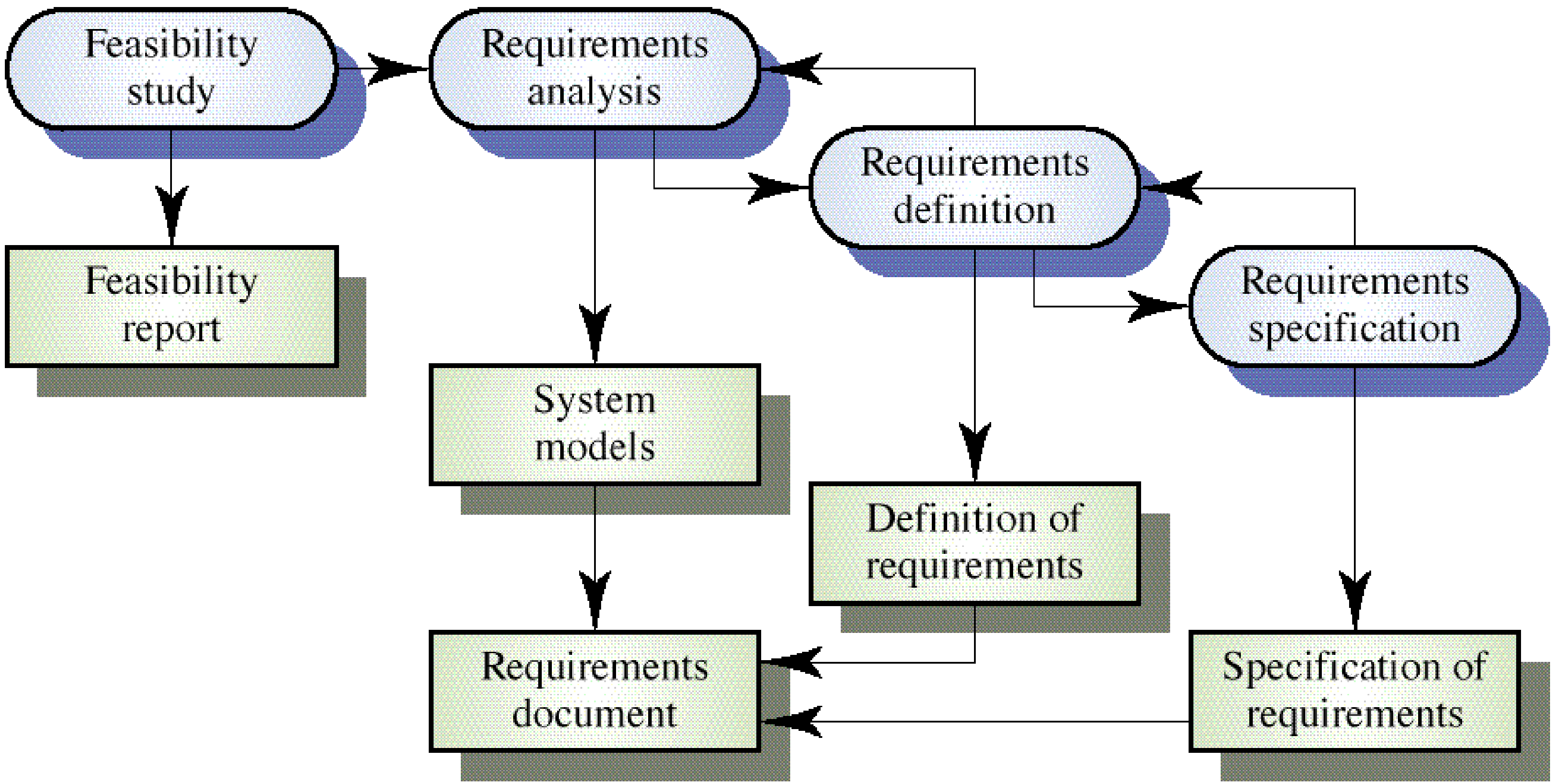
Overview:

- The Requirements Engineering Process
  - ☞ Requirements Analysis, Definition and Specification
- Use cases and scenarios
- Functional and non-functional requirements
- Evolutionary and throw-away prototyping
- Requirements checking and reviews

Sources:

- *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.
- *Software Engineering — A Practitioner's Approach*, R. Pressman, Mc-Graw Hill, Third Edn., 1994.
- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999

The Requirements Engineering Process



Requirements Engineering Activities

Feasibility study

- Determine if the user needs can be satisfied with the available technology and budget.

Requirements analysis

- Find out what system stakeholders require from the system.

Requirements definition

- Define the requirements in a form understandable to the customer.

Requirements specification

- Define the requirements in detail.

  Written as a contract between client and contractor.

"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.

Problems of Requirements Analysis

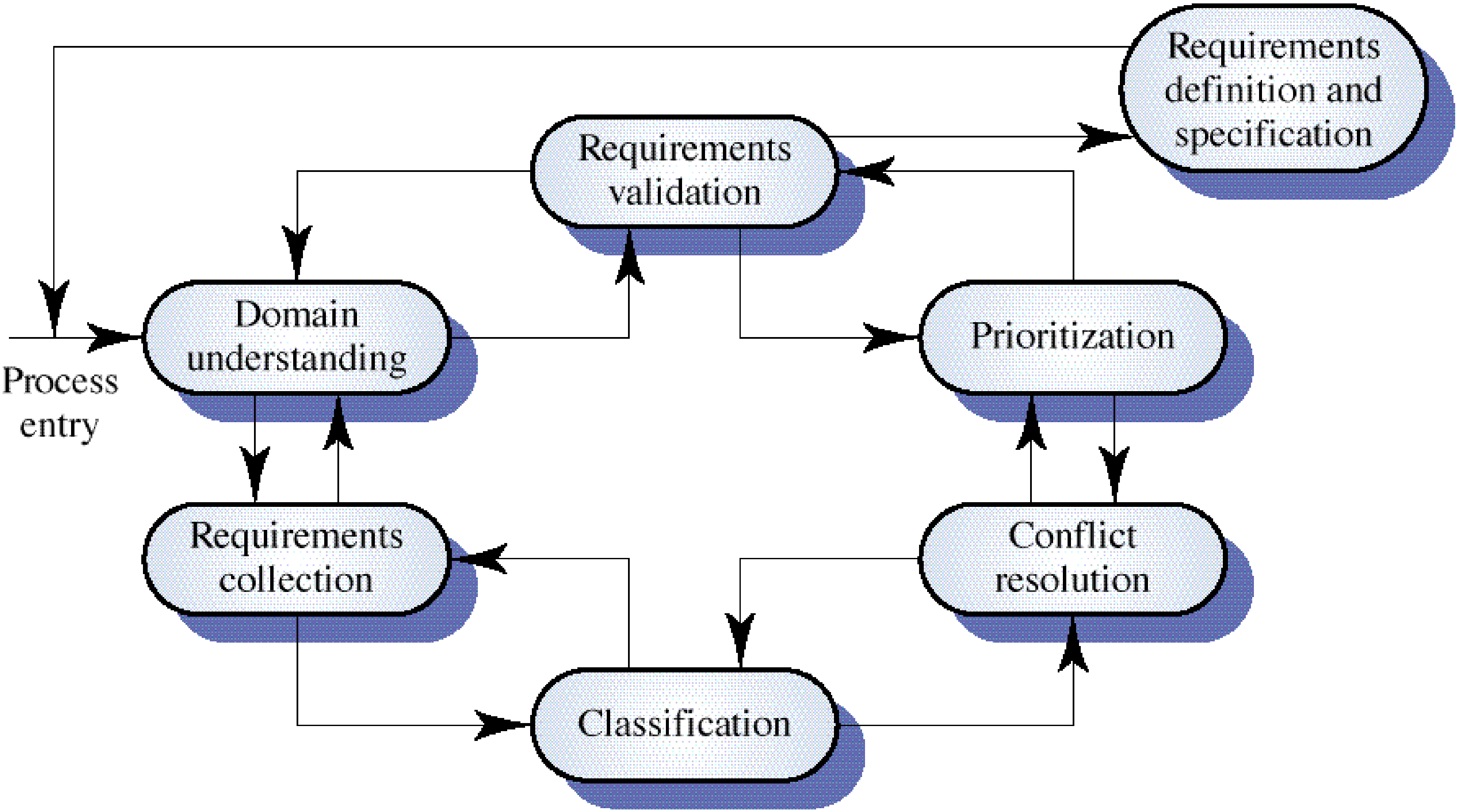
Various problems typically arise:

- ❑ 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.

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



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".

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

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

The "Unified Modeling Language" (UML) is an emerging industrial standard for documenting object-oriented analysis and design models.

- ***Class Diagrams:*** specify classes, objects and their relationships
  - ☞ visualize logical structure of system
- ***Use Case Diagrams:*** show external actors and use cases they participate in
- ***Sequence Diagrams:*** list the message exchanges in a use case scenario
  - ☞ visualizes temporal message ordering
- ***Collaboration Diagrams:*** show messages exchanged by objects
  - ☞ visualize object relationships
- ***State Diagrams:*** specify the possible internal states of an object

and others ...

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

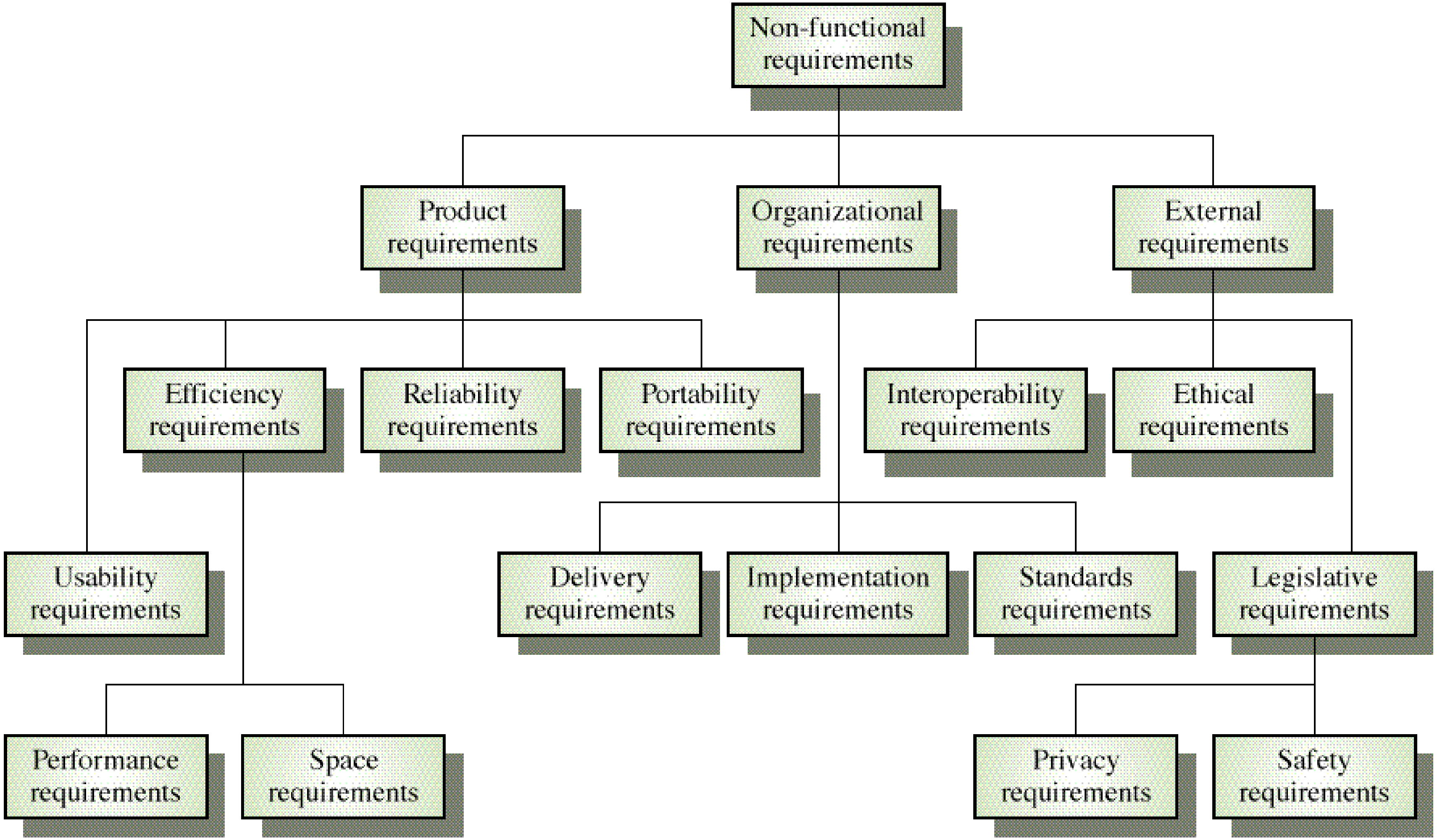
Functional requirements describe system *services* or *functions*

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

- Product requirements:
  - ☞ specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organisational requirements:
  - ☞ are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements:
  - ☞ arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

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

Types of Non-functional Requirements



Examples of Non-functional Requirements

Product requirement

- It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set.

Organisational requirement

- The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.

External requirement

- The system shall provide facilities that allow any user to check if personal data is maintained on the system. A procedure must be defined and supported in the software that will allow users to inspect personal data and to correct any errors in that data.

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

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

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

Validity:

- Does the system provide the functions *which best support* the customer's needs?

Consistency:

- Are there any requirements *conflicts?*

Completeness:

- Are *all functions* required by the customer included?

Realism:

- Can the requirements be implemented given *available budget* and *technology?*

Requirements Reviews

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

- *Verifiability.* Is the requirement realistically *testable*?
- *Comprehensibility.* Is the requirement properly *understood*?
- *Traceability.* Is the *origin* of the requirement clearly stated?
- *Adaptability.* Can the requirement be *changed* without a large *impact* on other requirements?

Traceability

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

	Comp 1	Comp 2	⋮	⋮	⋮	⋮	⋮	⋮	⋮	Comp m
Req 1				x			x			
Req 2	x									x
...										
...		x				x			x	
...										
...		x								
...					x					x
Req n										

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

Summary

You should know the answers to these questions:

- ❑ 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"?*

4. Responsibility-Driven Design

Overview:

- What is Object-Oriented Design?
- Finding Classes
- Identifying Responsibilities
- Finding Collaborations

Source:

- *Designing Object-Oriented Software*, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990.

Why Responsibility-driven Design?

Object-Oriented Decomposition	Functional Decomposition
Decompose according to the objects a system is supposed to manipulate.	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

Object-Oriented Decomposition

- ❑ Better for complex and evolving systems

However

- ❑ How to find the objects?

What is Object-Oriented Design?

"Object-oriented [analysis and] design is the process by which software requirements are turned into a detailed specification of objects. This specification includes a complete description of the respective roles and responsibilities of objects and how they communicate with each other."

- ❑ 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.*

Design Steps

The Initial Exploration

1. Find the classes in your system
2. Determine the responsibilities of each class
   - ☞ What are the client-server *contracts?*
3. Determine how objects collaborate with each other to fulfil their responsibilities
   - ☞ What are the client-server *roles?*

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
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* — does it really signal 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 (I)

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

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 (II)

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*

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

Class Selection Rationale (III)

Model interfaces to the system:

- ~~user~~ — *don't need to model user explicitly*
- ~~cursor~~ — *cursor motion handled by operating system*

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:

Class: Drawing	
superclasses	
subclasses	
responsibilities ...	*collaborations*

Write a short description of the purpose of the class on the back of the card

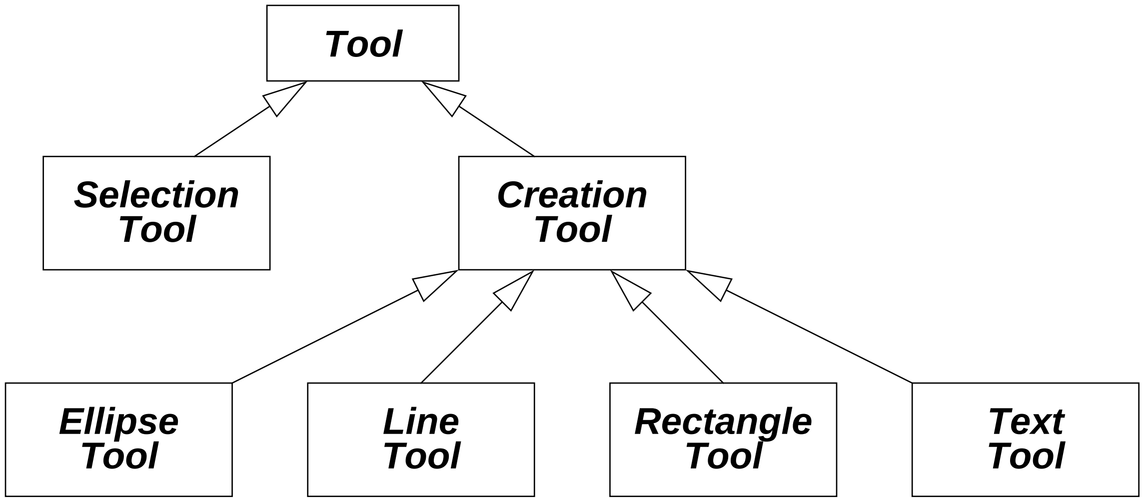
- ☞ compact, easy to manipulate, easy to modify or discard!
- ☞ easy to arrange, reorganize
- ☞ easy to retrieve discarded classes

Finding Abstract Classes

Abstract classes factor out common behaviour shared by other classes

They are *abstract* because they need not be completely implemented.

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



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

Identifying and Naming Groups

If you have trouble *naming* a group:

- ☞ enumerate common attributes to derive the name
- ☞ divide into more clearly defined subcategories

Attributes of abstract classes should serve to distinguish subgroups

- ☞ Physical vs. conceptual
- ☞ Active vs. passive
- ☞ Temporary vs. permanent
- ☞ Generic vs. specific
- ☞ Shared vs. unshared

Classes may be missing because the specification is incomplete or imprecise

- ☞ editing ⇒ undoing ⇒ need for a Cut Buffer

Recording Superclasses

Record superclasses and subclasses on all class cards:

Class: Creation Tool	
Tool	
Ellipse Tool, Line Tool, Rectangle Tool, Text Tool	

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, 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-though of the system
    - exploring 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
- 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 *implemented in subclass*
    2. determine location to place it *generic*
    3. instantiate the element *implemented in subclass*
- 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

Recording Responsibilities

List responsibilities as succinctly as possible:

Class: Drawing	
Know which elements it contains	

Too many responsibilities to fit onto one card suggests over-centralization:

- ☞ Check if responsibilities really belong in a superclass, or if they can be distributed to cooperating classes.

Having more classes leads to a more flexible and maintainable design. If necessary, classes can later be consolidated.

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!)

Check for these relationships:

- ❑ The “Is-Part-Of” Relationship
- ❑ The “Has-Knowledge-Of” Relationship
- ❑ The “Depends-Upon” Relationship

Recording Collaborations

Collaborations exist only to fulfil responsibilities.

Enter the class name of the server role next to client's responsibility:

Class: Drawing	
Know which elements it contains	
Maintain ordering between elements	Drawing Element

Note *each* collaboration required for a responsibility.

Include also collaborations between peers.

Validate your preliminary design with another walk-through.

Summary

You should know the answers to these questions:

- ❑ What criteria can you use to identify potential classes?
- ❑ How can class 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?

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?*

5. Detailed Design

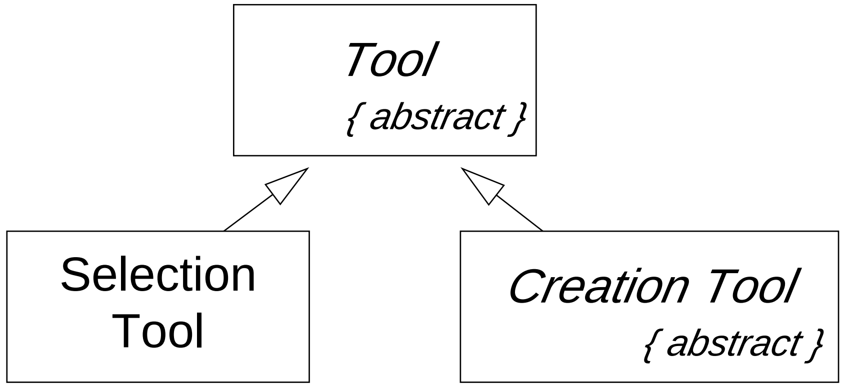
Overview:

- ❑ Structuring Inheritance Hierarchies
- ❑ Identifying Subsystems
- ❑ Specifying Class Protocols (Interfaces)

Source:

- ❑ *Designing Object-Oriented Software*, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990

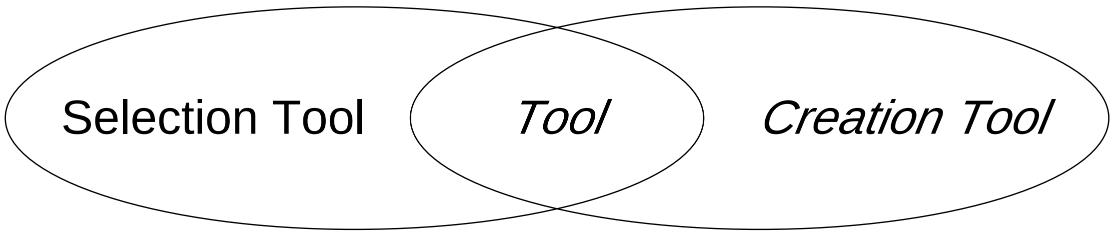
Sharing Responsibilities



Concrete classes may be both instantiated and inherited from.

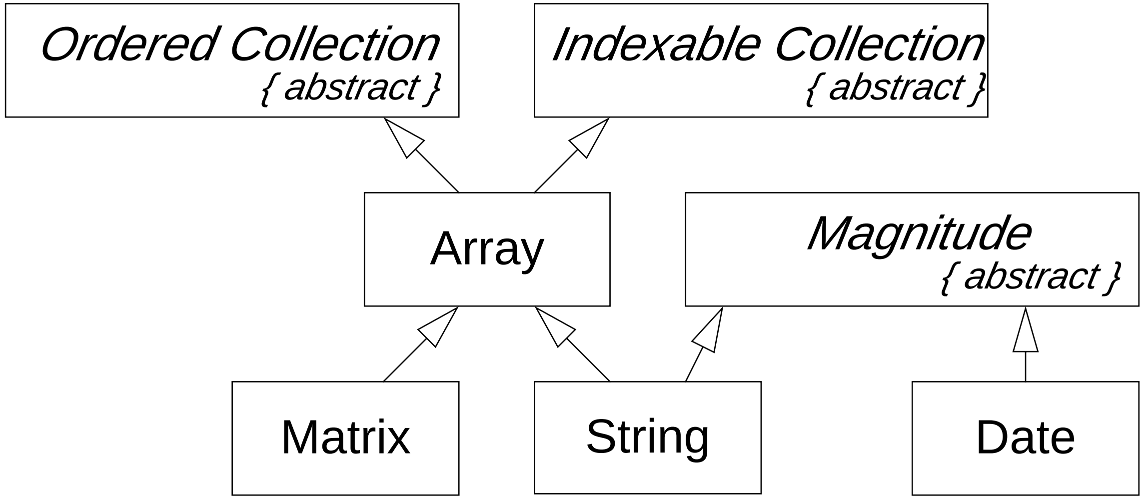
Abstract classes may only be inherited from. *Note on class cards and on class diagram.*

Venn Diagrams can be used to visualize shared responsibilities:



(Warning: not part of UML!)

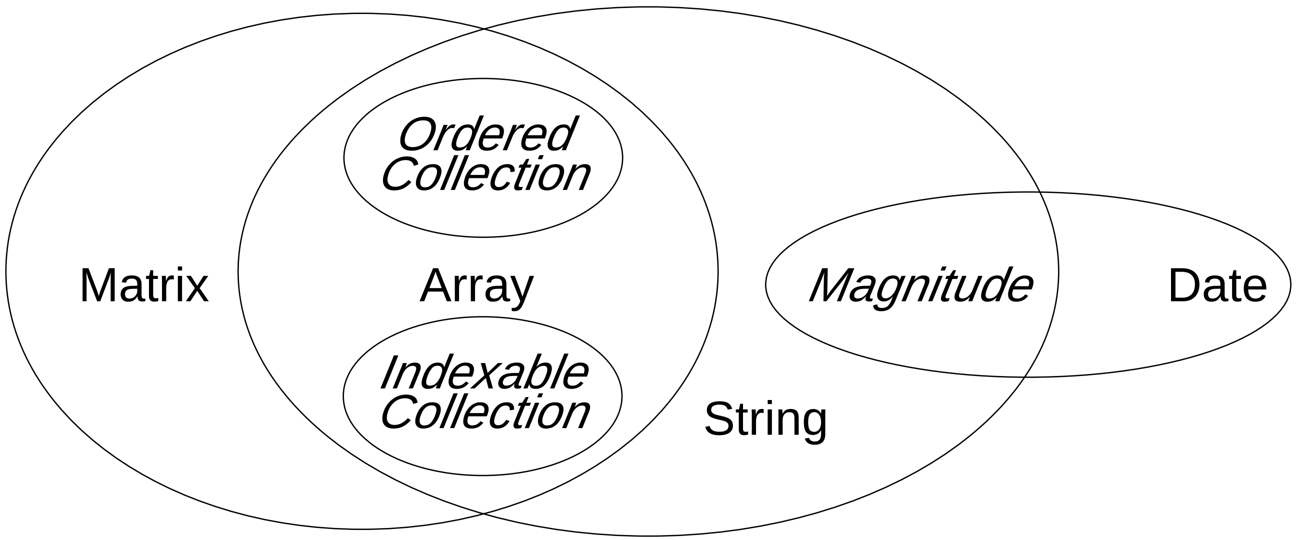
Multiple Inheritance



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

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

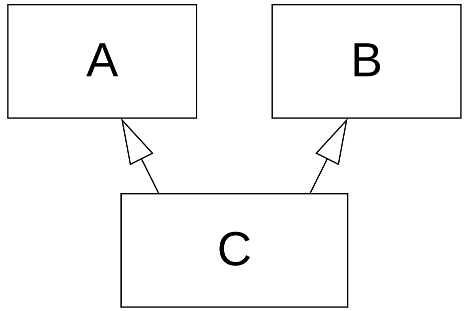
- ☞ Eliminate by introducing common abstract superclass: *abstract classes should support responsibilities in an implementation-independent way*

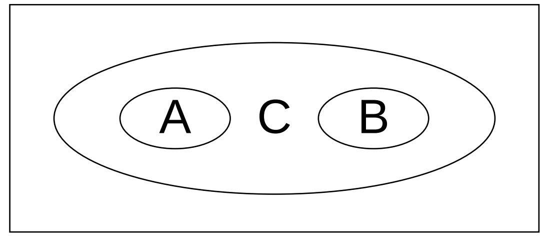
Eliminate classes that do not add functionality:

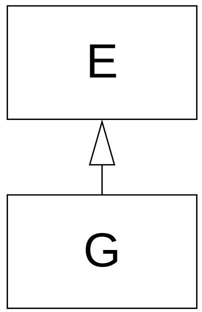
- ☞ Classes should either add new responsibilities, or a particular way of implementing inherited ones

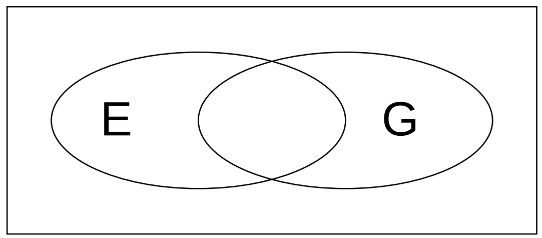
Building Kind-Of Hierarchies

Correctly Formed Subclass Responsibilities







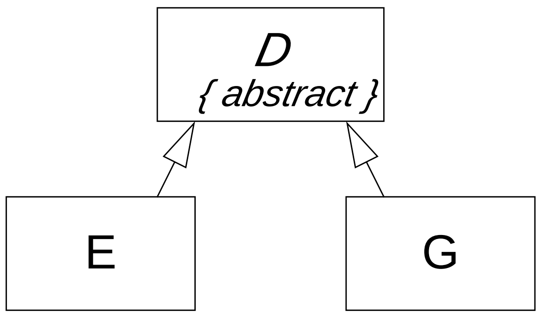


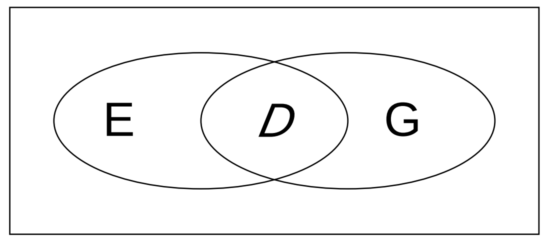
Incorrect Subclass/Superclass Relationships

Subclasses should assume *all* superclass responsibilities

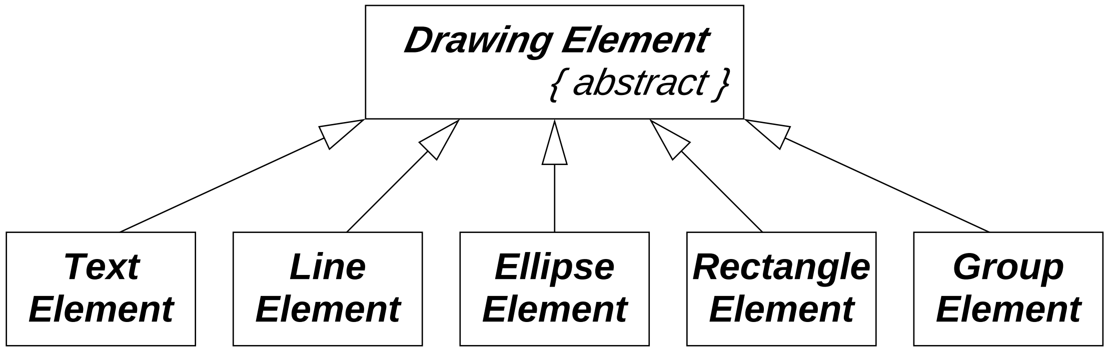
Revised Inheritance Relationships

Introduce abstract superclasses to encapsulate common responsibilities



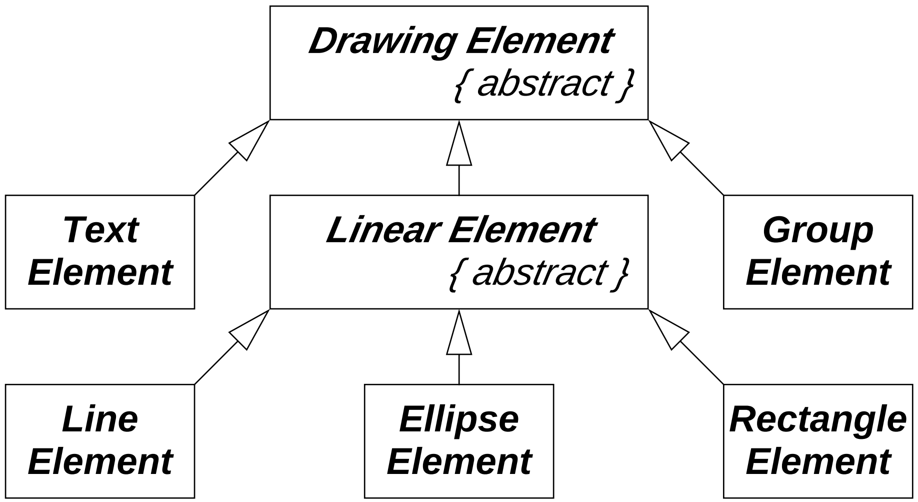


Refactoring Responsibilities



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.



Identifying Contracts

A *contract* defines a set of requests that a client can make of a server related to a cohesive set of closely-related responsibilities.

Contracts introduce another level of abstraction, and help to simplify your design.

- *Group* responsibilities used by the same clients:
  - ☞ conversely, separate clients suggest separate contracts
- *Maximize* the cohesiveness of classes:
  - ☞ unrelated contracts belong in subclasses
- *Minimize* the number of contracts:
  - ☞ unify responsibilities and move as high in the hierarchy as appropriate

Applying the Guidelines

1. Start by defining contracts at the top of your hierarchies

2. Introduce new contracts only for subclasses that add significant new functionality
   - ☞ do new responsibilities represent new functionality, or do they just specialize inherited functionality?

3. For each class card, assign responsibilities to an appropriate contract
   - ☞ briefly describe each contract and assign a unique number
   - ☞ number responsibilities according to the associated contract

4. For each collaboration on each class card, determine which contract represents it
   - ☞ model collaborations as associations in class diagrams (AKA “collaboration graphs”)

What are Subsystems?

Subsystems are groups of classes that collaborate to support a set of contracts.

- ❑ Subsystems simplify design by raising abstraction levels:
  - ☞ subsystems group logically related responsibilities, and encapsulate related collaborations
- ❑ Don’t confuse with superclasses:
  - ☞ subsystems group related responsibilities rather than factoring out common responsibilities

Find subsystems by looking for *strongly-coupled* classes:

- ☞ list the collaborations and identify strong inter-dependencies
- ☞ identify and highly frequently-travelled communication paths

Subsystems, like classes, also support contracts. Identify the services provided to clients *outside* the subsystem to determine the subsystem contracts.

Subsystem Cards

For each subsystem, record its name, its contracts, and, for each contract, the internal class or subsystem that supports it:

Subsystem: Drawing Subsystem	
Access a drawing	Drawing
Modify part of a drawing	Drawing Element
Display a drawing	Drawing

Class Cards

For each collaboration from an outside client, change the client's class card to record a collaboration with the subsystem:

Class: File	(*Abstract*)
Document File, Graphics File, Text File	
Knows its contents	
Print its contents	Printing Subsystem

Record on the subsystem card the delegation to the agent class.

Simplifying Interactions

Complex collaborations lead to unmaintainable systems.
Exploit subsystems to simplify overall structure.

- ❑ Minimize the number of collaborations a class has with other classes:
  - ☞ centralizing communications into a subsystem eases evolution
- ❑ Minimize the number of classes to which a subsystem delegates:
  - ☞ centralized subsystem interfaces reduce complexity
- ❑ Minimize the number of different contracts supported by a class:
  - ☞ group contracts that require access to common information

Checking Your Design:

- ☞ model collaborations as associations in class diagrams
- ☞ update class/subsystem cards and class hierarchies
- ☞ walk through scenarios:
  - ➪ Has coupling been reduced? Are collaborations simpler?

Protocols

A *protocol* is a set of signatures (i.e., method names, parameter types and return types) 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

Refining Responsibilities

Select method names carefully:

- ☞ Use a single name for each conceptual operation in the system
- ☞ Associate a single conceptual operation with each method name
- ☞ Common responsibilities should be explicit in the inheritance hierarchy

Make protocols as generally useful as possible:

- ☞ The more general it is, the *more* messages that should be specified

Define reasonable defaults:

1. Define the most general message with all possible parameters
2. Provide reasonable default values where appropriate
3. Define specialized messages that rely on the defaults

Specifying Your Design: Classes

Specifying Classes

1. Class name; abstract or concrete
2. Immediate superclasses and subclasses
3. Location in inheritance hierarchies and class diagrams
4. Purpose and intended use
5. Contracts supported (as server); inherited contracts and ancestor
6. For each contract, list responsibilities, method signatures, brief description and any collaborations
7. List private responsibilities; if specified further, also give method signatures etc.
8. Note: implementation considerations, possible algorithms, real-time or memory constraints, error conditions etc.

Specifying Subsystems and Contracts

Specifying Subsystems

1. Subsystem name; list all encapsulated classes and subsystems
2. Purpose of the subsystem
3. Contracts supported
4. For each contract, list the responsible class or subsystem

Formalizing Contracts

1. Contract name and number
2. Server(s)
3. Clients
4. A description of the contract

Summary

You should know the answers to these questions:

- ❑ 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?
- ❑ What is the difference between contracts and responsibilities?
- ❑ What are subsystems (“categories”) and how can you find them?
- ❑ What is the difference between protocols and contracts?

Can you answer the following questions?

- ✎ *What use is multiple inheritance during design if your programming language does not support it?*
- ✎ *Why should you try to minimize coupling and maximize cohesion?*
- ✎ *How would you use Responsibility Driven design together with the Unified Modeling Language?*

6. Modeling Objects and Classes

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

Sources:

- ❑ *Unified Modeling Language — Notation Guide*, version 1.3, Rational Software Corporation, 1997.
- ❑ *Object-Oriented Development — The Fusion Method*, D. Coleman, et al., Prentice Hall, 1994.
- ❑ *UML Distilled,* Martin Fowler, Kendall Scott, Addison-Wesley, Second Editon, 2000.

Why UML?

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 artefact

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

What is UML?

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

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:

Polygon
centre: Point vertices: List of Point borderColour: Colour fillColour: Colour
display (on: Surface) rotate (angle: Integer) erase ( ) destroy ( ) select (p: Point): Boolean

A collapsed class view:

Polygon

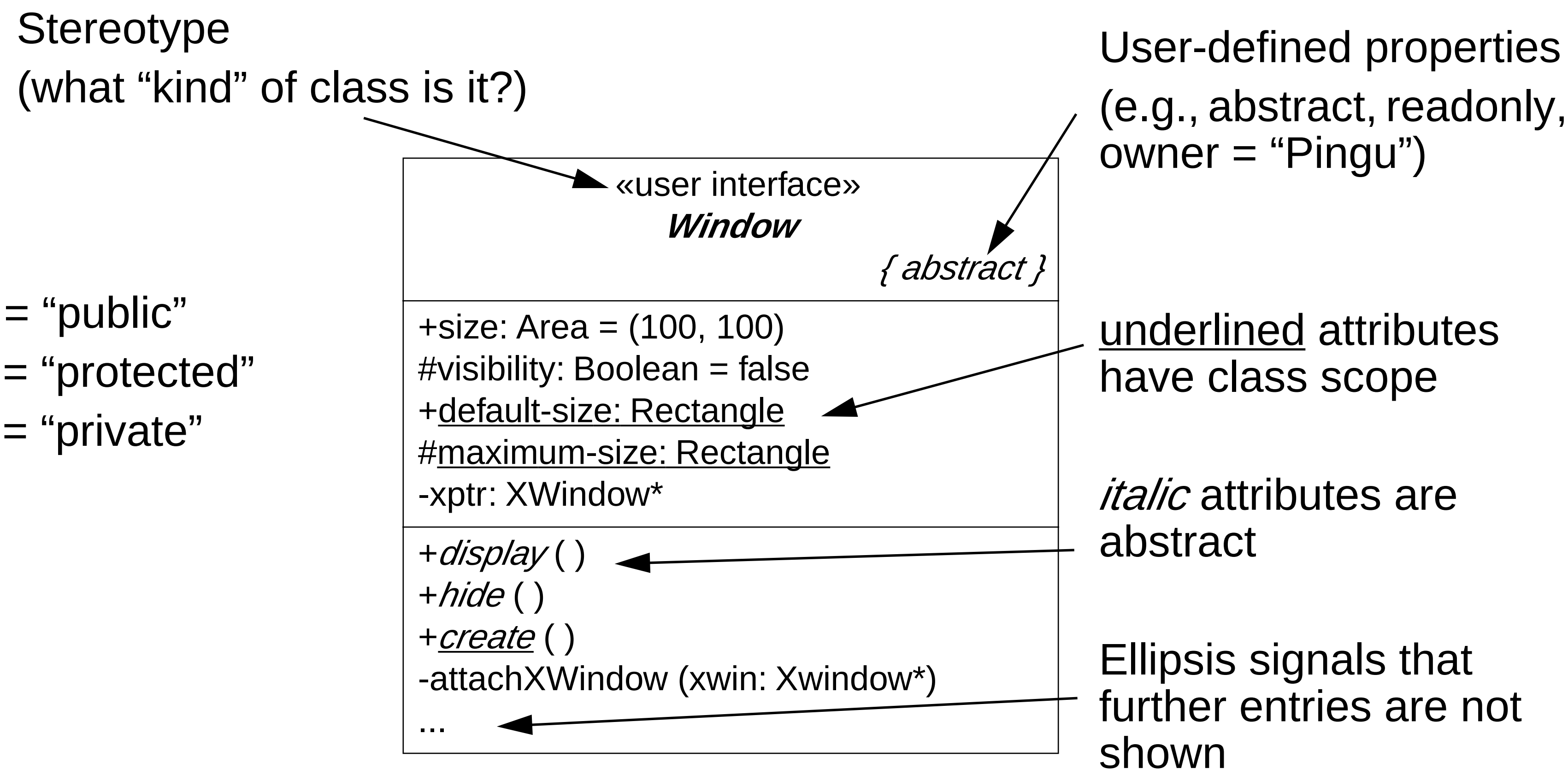
Class with Package name:

ZWindows::Window

Attributes and operations are also collectively called *features*.

Visibility and Scope of Features

+ = "public"
# = "protected"
– = "private"



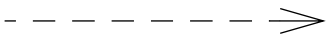
Attributes are specified as: *name: type = initialValue { property string }*

Operations are specified as: *name (param: type = defaultValue, ...) : resultType*

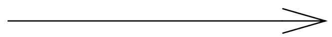
UML Lines and Arrows

Constraint
(usually annotated)

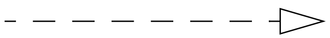
Association
e.g., «uses»



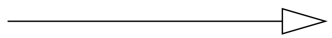
Dependency
e.g., «requires», «imports» ...



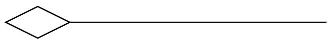
Navigable association
e.g., part-of



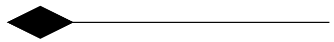
Realization
e.g., class/template, class/interface



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



Aggregation
i.e., "consists of"

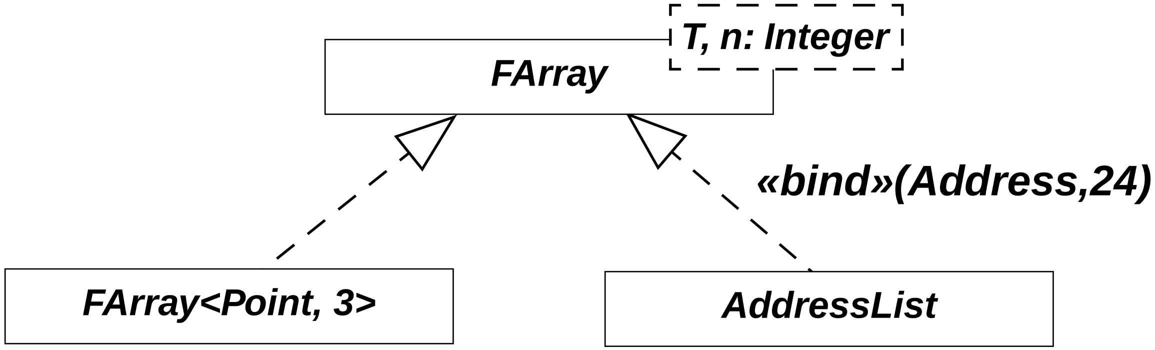


"Composition"
i.e., containment

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***).

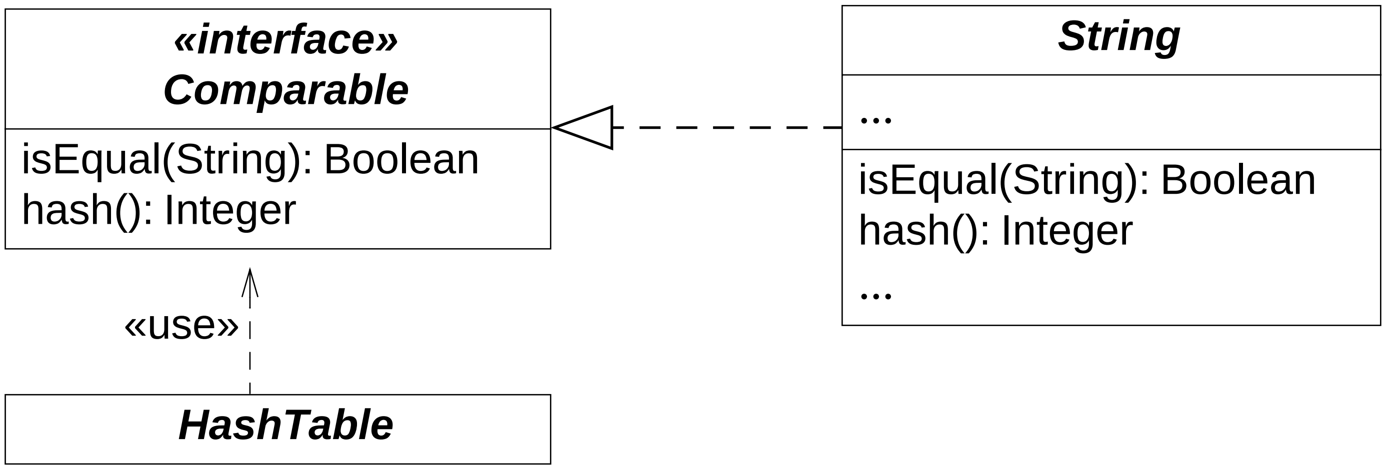


Instantiation of a class from a template can be shown by a dashed arrow (***Realization***).

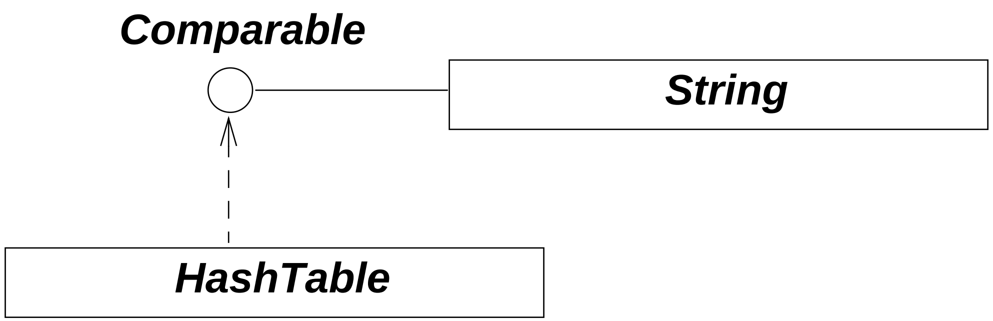
NB: All forms of arrows (directed arcs) go from the client to the supplier!

Interfaces

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

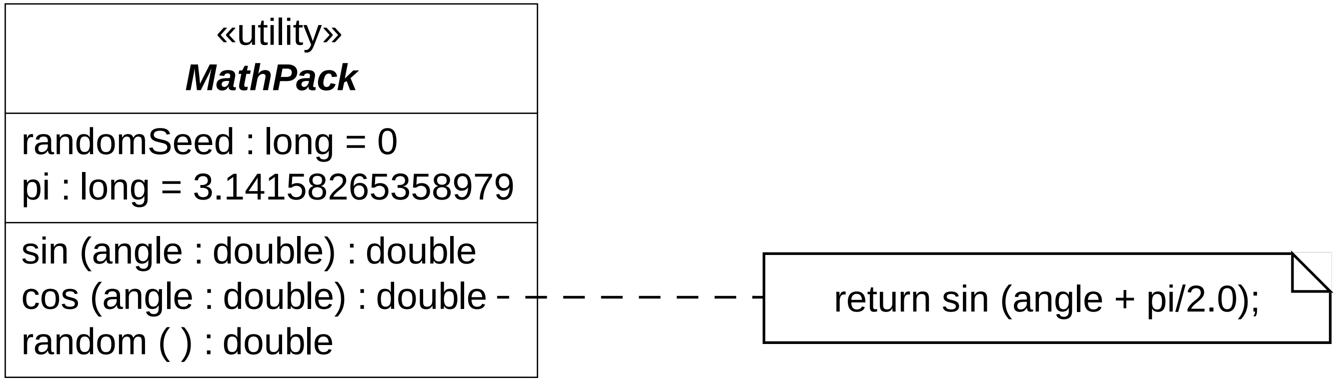


NB: Interfaces cannot have (navigable) associations!



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.



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.

triangle1: Polygon
centre = (0, 0) vertices = ((0,0), (4,0), (4,3)) borderColour = black fillColour = white

triangle1: Polygon

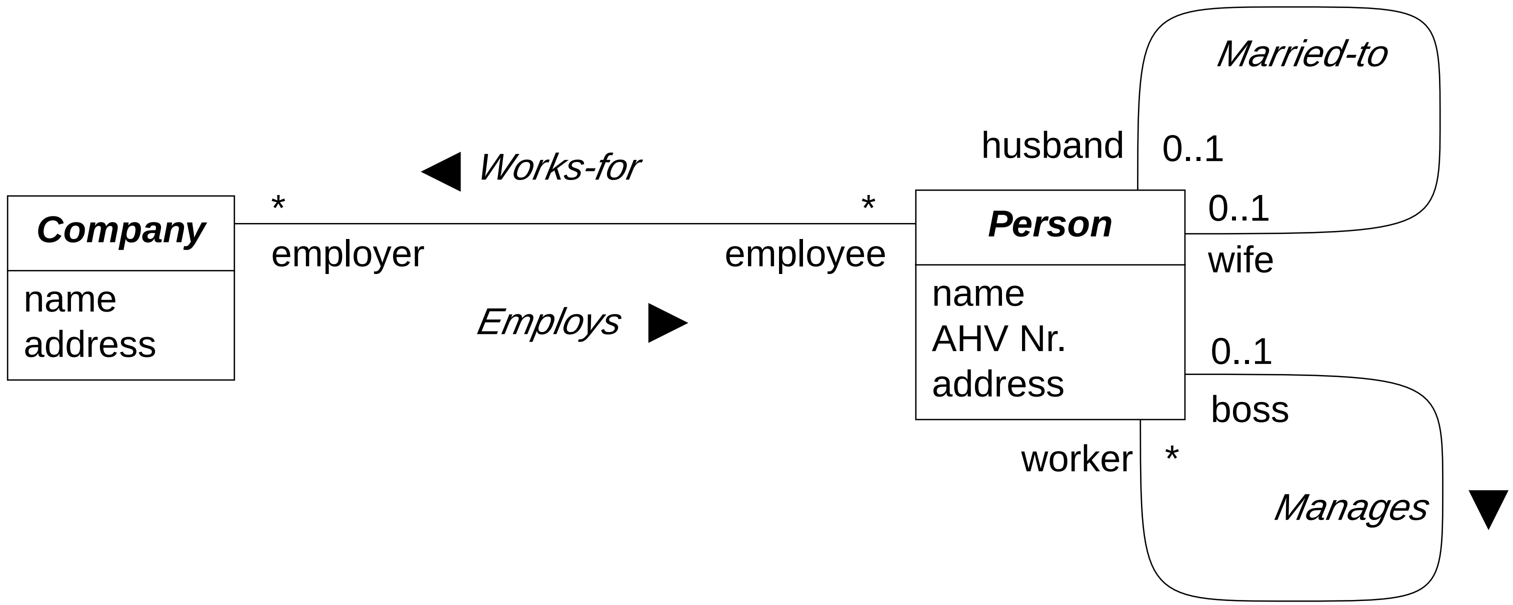
triangle1

: Polygon

At least one of the name or the type must be present.

Associations

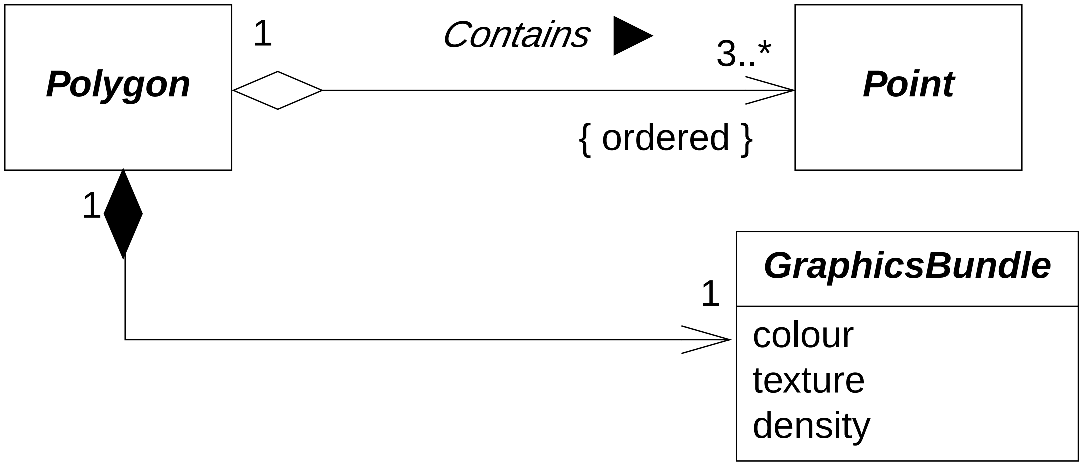
Associations represent structural relationships between objects of different classes.



- 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:



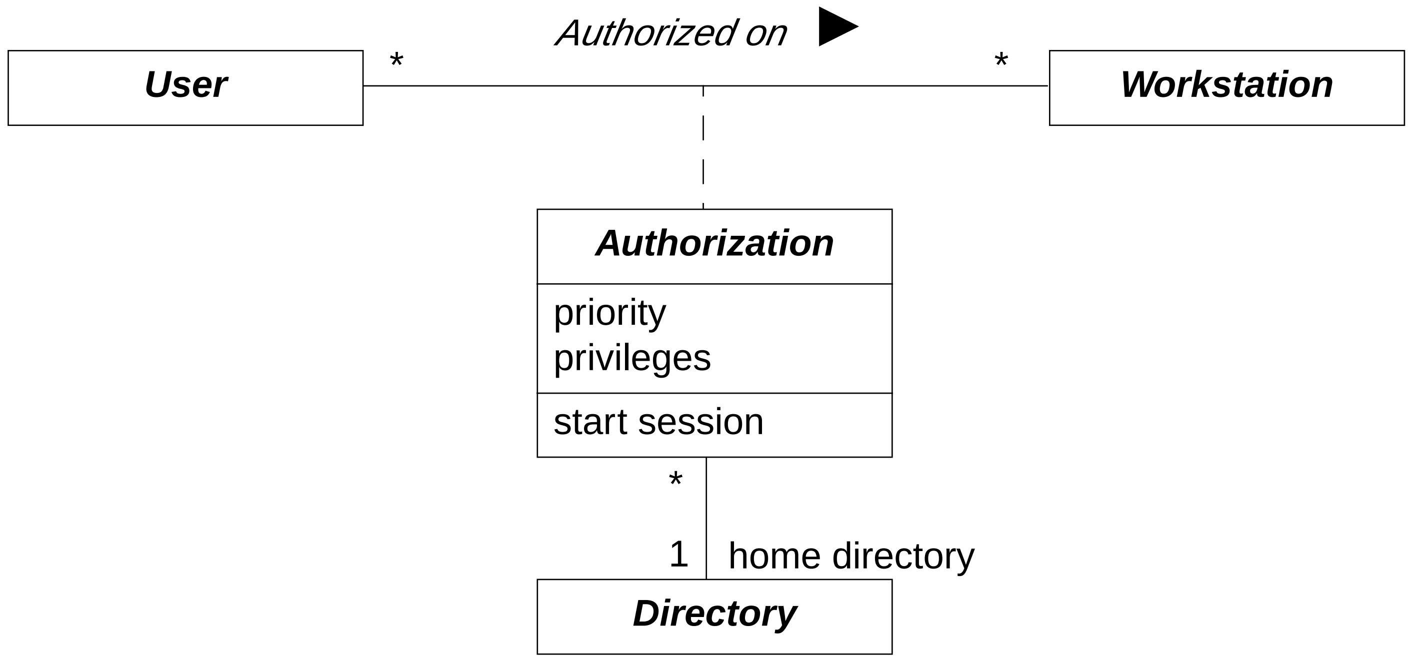
A hollow diamond indicates a reference; a solid diamond an implementation.

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

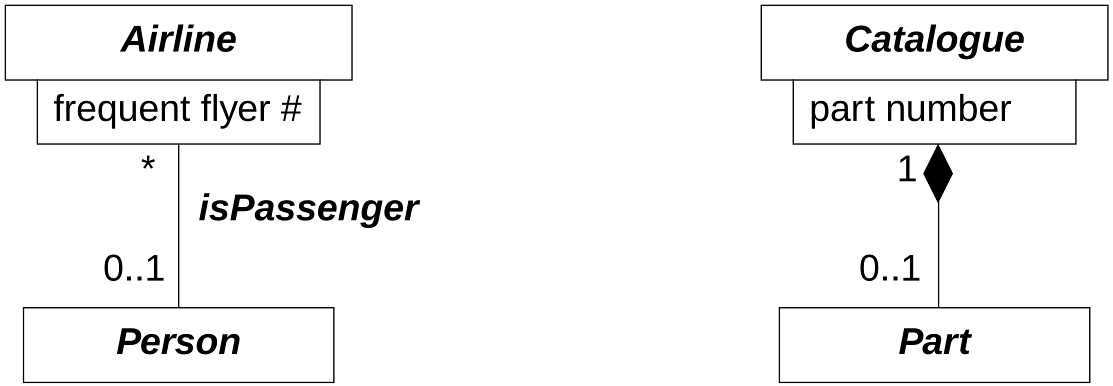
An association may be an instance of an association class:



In many cases the association class only stores attributes, and its name can be left out.

Qualified Associations

A qualified association uses a special qualifier value to identify the object at the other end of the association:

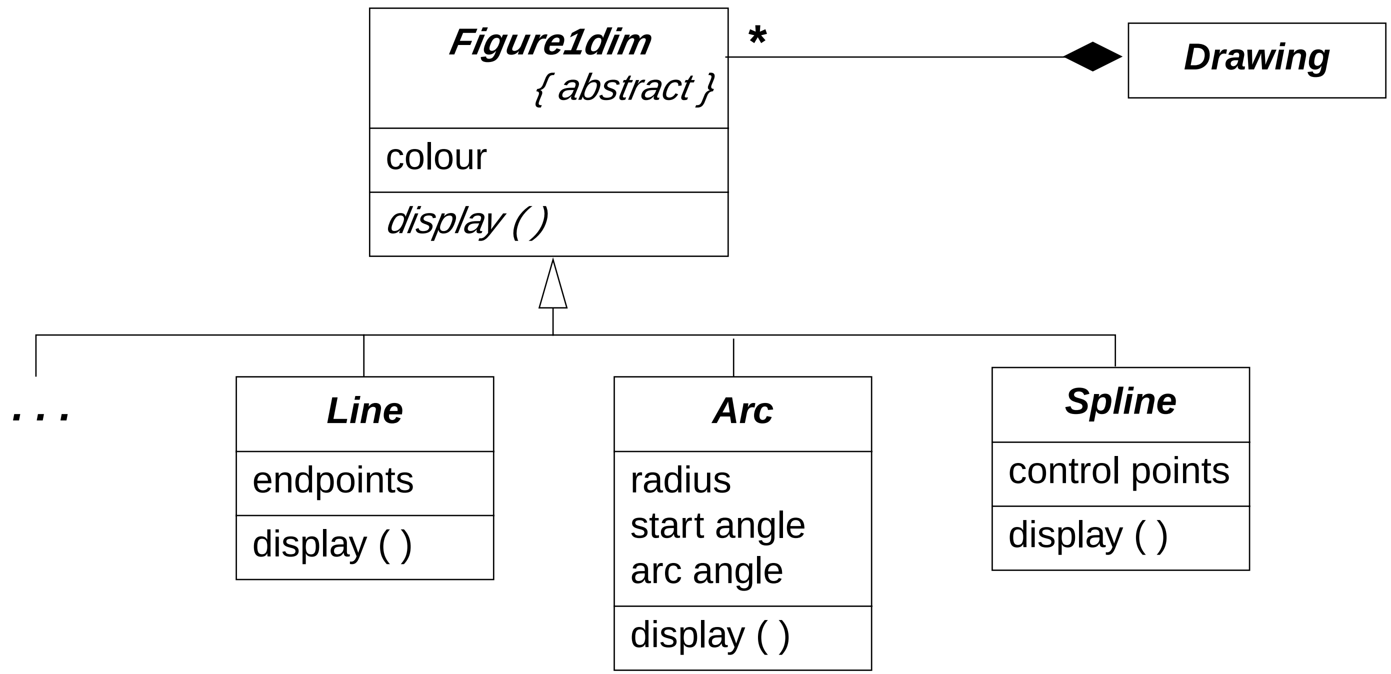


"The multiplicity attached to the target role denotes the possible cardinalities of the set of target objects selected by the pairing of a source object and a qualifier value."

NB: Qualifiers are part of the association, not the class

Inheritance

A subclass inherits the features of its superclasses:



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:

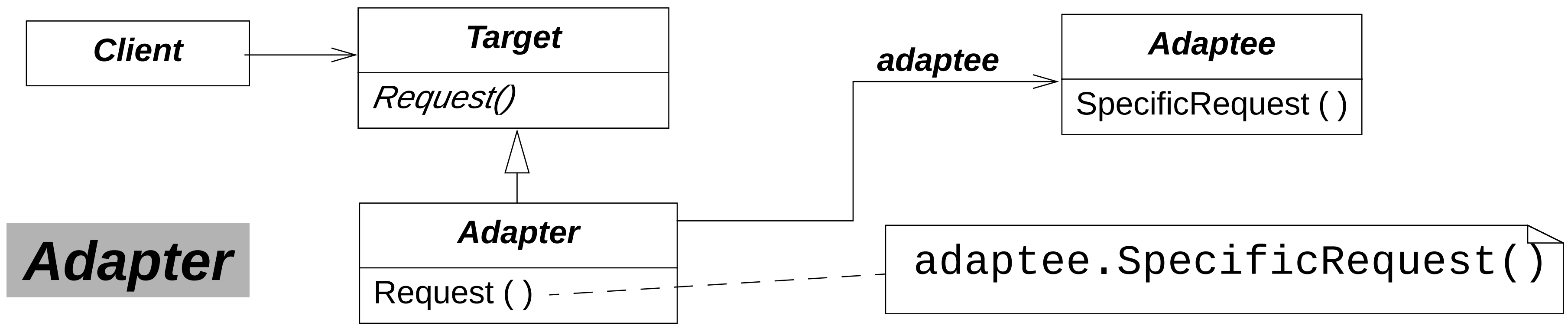
- ❑ related classes may share interfaces, data structures or behaviour
  - ☞ geometric objects ...

Polymorphism:

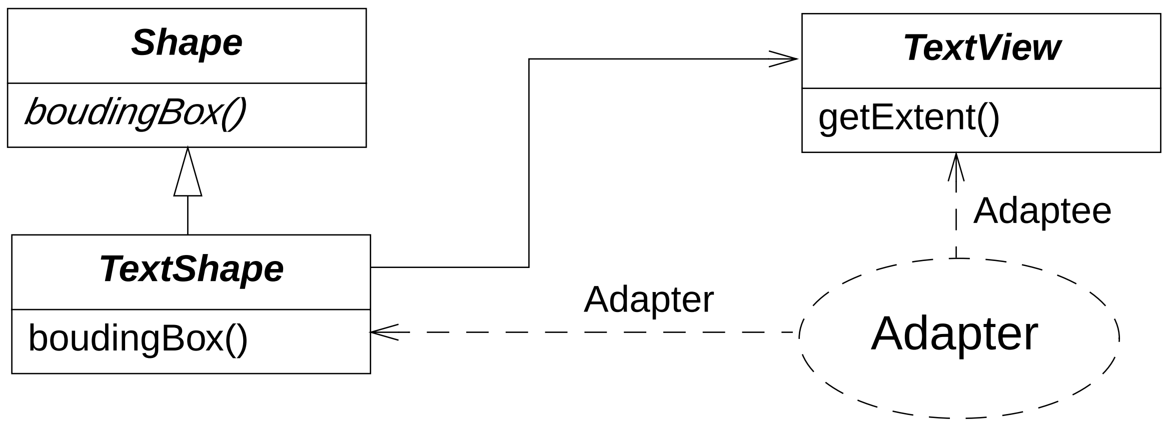
- ❑ objects of distinct, but related classes may be uniformly treated by clients
  - ☞ array of geometric objects

Design Patterns as Collaborations

Design Patterns can be represented as parameterized collaborations:



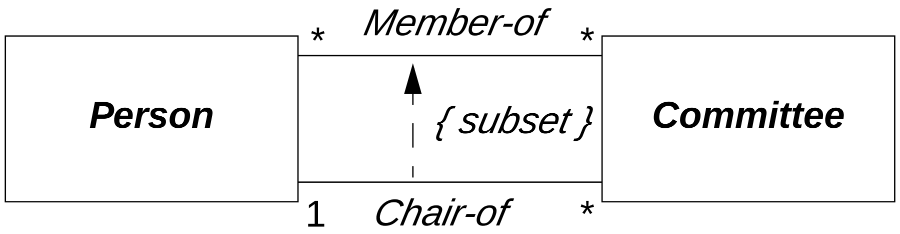
A Design Pattern in use (an *instantiation*) can be described with a dashed oval:

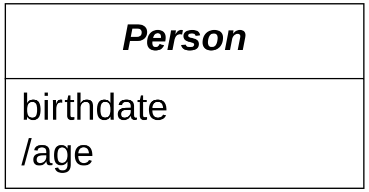


Constraints

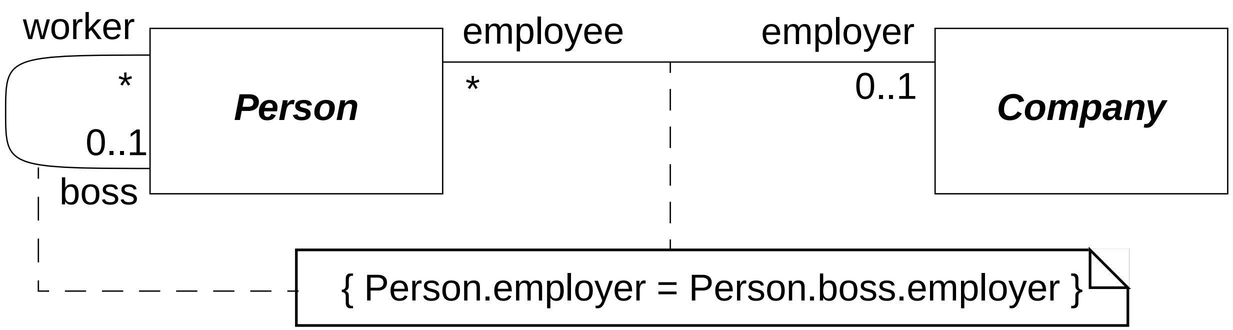
Constraints are restrictions on values attached to classes or associations.

- ☞ Binary constraints may be shown as dashed lines between elements
- ☞ Derived values and associations can be marked with a "/"



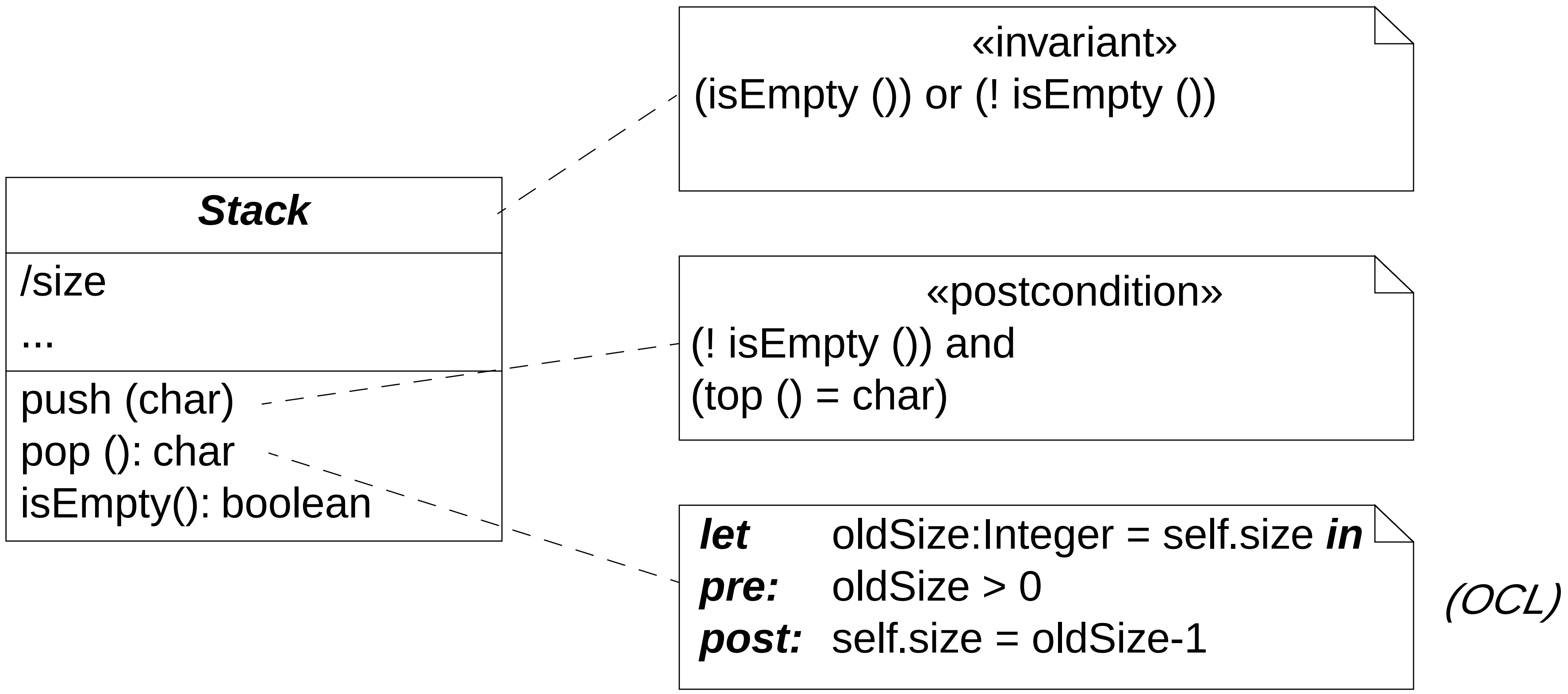


Constraints are specified between braces, either free or within a note:



Design by Contract in UML

Combine constraints with stereotypes:



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

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.

Summary

You should know the answers to these questions:

- ❑ 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?*

7. Modeling Behaviour

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

Sources:

- *Unified Modeling Language — Notation Guide*, version 1.1, Rational Software Corporation, 1997.
- *Object-Oriented Development — The Fusion Method*, D. Coleman, et al., Prentice Hall, 1994.

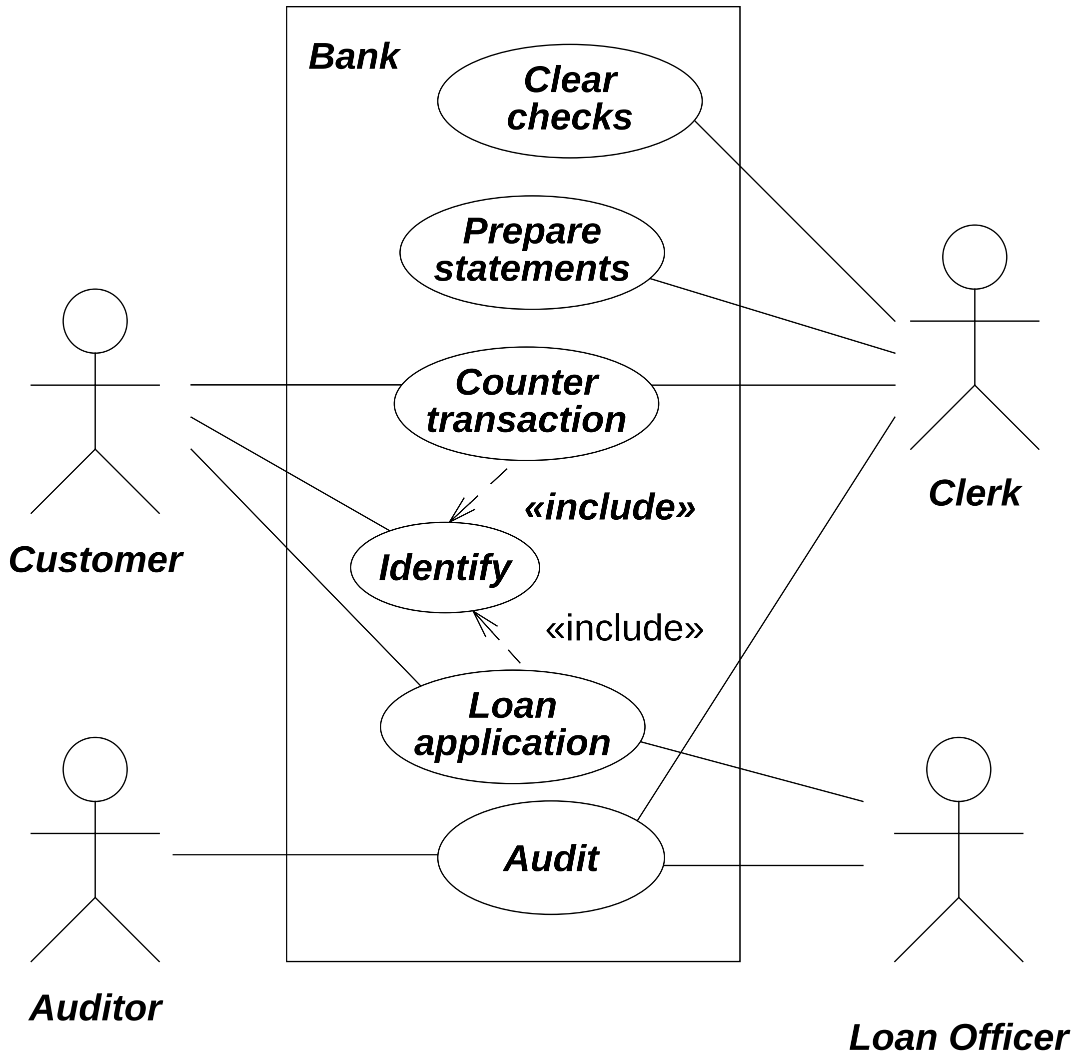
Use Case Diagrams

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.

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



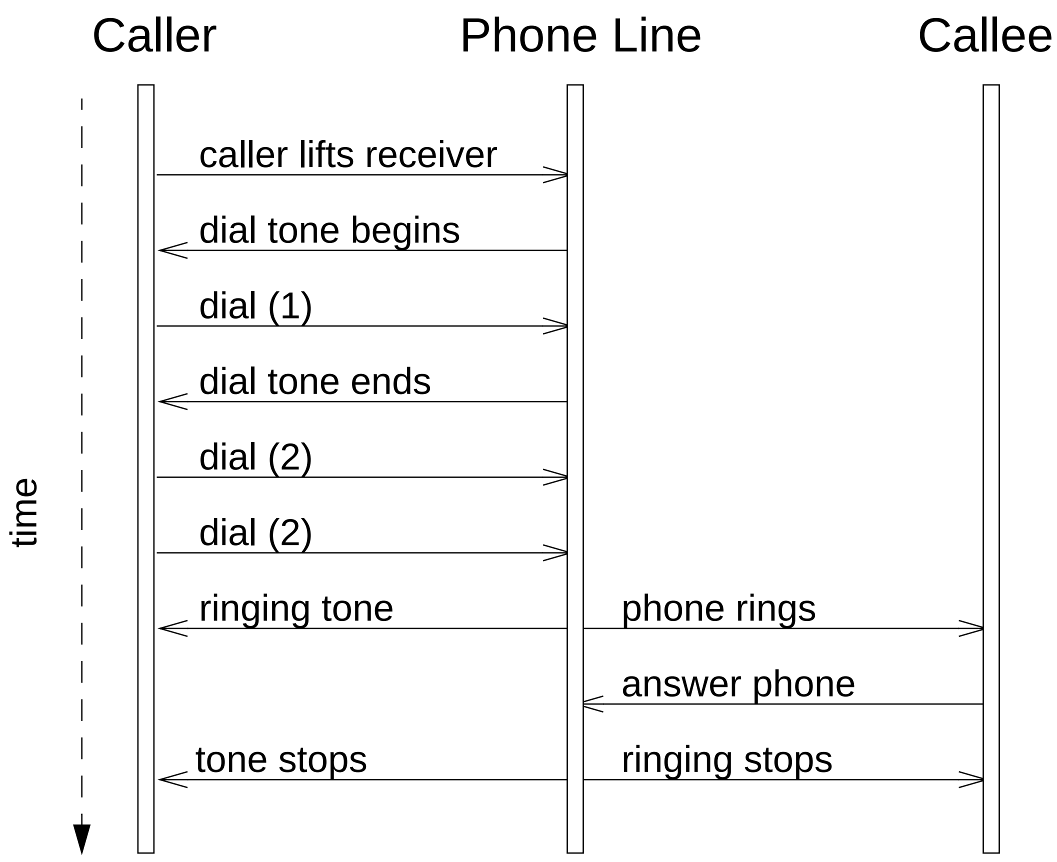
Sequence Diagrams

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

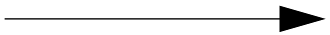
Objects (not classes!) are shown as vertical bars.

Events or message dispatches are shown as horizontal (or slanted) arrows from the send to the receiver.

Recall that a scenario describes a typical *example* of a use case, so conditionality is not expressed!

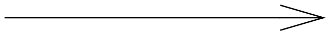


UML Message Flow Notation



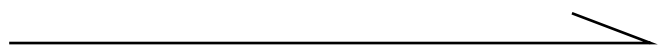
Filled solid arrowhead

procedure call or other nested control flow



Stick arrowhead

flat, sequential control flow (usually asynchronous)

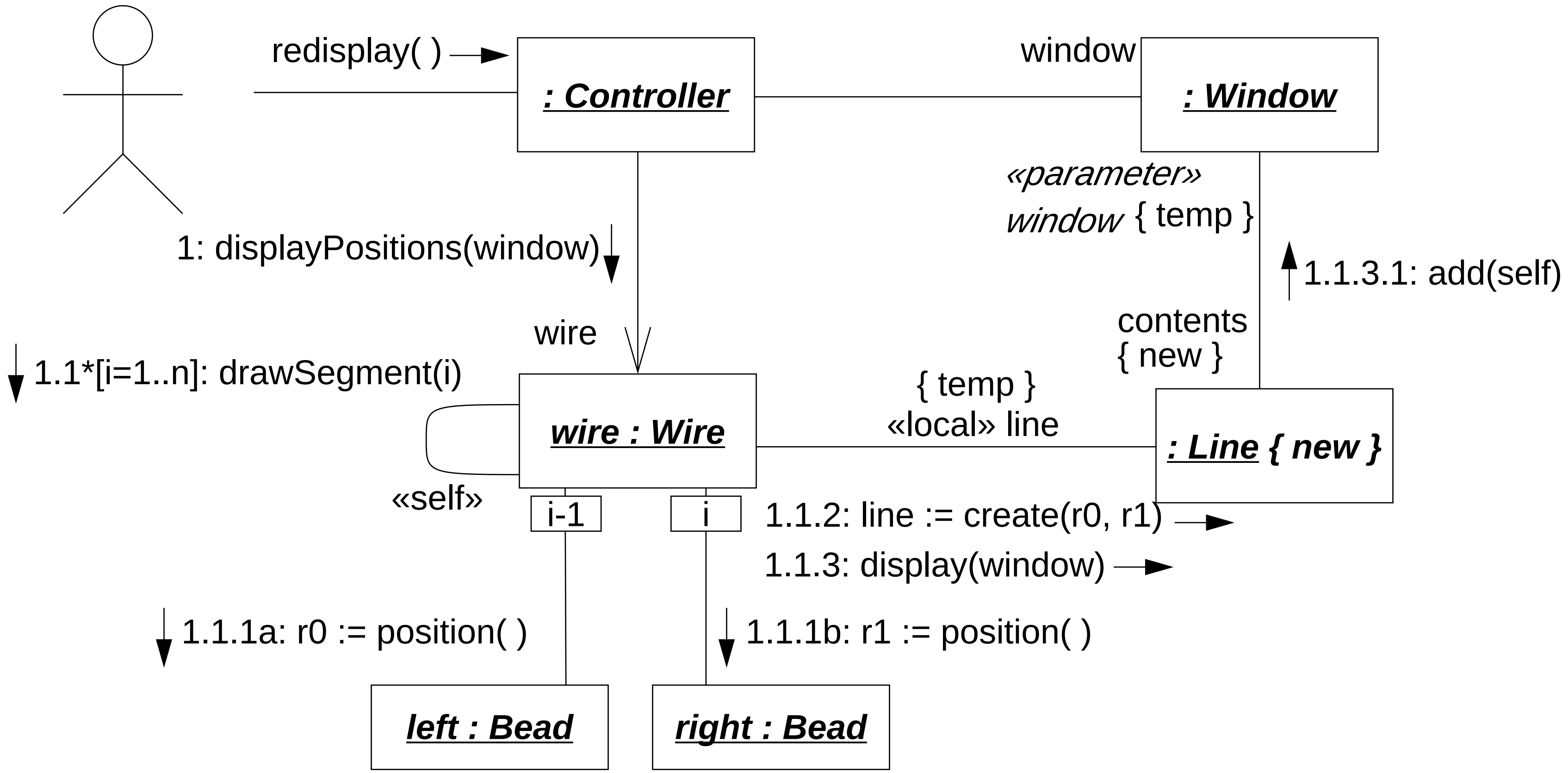


Half-stick arrowhead

asynchronous control flow between objects within a procedural sequence

Collaboration Diagrams

Collaboration diagrams depict scenarios as flows of messages between objects:



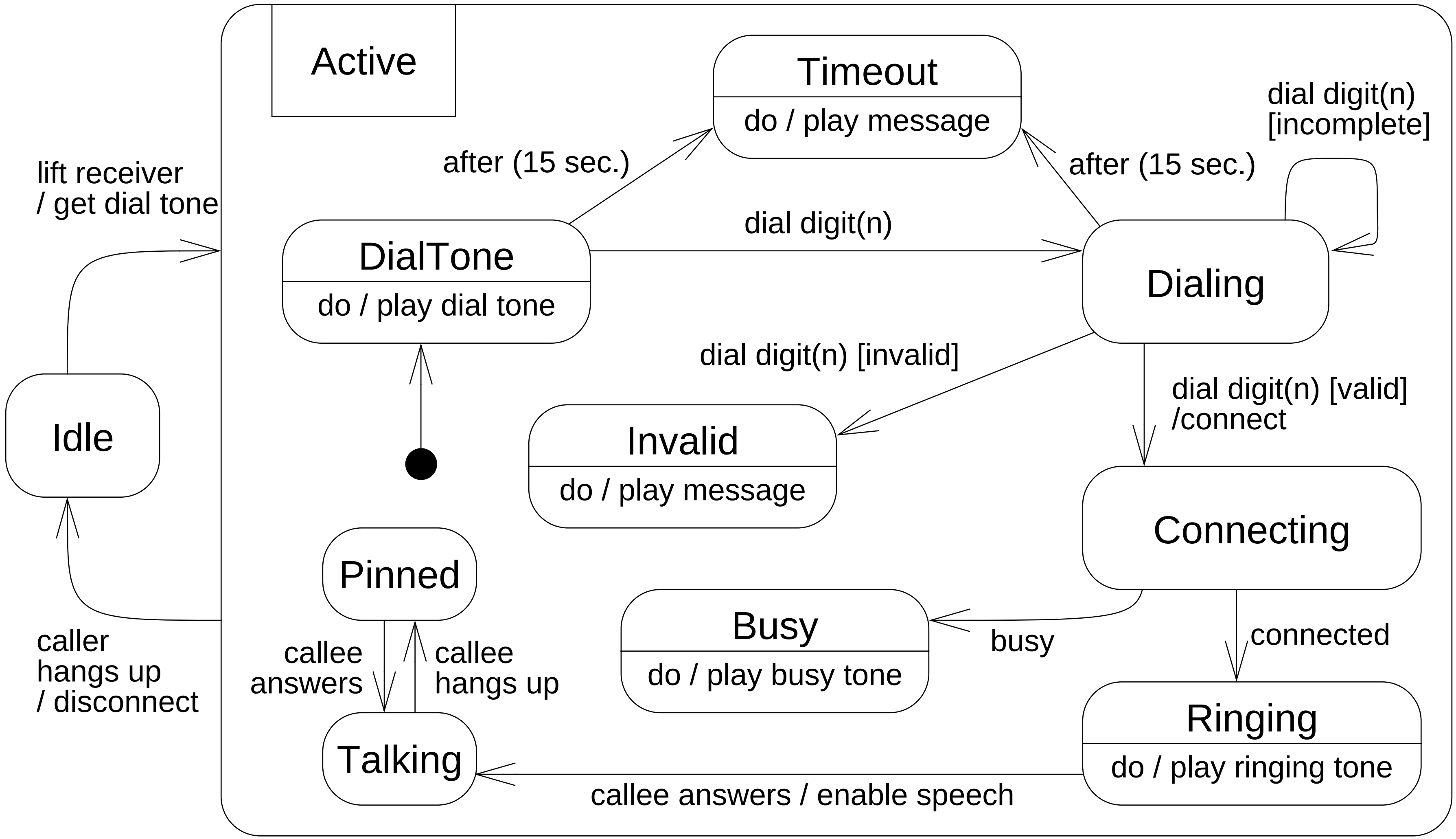
Message Labels

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

Message labels:

1. Prior messages from other threads (e.g. “[A1.3, B6.7.1]”)
   - ☞ only need 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]”)
3. Return value binding (e.g., “status :=”)
4. Message name
5. Argument list

State Diagrams



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.

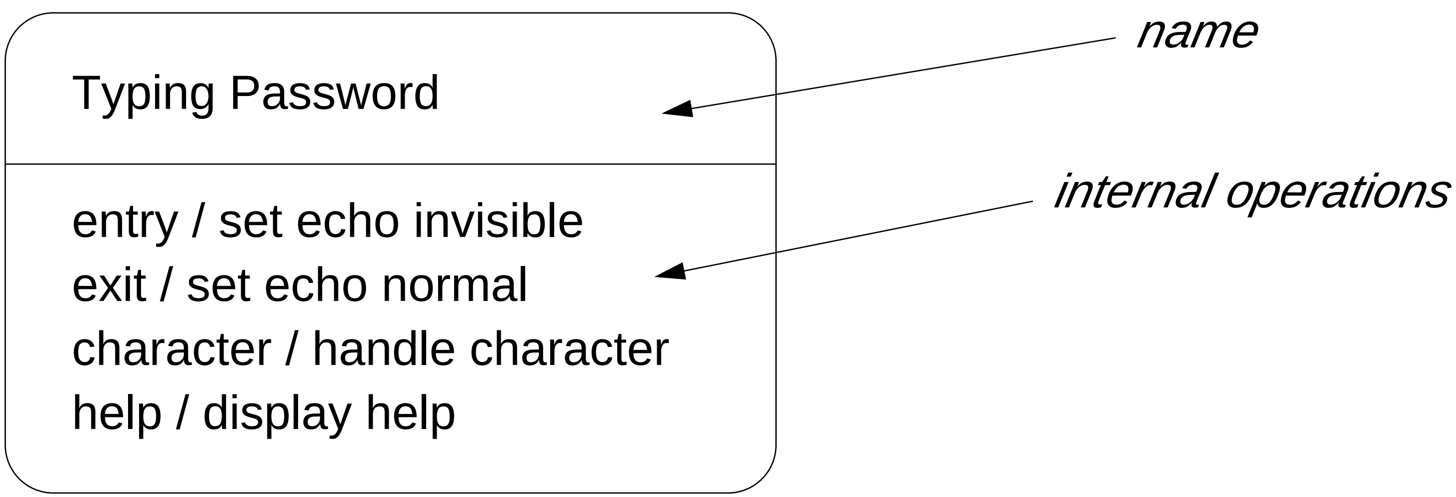
An *event* is a one-way (asynchronous) communication from one object to another:

- ❑ 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

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.

Transitions and Operations

Transitions:

- A response to an external event received by an object in a given state
- May invoke an operation, and cause object to change state
- May send an event to an external object
- Transition syntax (each part is optional):

	event (arguments) [condition]	event-signature plus guard
/	***^target.sendEvent operation (arguments)***	action-expression

- *External* transitions label arcs between states; *internal* transitions are part of the triggered operations of a state

Operations:

- Operations invoked by transitions are atomic *actions*
- *Entry* and *exit* operations can be associated with states

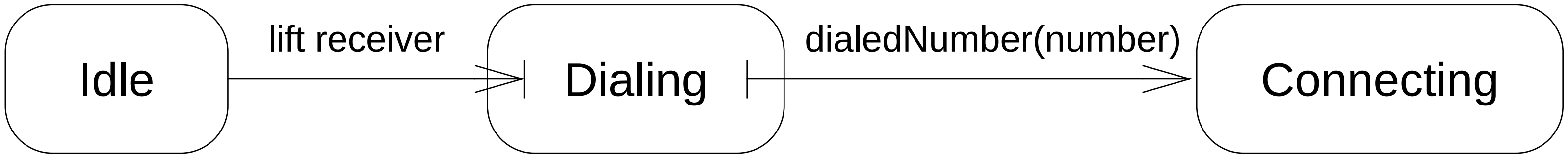
Activities:

- Ongoing operations while object is in a given state
- Modelled as internal transitions labelled with the pseudo-event ***do***

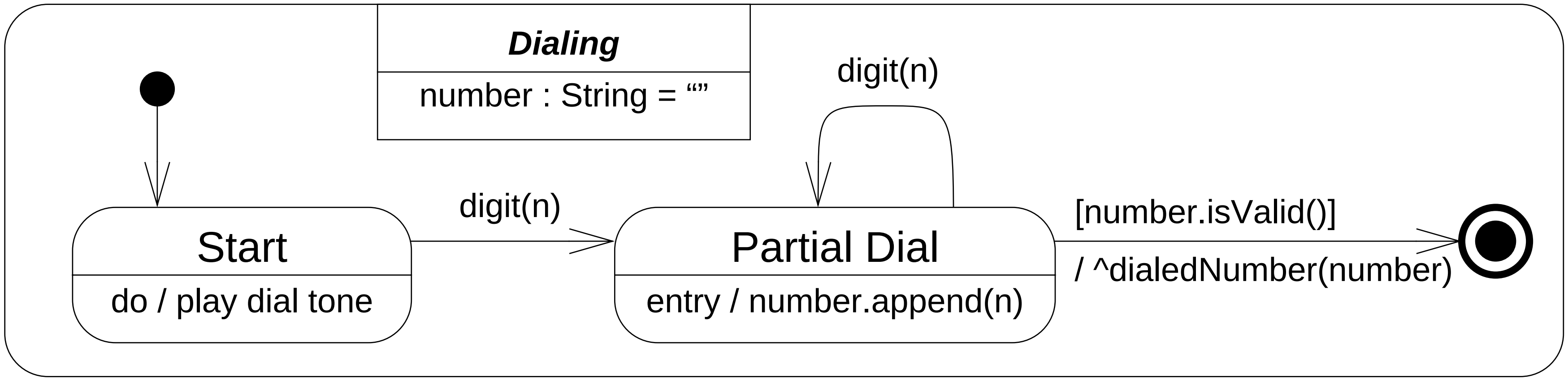
Composite States

Composite states may depicted either as high-level or low-level views.

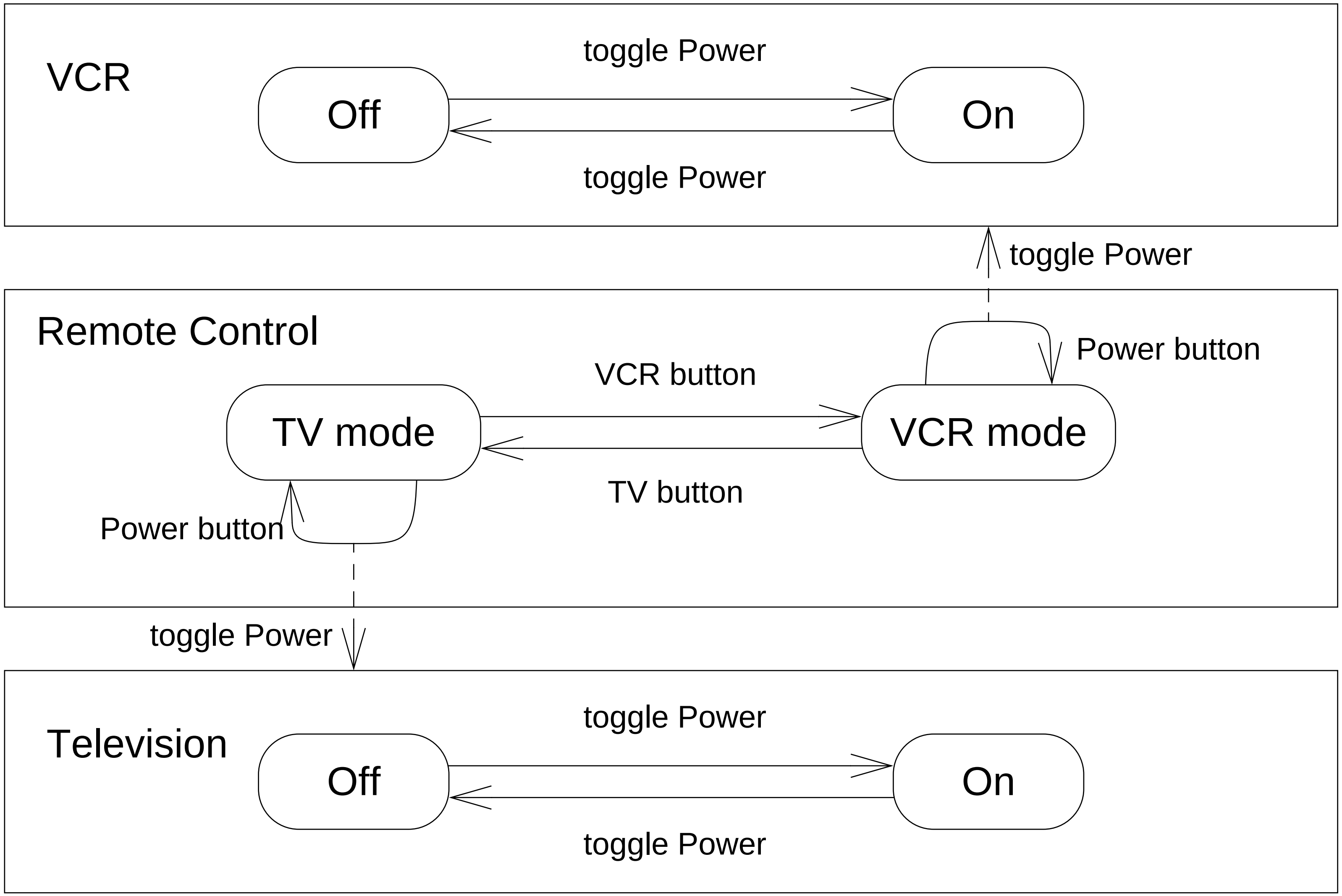
To indicate the presence of internal states, “stubbed transitions” may be used in the high-level view:



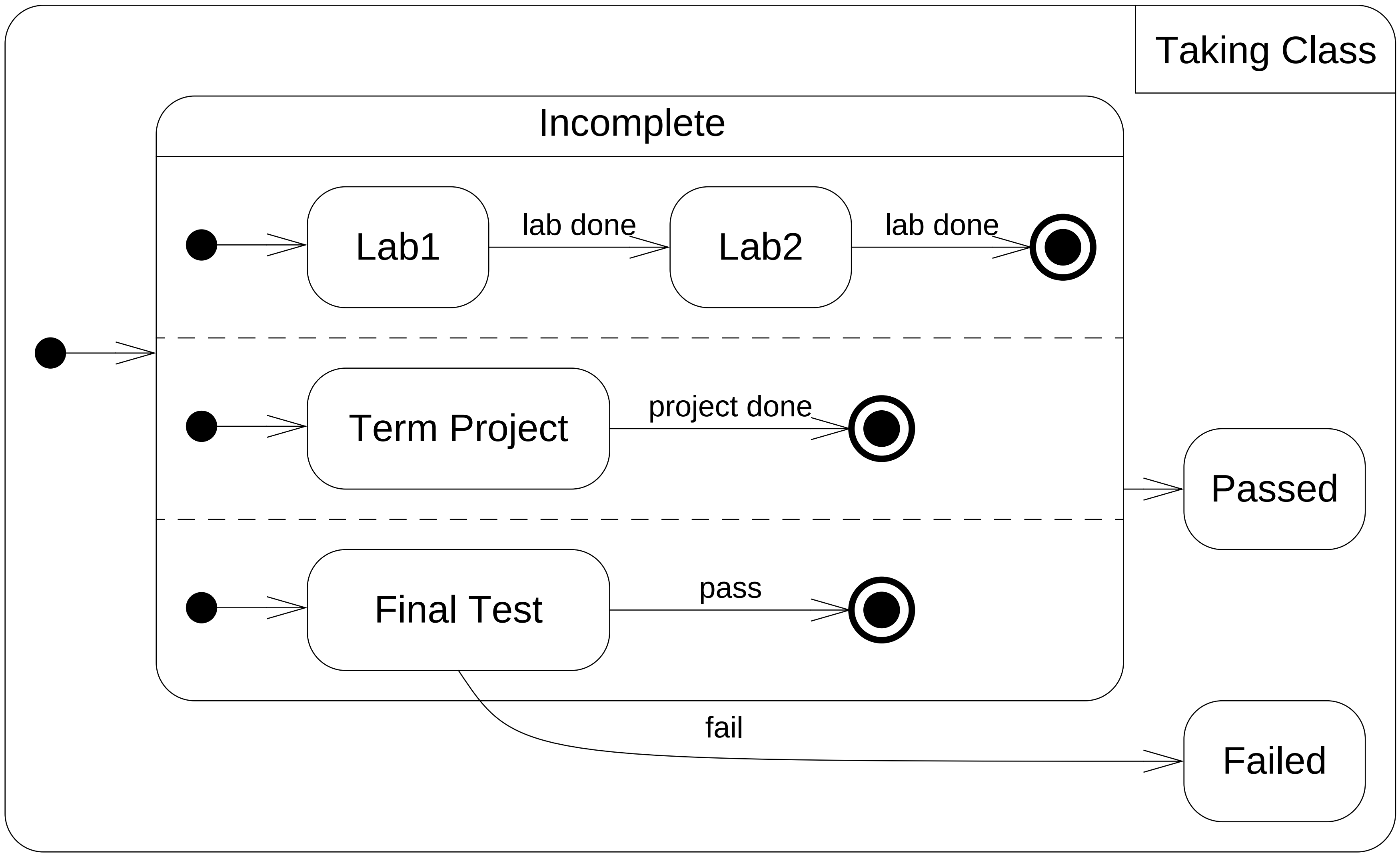
Starting and termination substates are shown as black spots and “bulls-eyes”:



Sending Events between Objects



Concurrent Substates



Branching and Merging

Entering concurrent states:

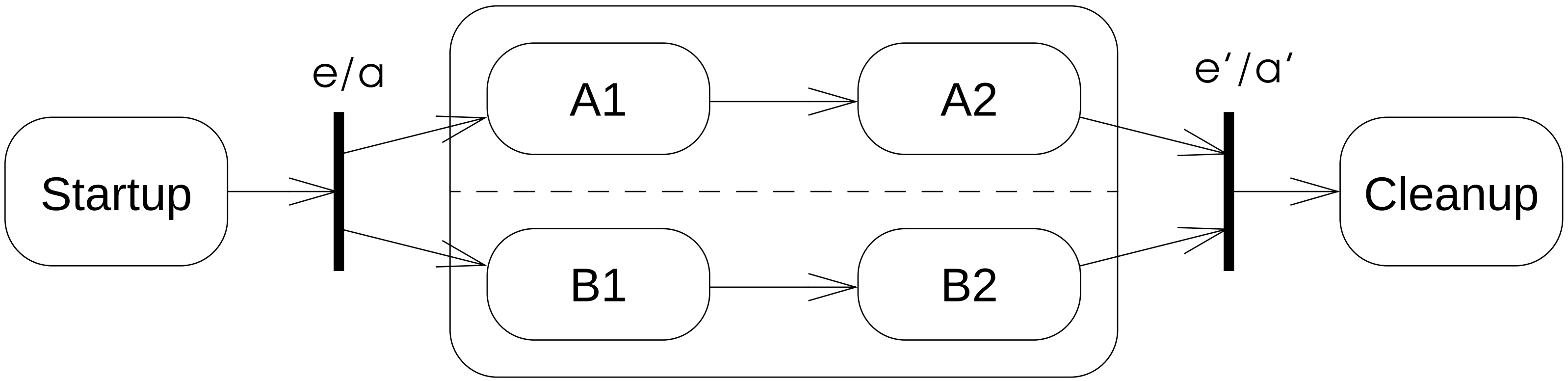
Entering a state with concurrent substates means that *each* of the substates is entered concurrently (one logical thread per substate).

Leaving concurrent states:

A labelled transition out of any of the substates terminates *all* of the substates.

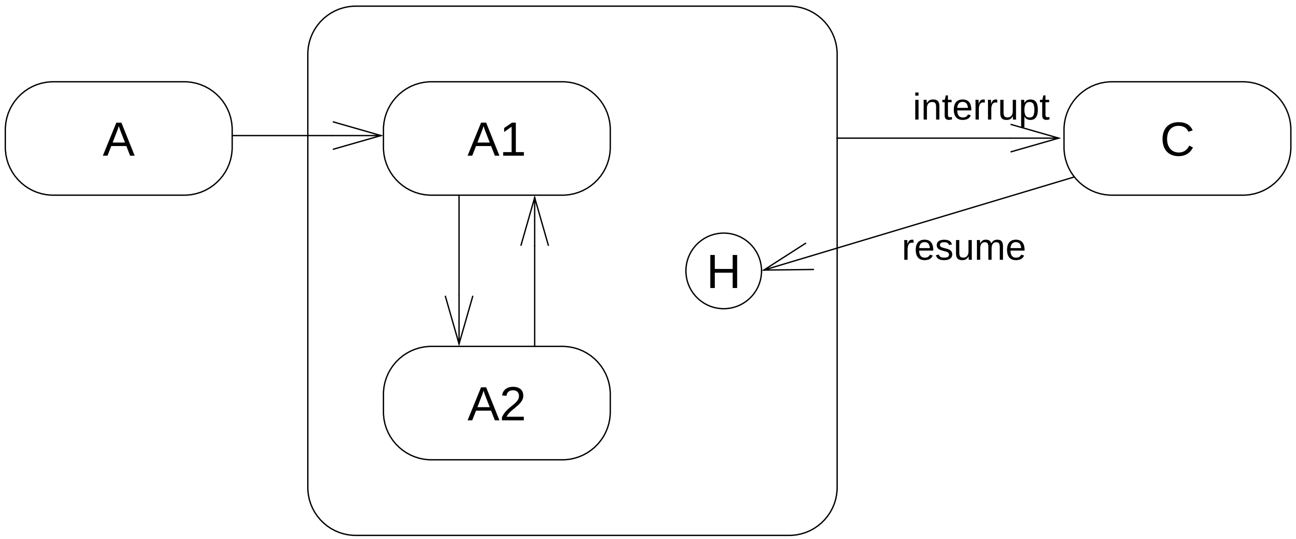
An unlabelled transition out of the overall state waits for all substates to terminate.

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



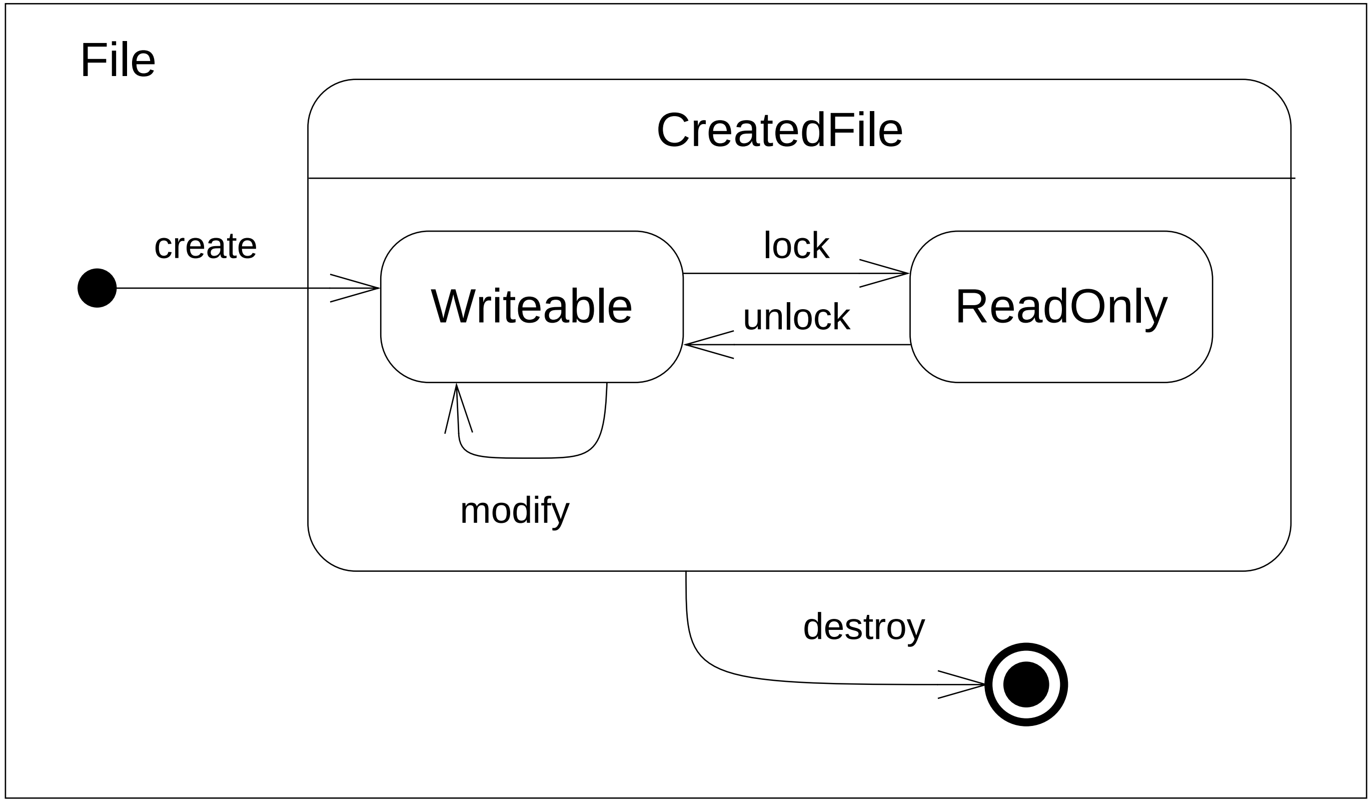
History Indicator

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:



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

Summary

You should know the answers to these questions:

- ❑ 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?

- ✎ *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?*

8. Software Architecture

Overview:

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

Sources:

- *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.
- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999
- *Pattern-Oriented Software Architecture — A System of Patterns*, F. Buschmann, et al., John Wiley, 1996
- *Software Architecture: Perspectives on an Emerging Discipline*, M. Shaw, D. Garlan, Prentice-Hall, 1996

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.

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.

- 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

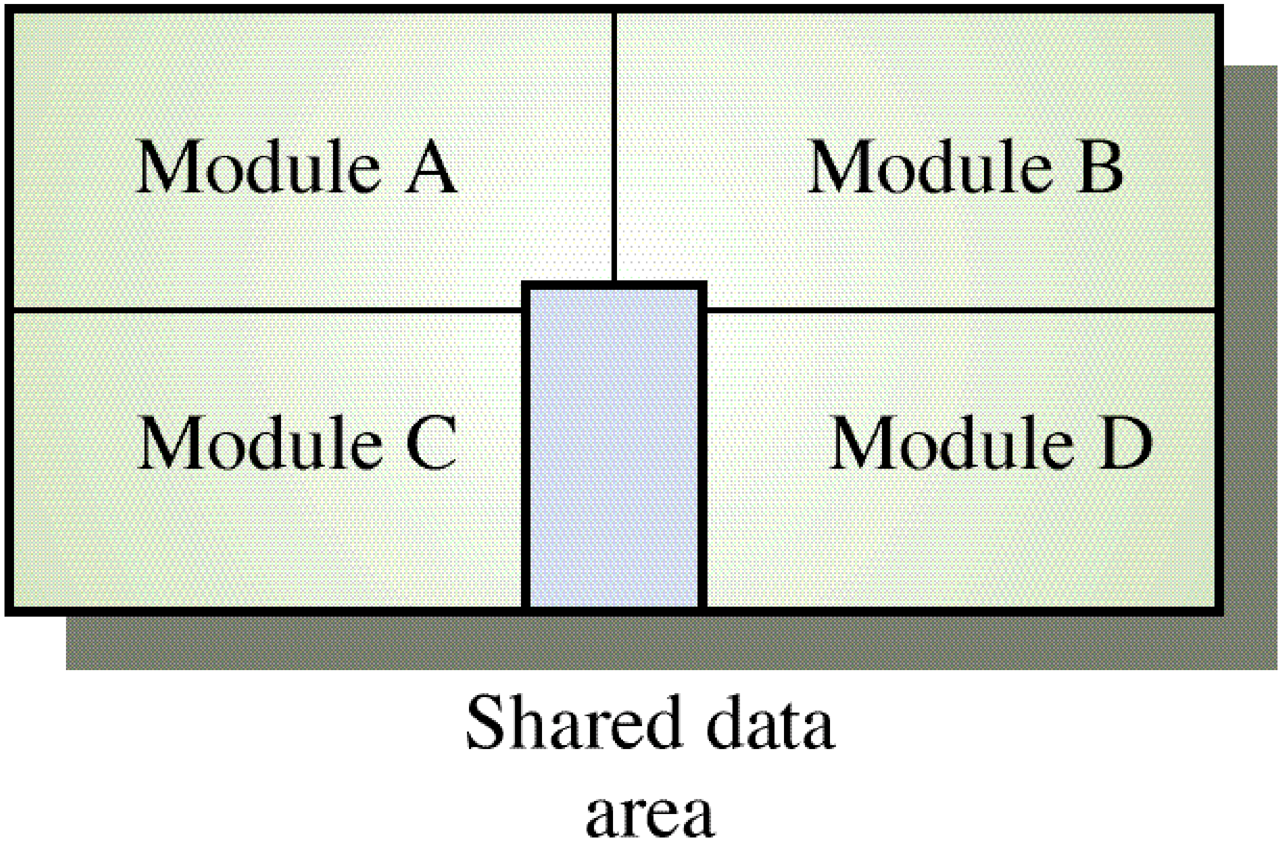
Coupling is a measure of the *strength of the interconnections* between system components.

Coupling is *tight* between components if they depend heavily on one another, (e.g., there is a lot of communication between them).

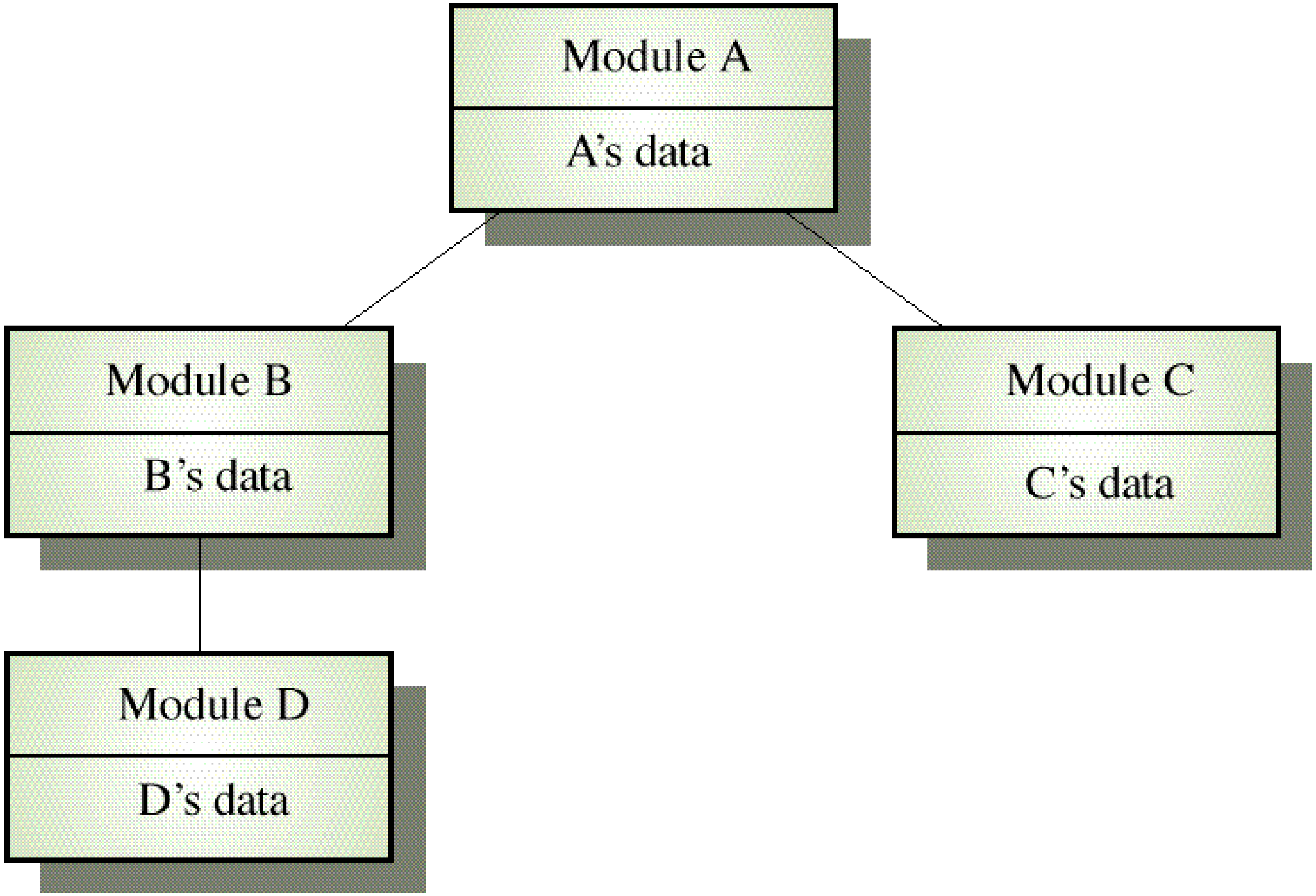
Coupling is *loose* if there are few dependencies between components.

Loose coupling *promotes maintainability* and *adaptability* since changes in one component are less likely to affect other ones.

Tight Coupling



Loose Coupling



Architectual 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

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.

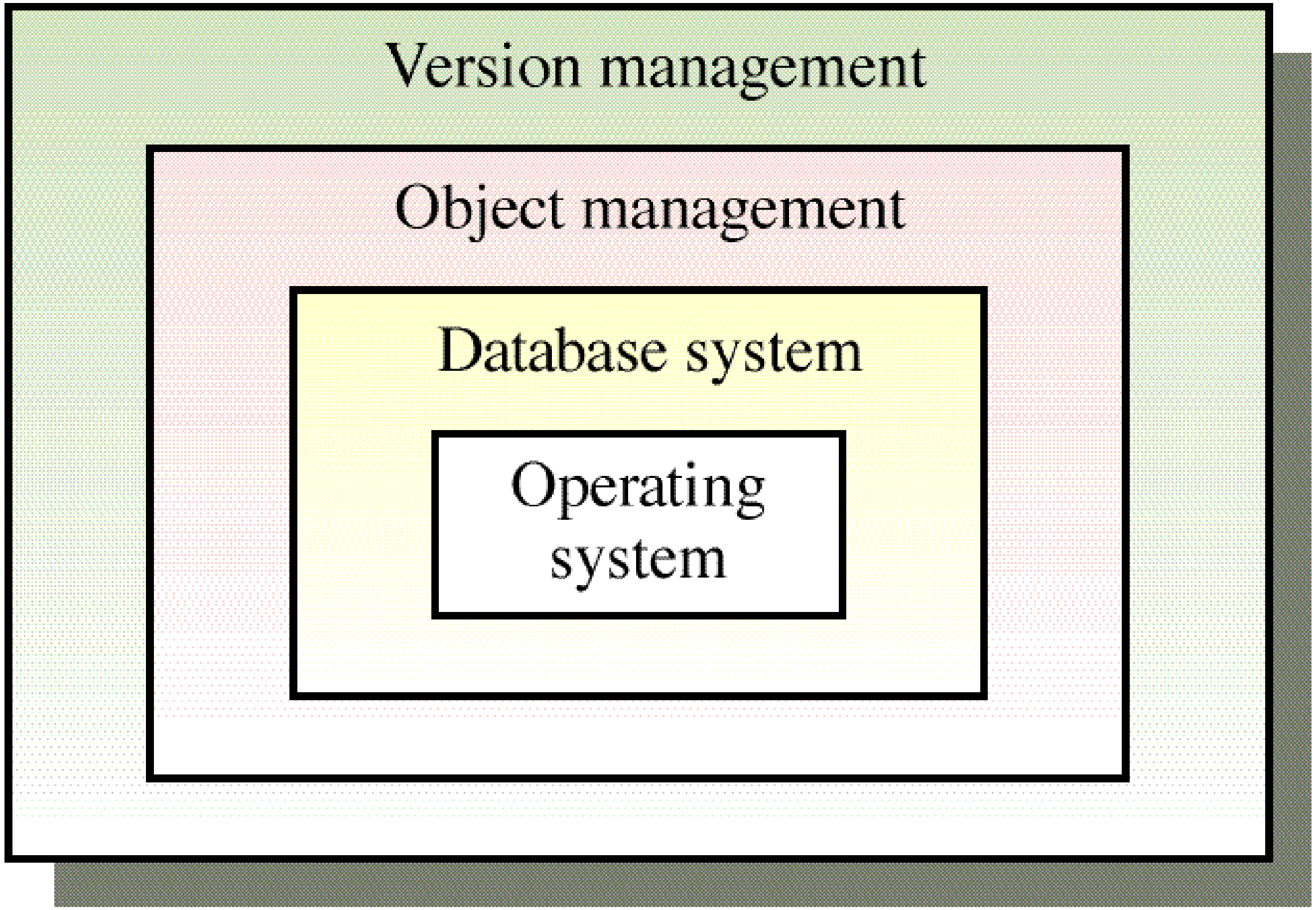
— Shaw and Garlan

Layered Architectures

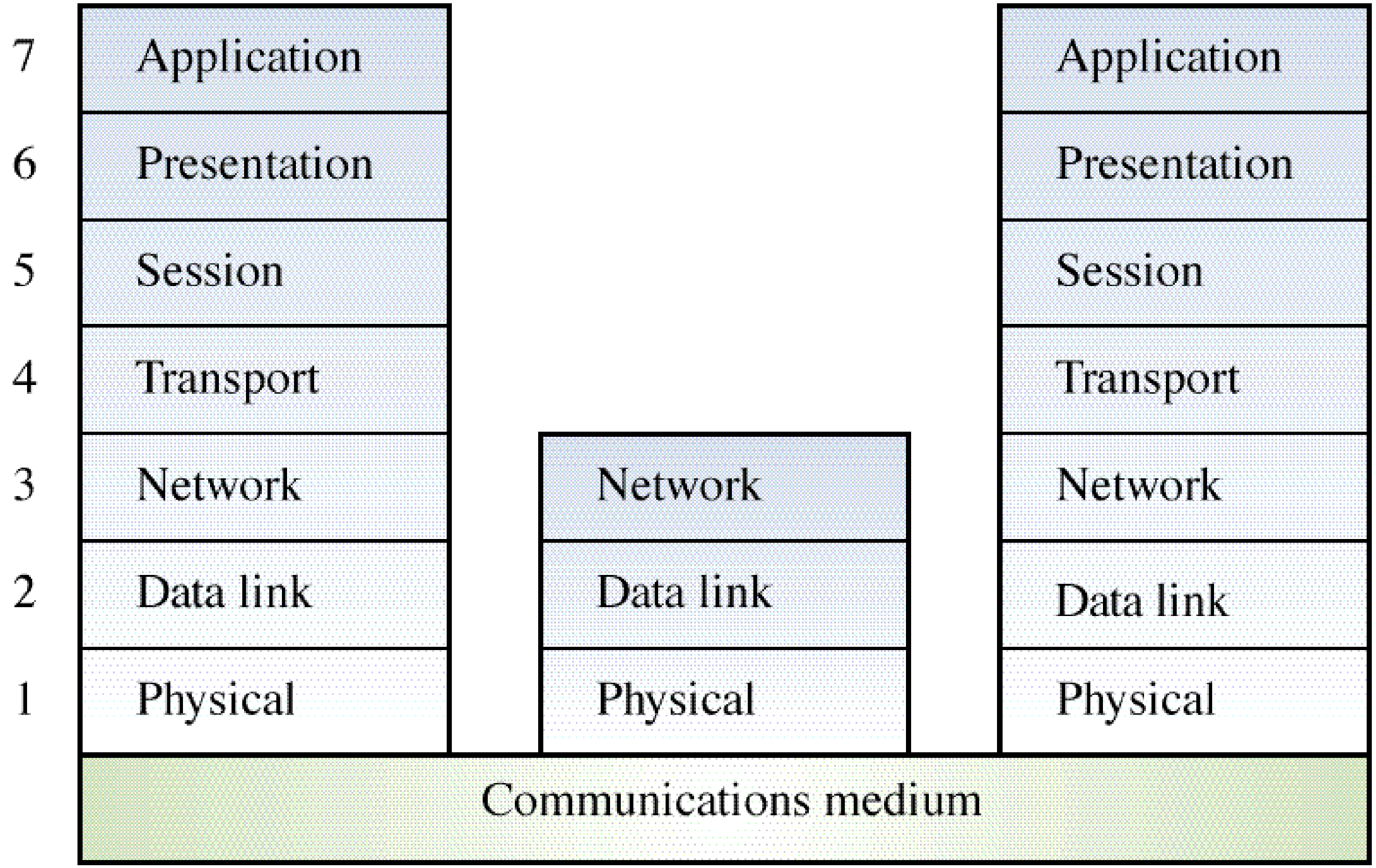
A layered architecture organises a system into a set of layers each of which provide a set of services to the layer “above.”

- ❑ 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



OSI Reference Model



Client-Server Architectures

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

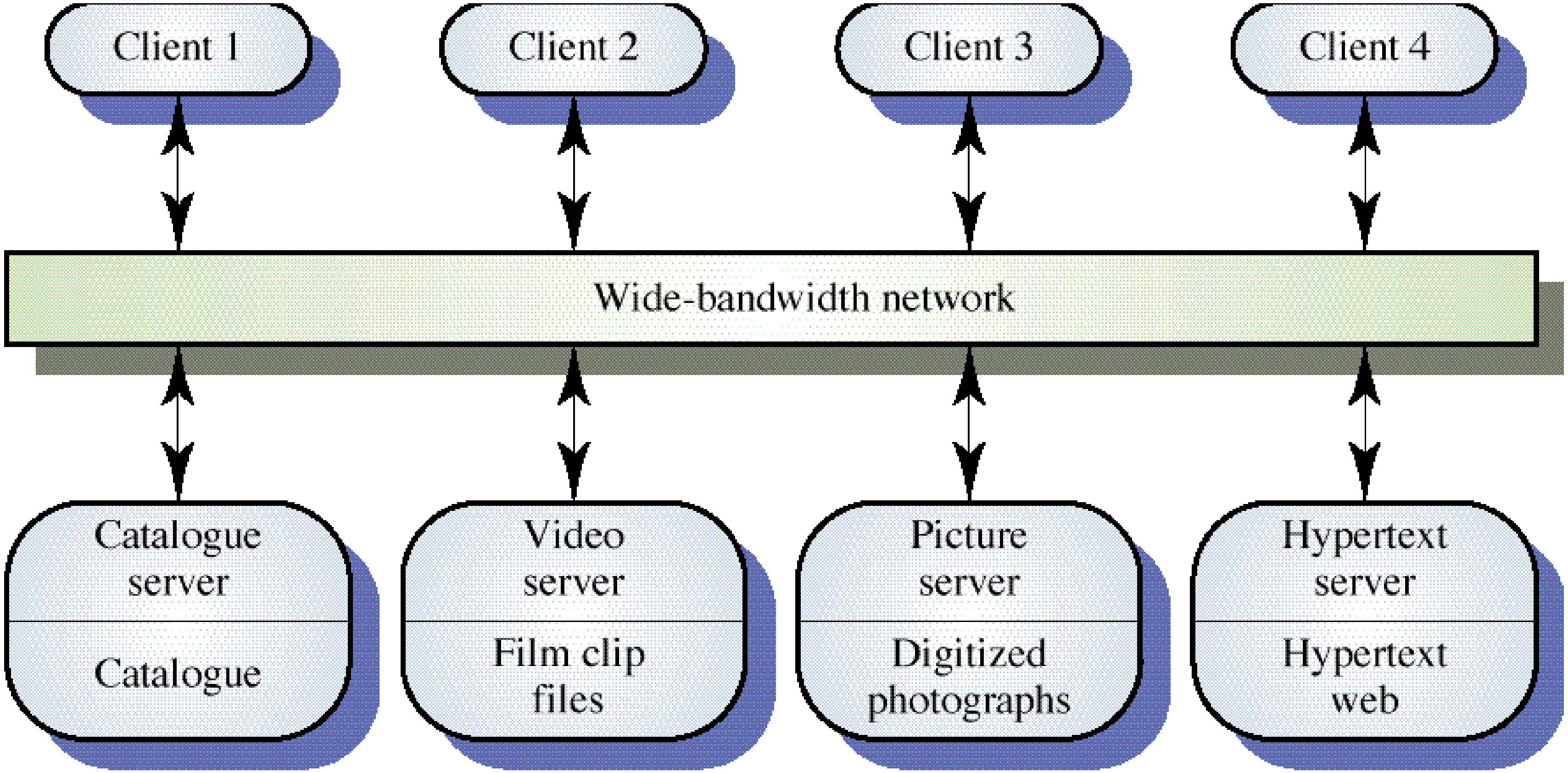
Advantages

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

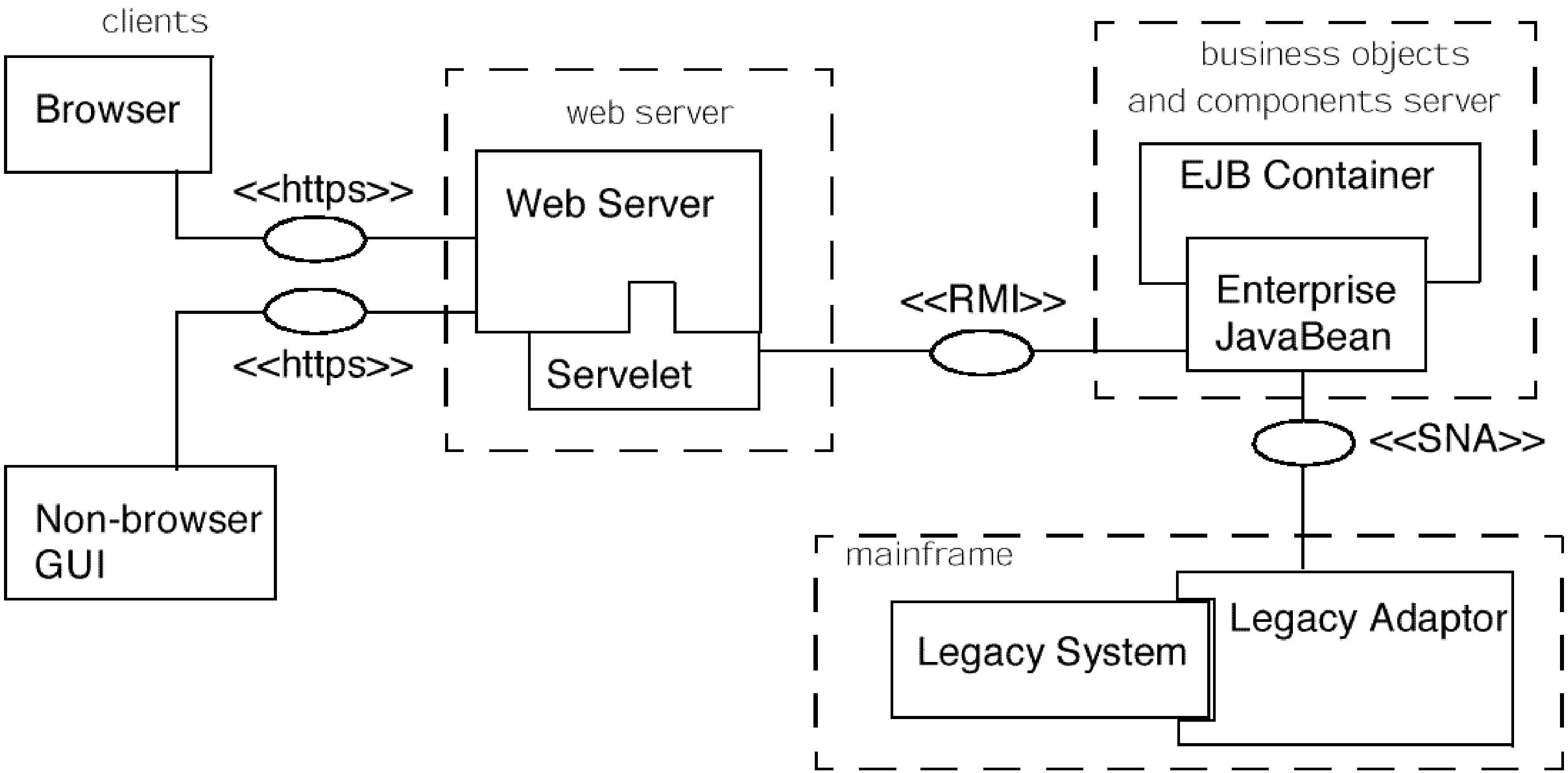
Disadvantages

- ❑ 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 register of names and services — it may be hard to find out what servers and services are available

Client-Server Architectures



Four-Tier Architectures



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”).

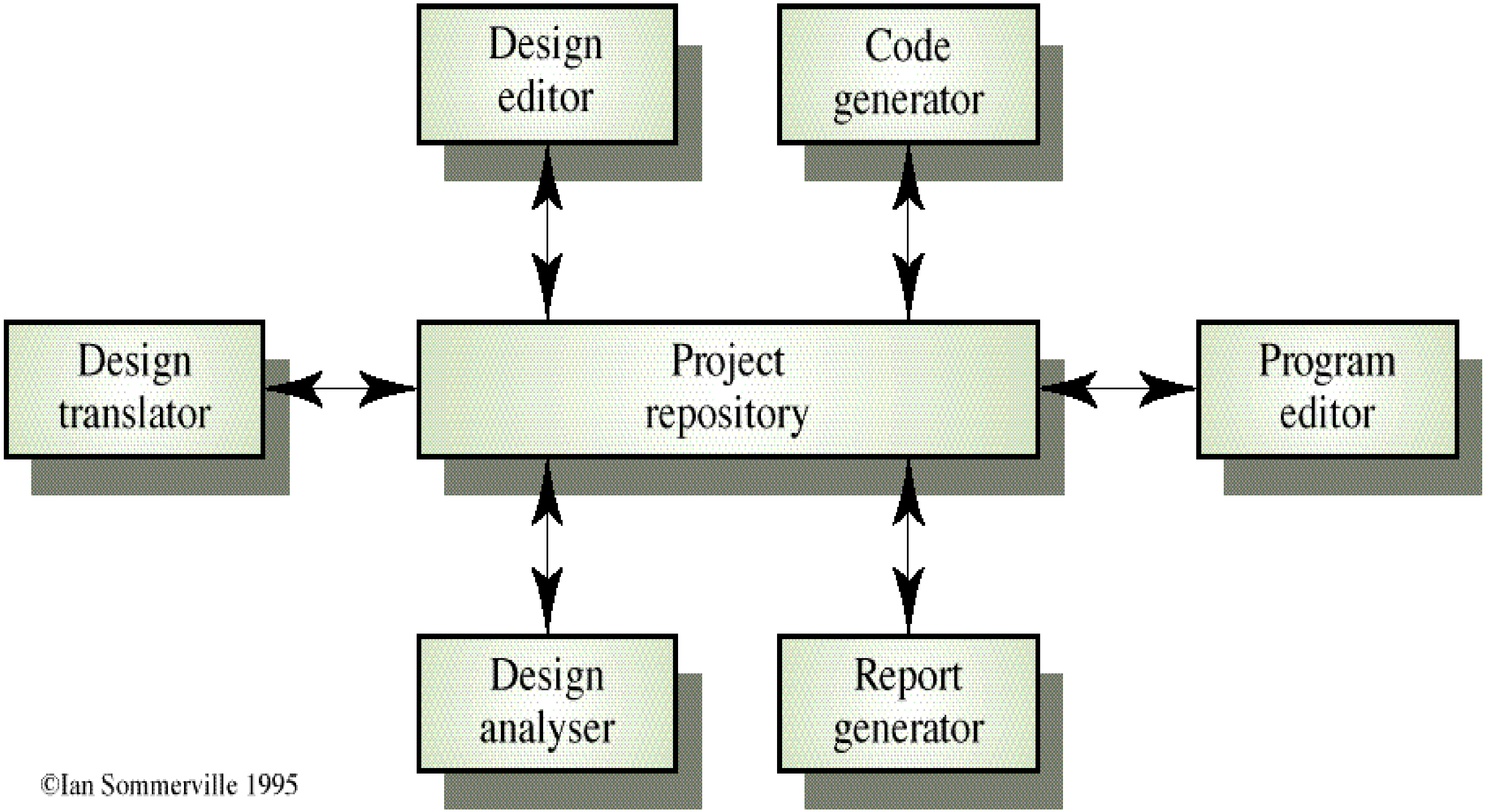
Advantages

- 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

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



Event-driven Systems

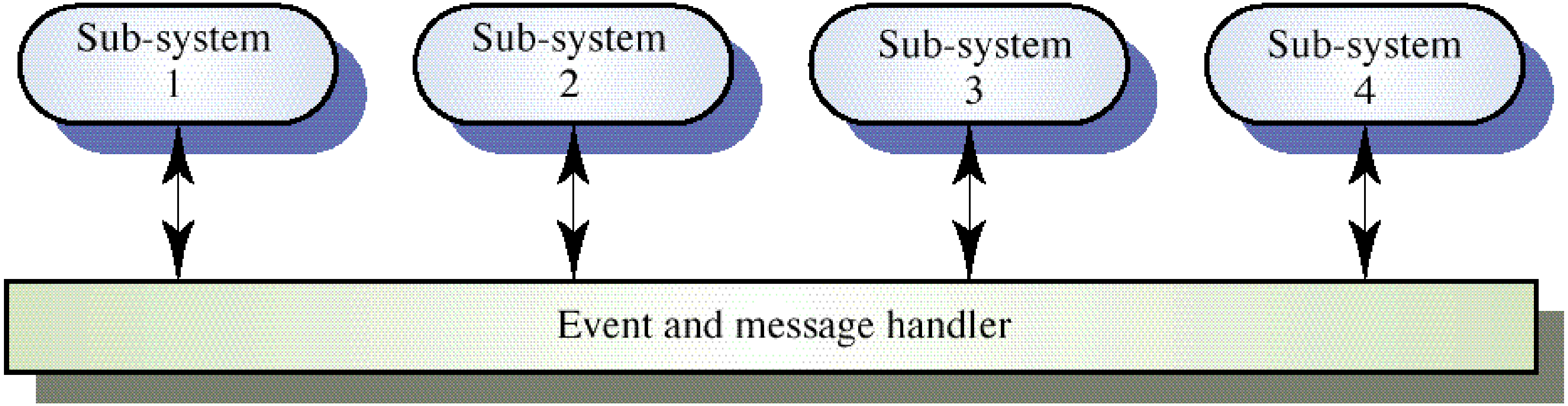
In an event-driven architecture components perform services in reaction to *external events* generated by other components.

- In *broadcast* models an event is broadcast to all sub-systems. 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



Dataflow Models

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

- ❑ Dataflows should be free of cycles
- ❑ The single-input, single-output variant is known as *pipes and filters*
  - ☞ e.g., UNIX (Bourne) shell

```
tar cf - . | gzip -9 | rsh picasso dd
```

data source ***filter*** ***data sink***

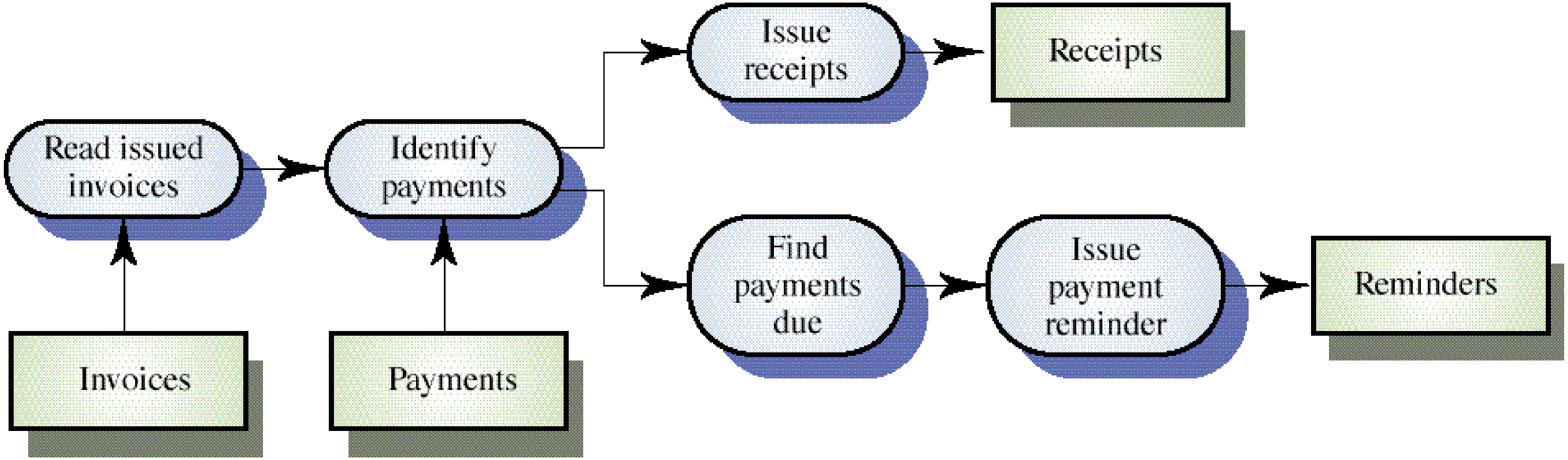
  - ☞ e.g., CGI Scripts for interactive Web-content

```
HTML Form    CGI Script   generated HTML page
```

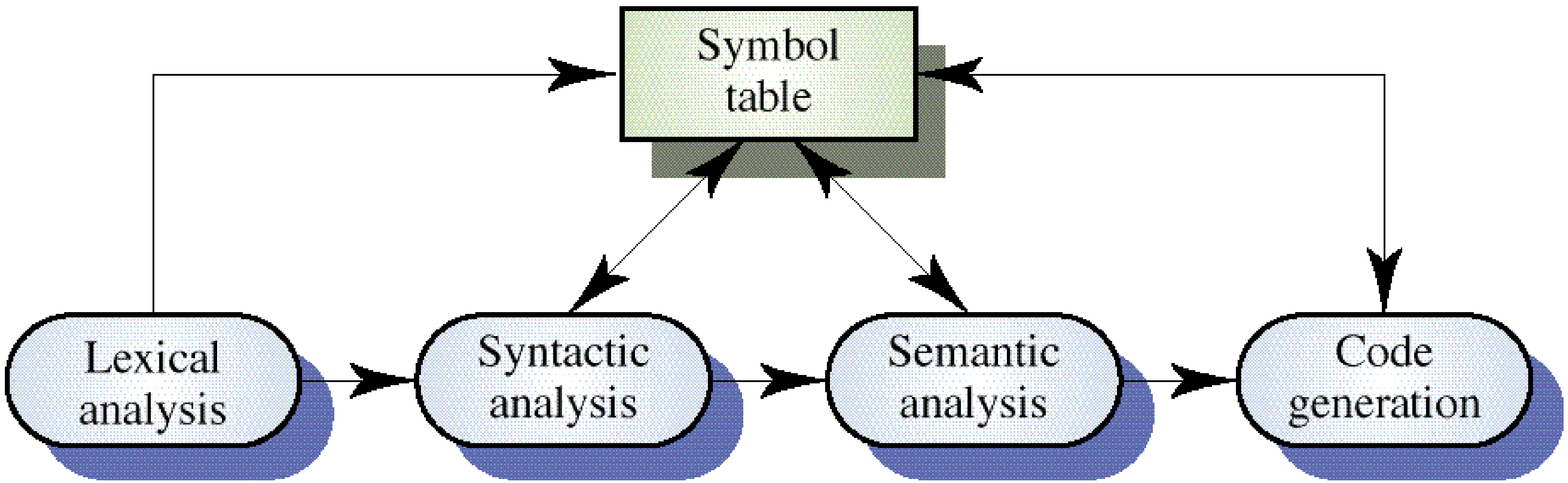
data source ***filter*** ***data sink***

- ❑ Not really suitable for interactive systems

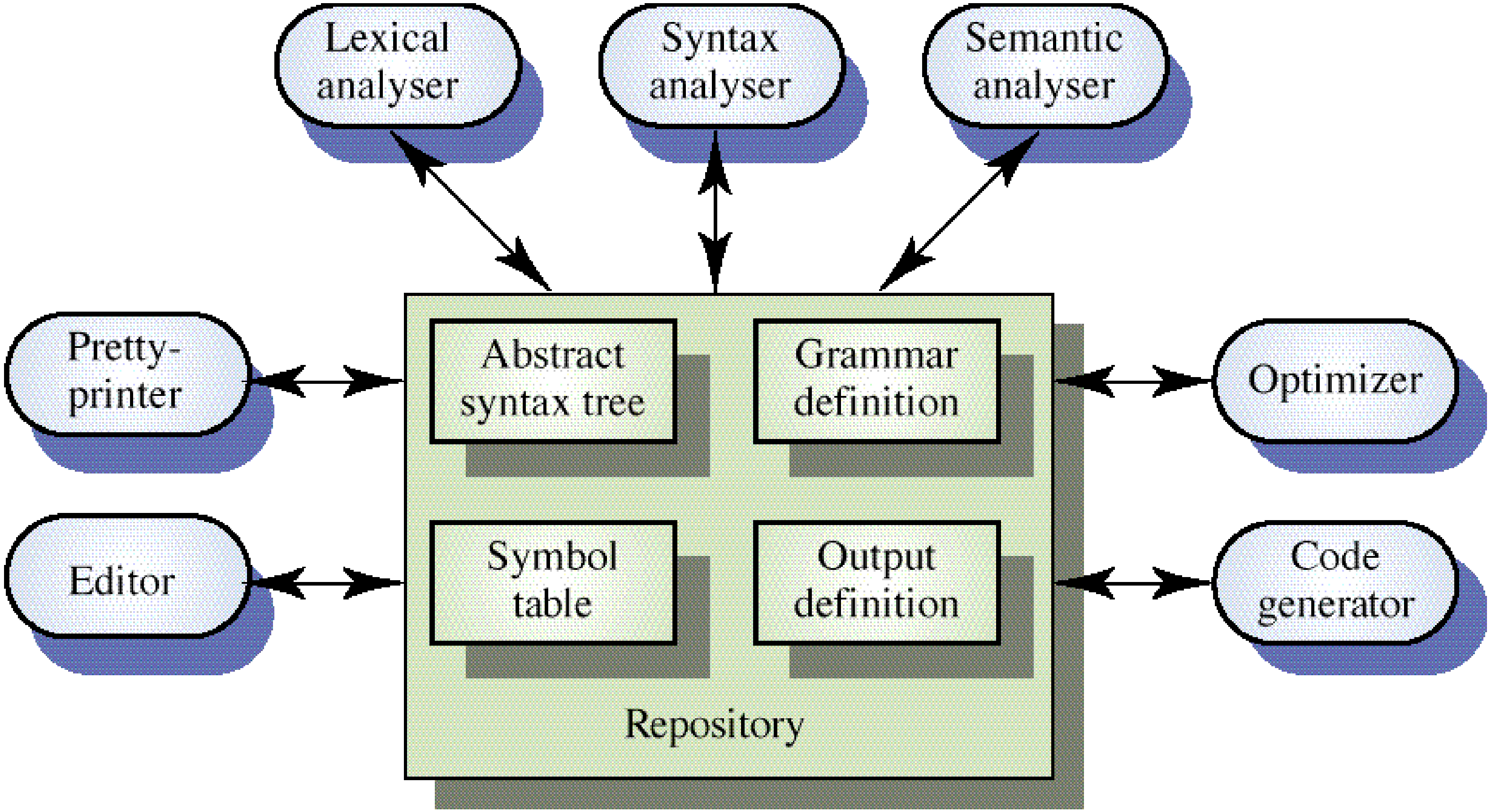
Invoice Processing System



Compilers as Dataflow Architectures

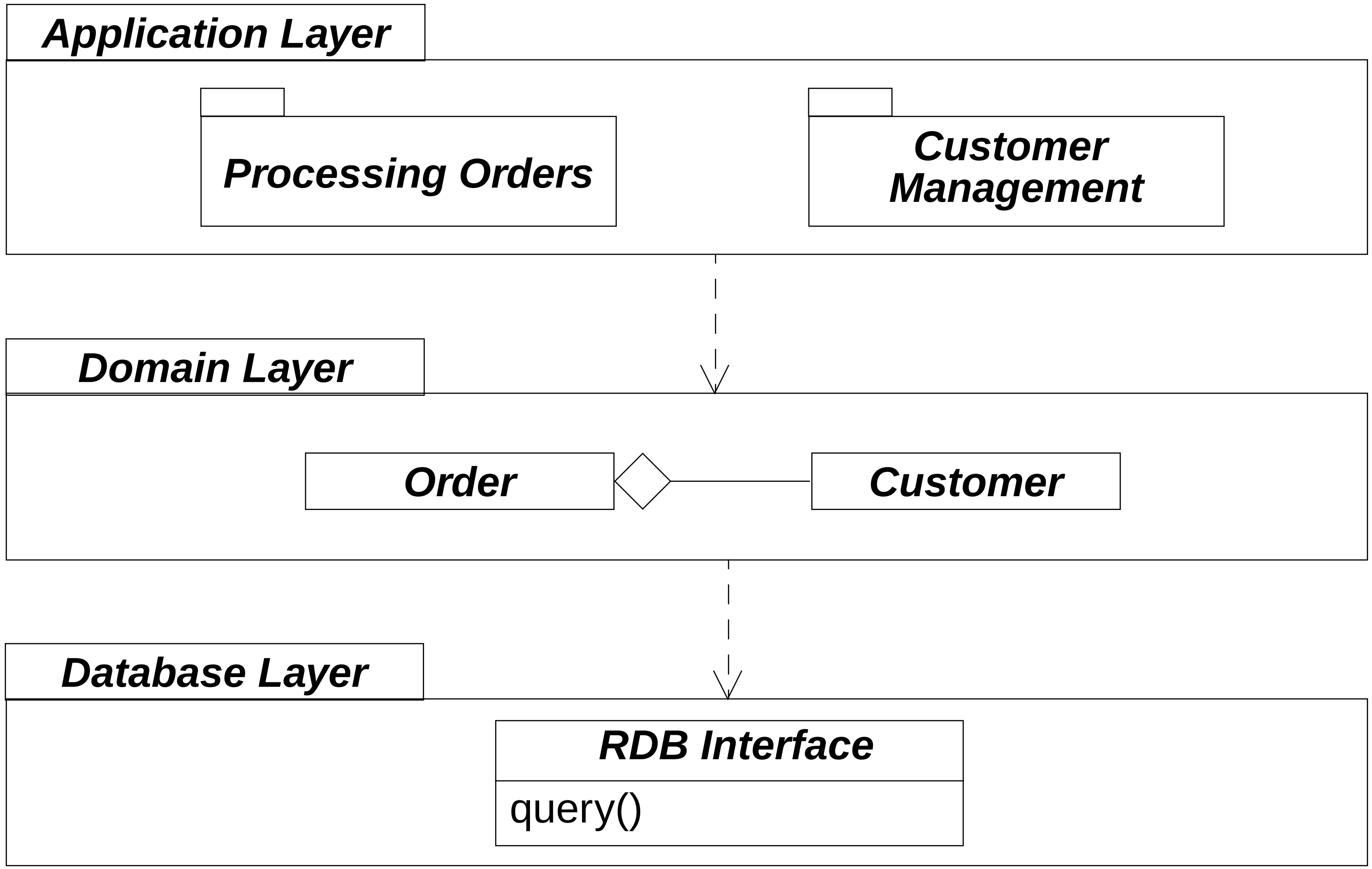


Compilers as Blackboard Architectures



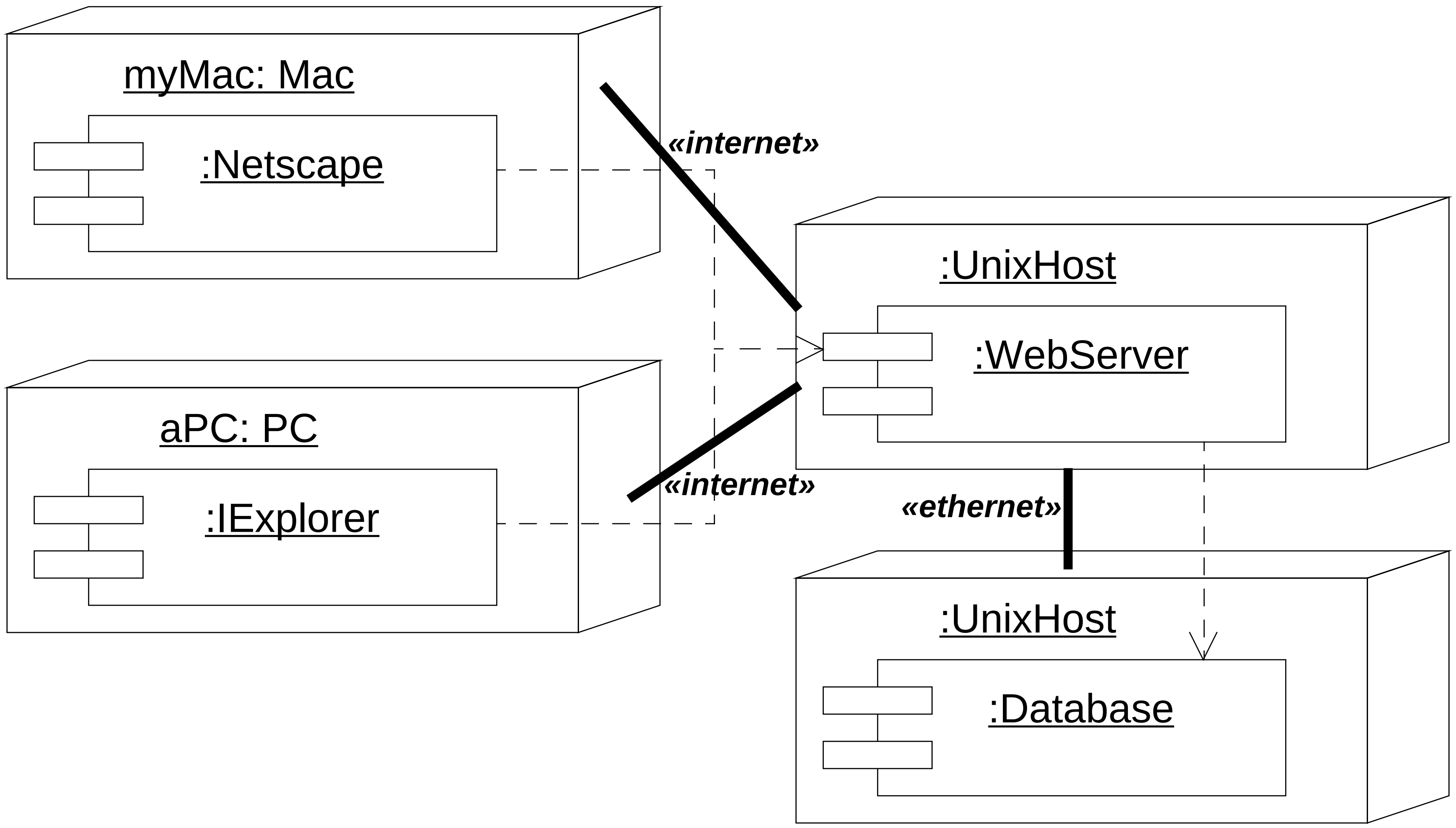
UML: Package Diagram

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



UML: Deployment Diagram

Shows physical lay-out of run-time components on hardware nodes.



Summary

You should know the answers to these questions:

- ❑ 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?*

9. User Interface Design

Overview:

- ❑ Interface design models
- ❑ Design principles
- ❑ Information presentation
- ❑ User Guidance
- ❑ Evaluation

Sources:

- ❑ *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.
- ❑ *Software Engineering — A Practitioner's Approach*, R. Pressman, Mc-Graw Hill, Third Edn., 1994.

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.)

GUI Characteristics

Characteristic	*Description*
Windows	Multiple windows allow *different information* to be displayed *simultaneously* on the user's screen.
Icons	Usually icons represent *files* (including folders and applications), but they may also stand for *processes* (e.g., printer drivers).
Menus	Menus bundle and organize *commands* (eliminating the need for a command language).
Pointing	A pointing device such as a mouse is used for *selecting* choices from a menu or indicating items of interest in a window.
Graphics	Graphical elements can be *mixed with text* on the same display.

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

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

User Interface Design Principles

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

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 editors

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.

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

Advantages

- Allow experienced users to *interact quickly* with the system
- Commands can be *scripted*

Problems

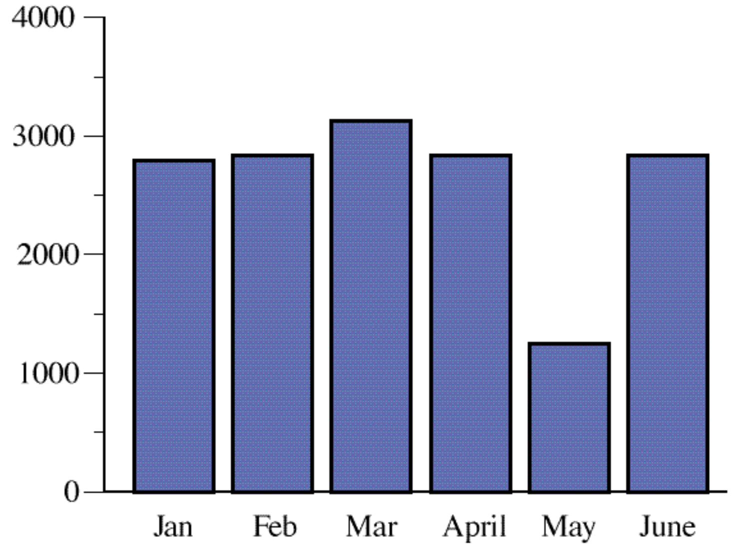
- Users have to *learn and remember* a command language
- Not suitable for *occasional* or inexperienced users
- An *error detection* and recovery system is required
- *Typing* ability is required

Information Presentation

Information display 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?

Jan	Feb	Mar	April	May	June
2842	2851	3164	2789	1273	2835



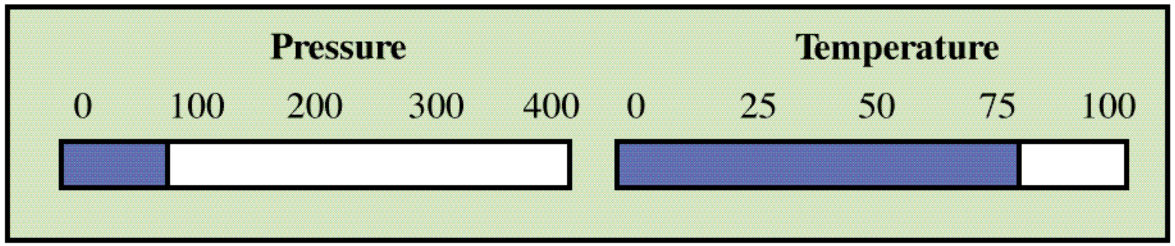
Analogue vs. Digital Presentation

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



Colour Displays

Colour can help the user *understand complex information structures.*

Colour use guidelines

- *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*

<u>User Guidance</u>

The <u>user guidance system</u> 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

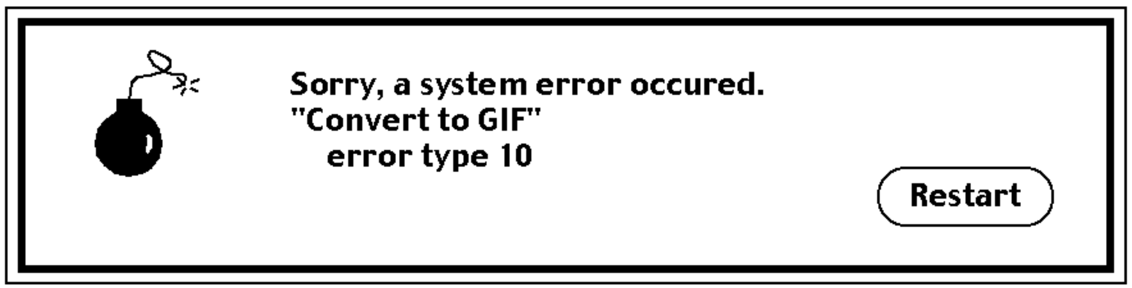
Design Factors in Message Wording

Context	The user guidance system should be aware of what the user is doing and should *adjust the output message to the current context.*
Experience	The user guidance system should provide both longer, *explanatory messages for beginners*, and more *terse messages for experienced users*.
Skill level	Messages should be *tailored to the user's skills* as well as their experience. I.e., depending on the *terminology* which is familiar to the reader.
Style	Messages should be *positive rather than negative.* They should never be insulting or try to be funny.
Culture	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.

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



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

Attribute	*Description*
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user's work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

Summary

You should know the answers to these questions:

- ❑ 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 every 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?*

10. Software Validation

Overview:

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

Source:

- *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.

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.

Failure class	*Description*
Transient	Occurs only with certain inputs
Permanent	Occurs with all inputs
Recoverable	System can recover without operator intervention
Unrecoverable	Operator intervention is needed to recover from failure
Non-corrupting	Failure does not corrupt data
Corrupting	Failure corrupts system data

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 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
- *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!)

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.

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

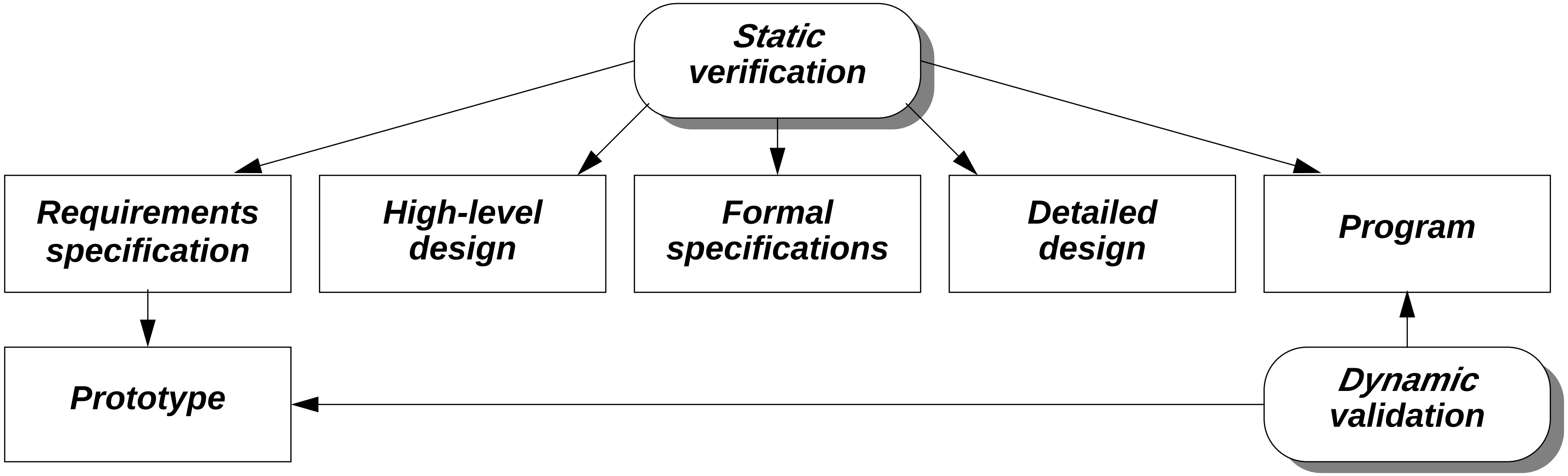
Verification and Validation

Validation:

- ❑ Are we building the right product?

Verification:

- ❑ Are we building the product right?



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 sub-systems, 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!

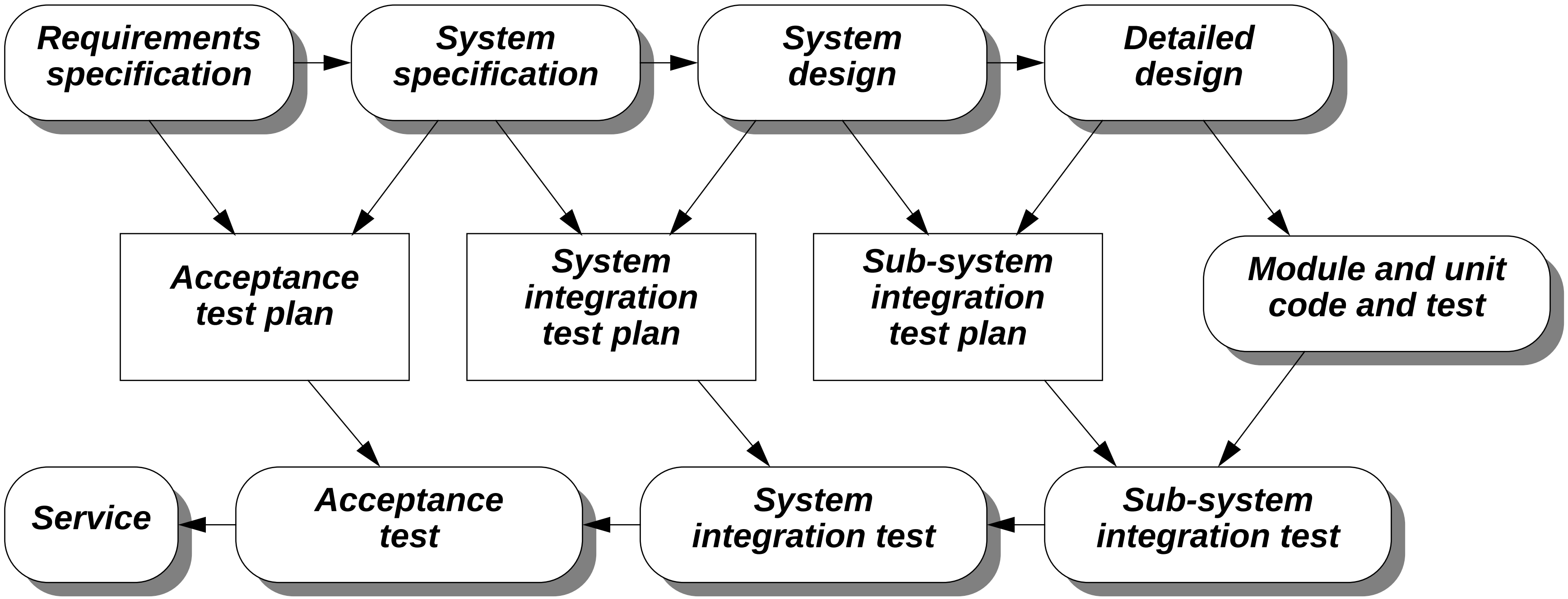
- tests must be *deterministic* and *repeatable*
- should test “all” functionality
  - every interface
  - all boundary situations
  - every feature
  - every line of code
  - everything that can conceivably go wrong!

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

NB: Testing can only reveal the *presence* of defects, not their absence!

Test Planning

The preparation of the test plan should begin when the system requirements are formulated, and the plan should be developed in detail as the software is designed.



The plan should be revised regularly, and tests should be repeated and extended wherever iteration occurs in the software process.

Testing Strategies

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.

Petschenik (1985) proposes:

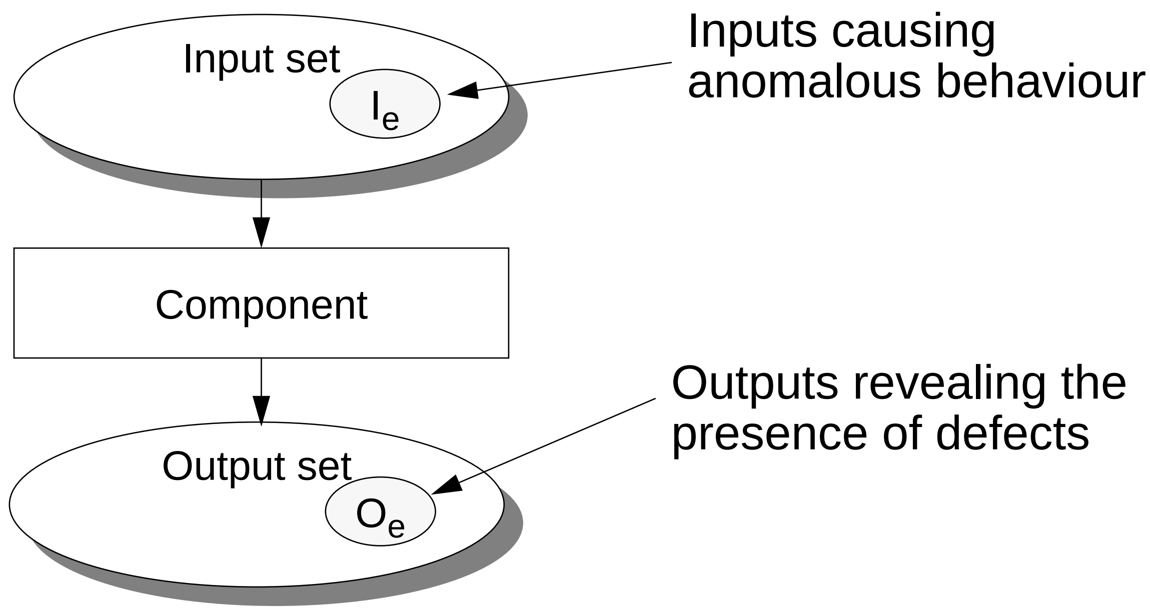
1. "Testing a system's capabilities is more important than testing its components."
   - ☞ Choose test cases that will identify situations that may prevent users from doing their job.

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 testing

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



Test cases are derived from the *external* specification of the component.

Coverage criteria:

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

Equivalence Partitioning

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.

Example:

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

Check input partitions:

- ❑ Do the inputs fulfil the pre-conditions?
- ❑ 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

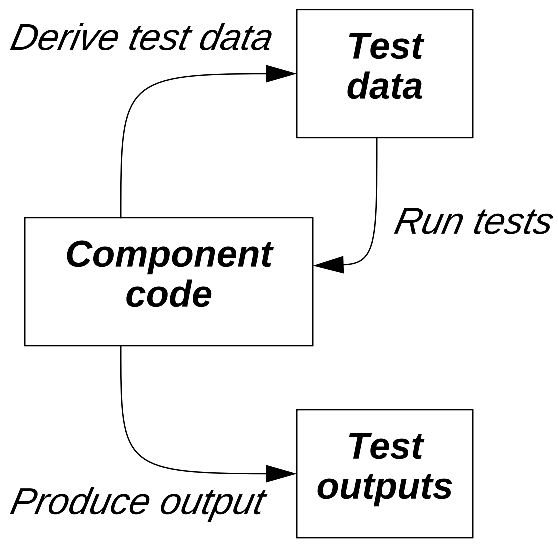
Generate test data that cover all meaningful equivalence partitions.

Test Cases	*Test Data*
Array length 0	key = 17, elements = { }
Array not sorted	key = 17, elements = { 33, 20, 17, 18 }
Array size 1, key in array	key = 17, elements = { 17 }
Array size 1, key not in array	key = 0, elements = { 17 }
Array size > 1, key is first element	key = 17, elements = { 17, 18, 20, 33 }
Array size > 1, key is last element	key = 33, elements = { 17, 18, 20, 33 }
Array size > 1, key is in middle	key = 20, elements = { 17, 18, 20, 33 }
Array size > 1, key not in array	key = 50, elements = { 17, 18, 20, 33 }
...	

Structural Testing

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 statement at least once
- all portions of control flow at least once
- all possible values of compound conditions at least once
- all portions of data flow at least once
- for all 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

Path testing is a white-box strategy which exercises every independent execution path through a component.

Binary Search Method

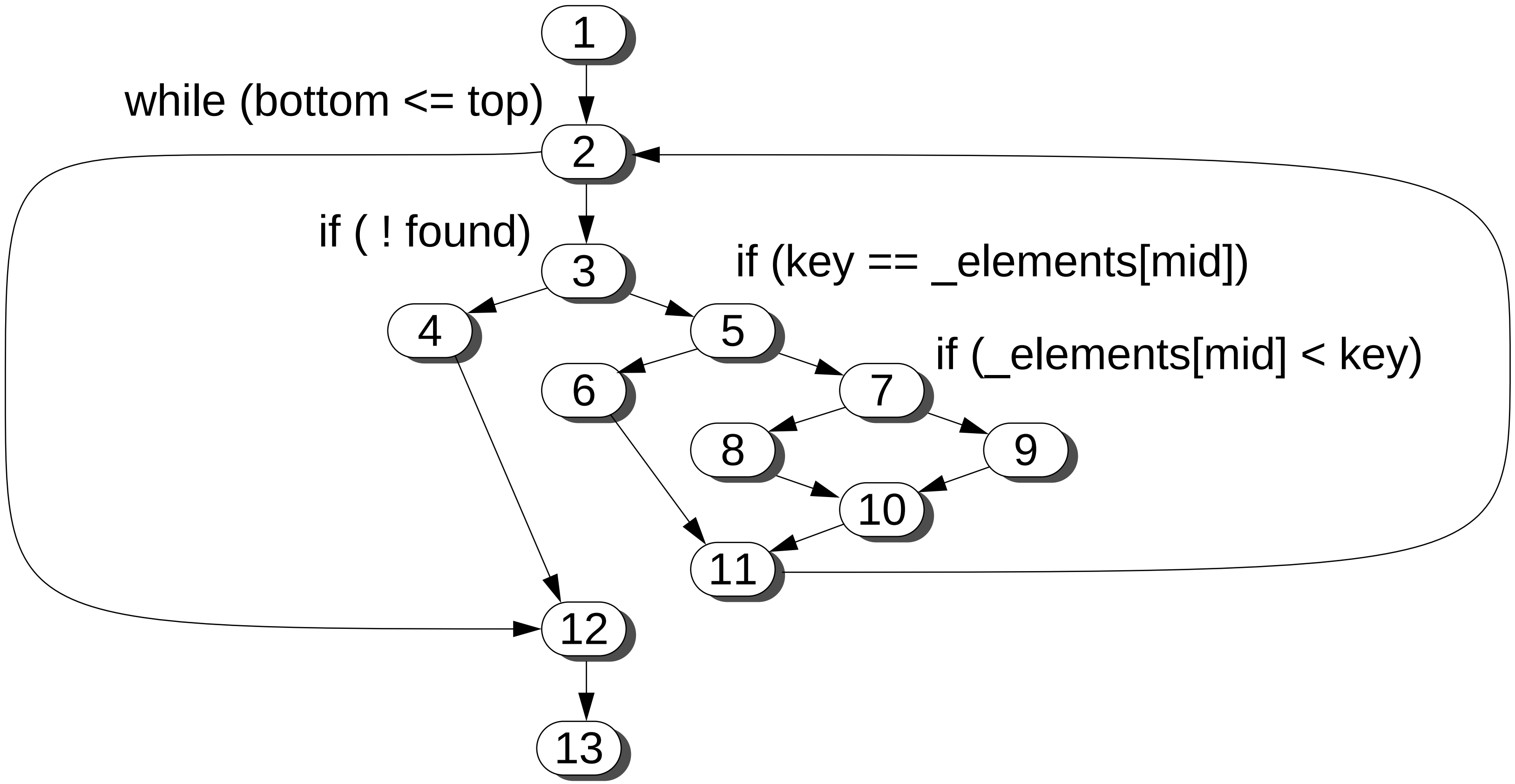
```
public boolean find(int key) throws assertionViolation {    // (1)
   assert(isSorted()); // pre-condition
   if (isEmpty()) { return false; } // Trivially can't find key in an empty list
   int bottom = 0;
   int top = _elements.length-1;
   int lastIndex = (bottom+top)/2;
   int mid;
   boolean 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) {                                           // (5)
         lastIndex = mid;                                    // (6)
      } else {
         if (_elements[mid] < key) {                         // (7)
            bottom = mid + 1;                                // (8)
         } else { top = mid - 1; }                           // (9)
      } // loop variant decreases: top - bottom
   }                                                         // (4)
   assert((key == _elements[lastIndex]) || !found); // post-condition
   return found;
}
```

Path Testing

A set of *independent paths* of a flow graph must cover all the edges in the graph:

e.g., {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}

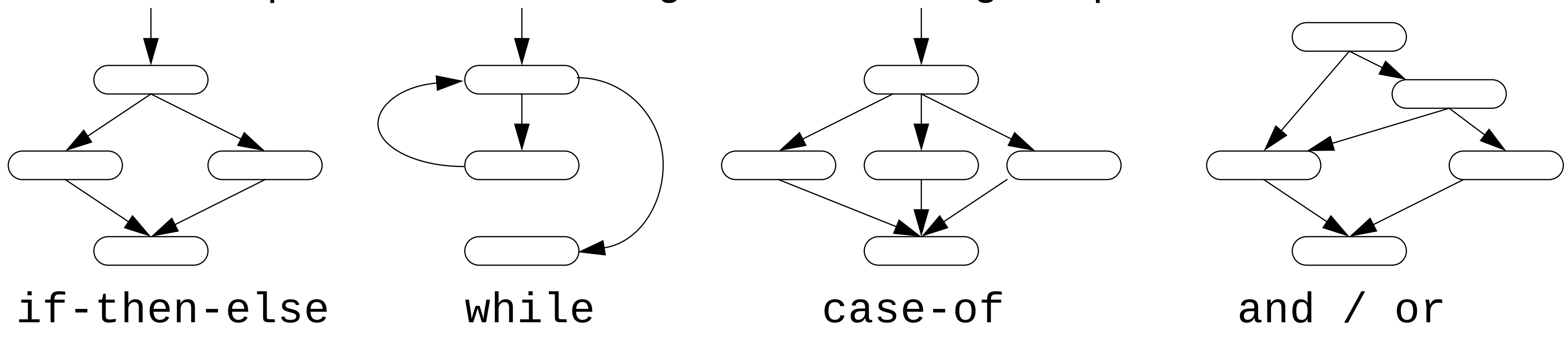


Test cases should be chosen to cover all independent paths through a routine.

Basis Path Testing: The Technique

See [Press92a]

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



2. Compute the Cyclomatic Complexity
   = #(edges) - #(nodes) + 2 = number of conditions + 1
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

Condition Testing

For complex boolean expressions, Basis Path Testing is not enough!

```
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;
}
```

Input values {x = 3, y=4} and {x = 4, y=3} will exercise all paths, but... {x = 3, y=3}

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

Statistical Testing

The objective of statistical testing is to determine the reliability of the software, rather than to discover software faults. Reliability may be expressed as:

- probability of failure on demand,
- rate of failure occurrence,
- mean time to failure,
- availability

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

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 (i) incremental development, (ii) formal specification, (iii) mathematical verification, and (iv) statistical testing

When to Stop?

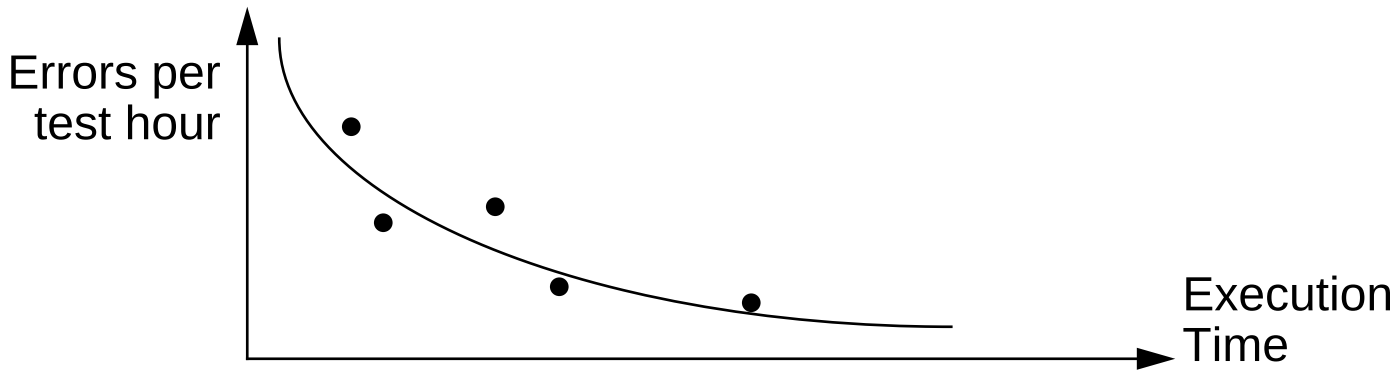
When are we done testing? When do we have enough tests?

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

Statistical Testing

- Test until you're reduced failure rate under risk threshold
  - Testing is like an insurance company calculating risks



Summary

You should know the answers to these questions:

- ❑ 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?*

11. Software Quality

Overview:

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

Sources:

- *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.
- *Software Engineering — A Practitioner's Approach*, R. Pressman, Mc-Graw Hill, Third Edn., 1994.
- *Fundamentals of Software Engineering,* C. Ghezzi, M. Jazayeri, D. Mandroli, Prentice-Hall 1991

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.

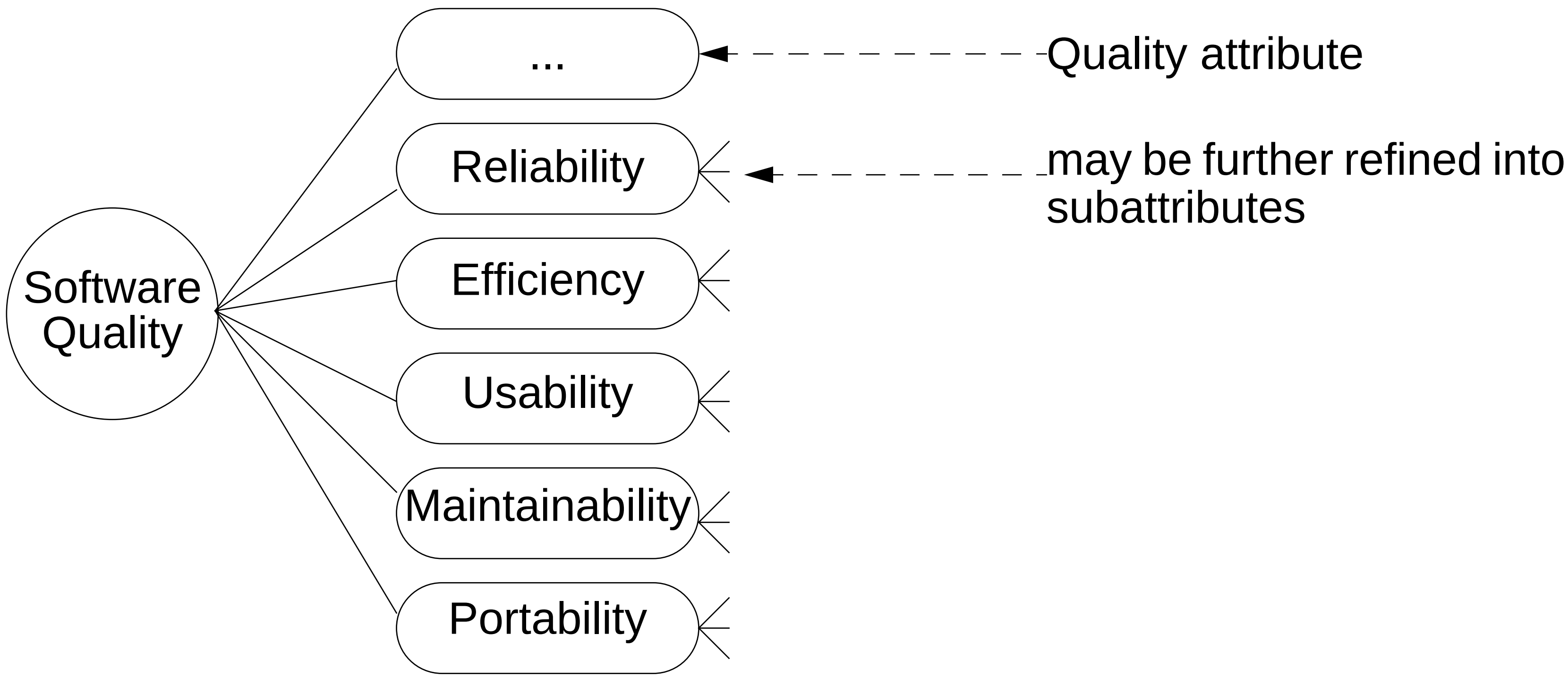
Problems:

- 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 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*)
- *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)

Correctness, Reliability, Robustness

3 external product attributes

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 undecideable

Reliability

- The user may rely on the system behaving properly
- 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)

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

2 external attributes, both process and product

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 man-power, time and money
  - ☞ relates to the “productivity” of a process

Usability (User Friendliness, Human Factors, Human Engineering)

- ❑ The degree to which the human users find the system (process) easy to use
  - ☞ 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 enthropy* => maintainability gradually decreases over time

Often refined in ...

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

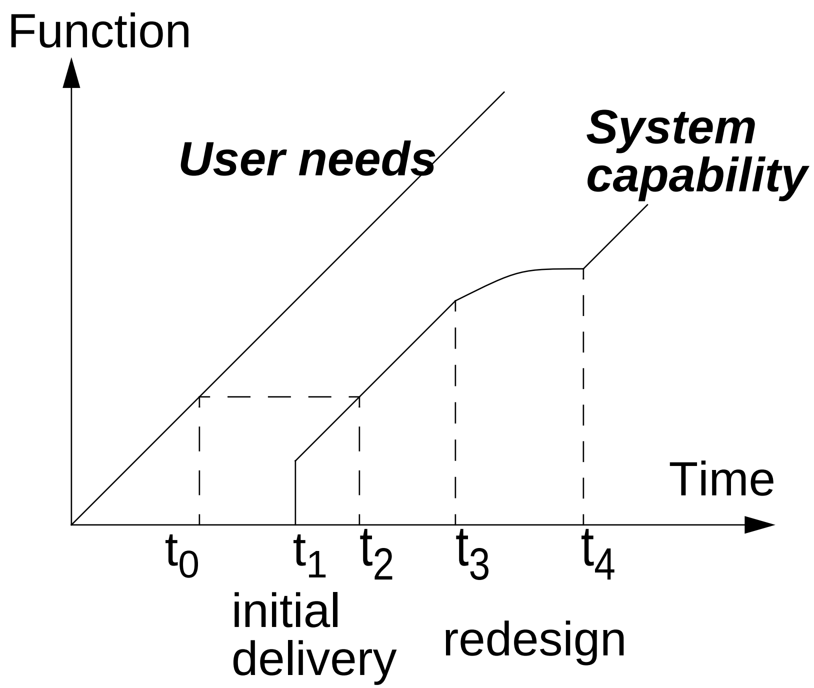
external process attribute (visibility also internal)

Productivity

- ❑ Amount of product produced by a process for a given number of resources
  - ☞ productivity among individuals varies a lot
  - ☞ often: productivity (∑ individuals) < ∑ productivity (individuals)

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

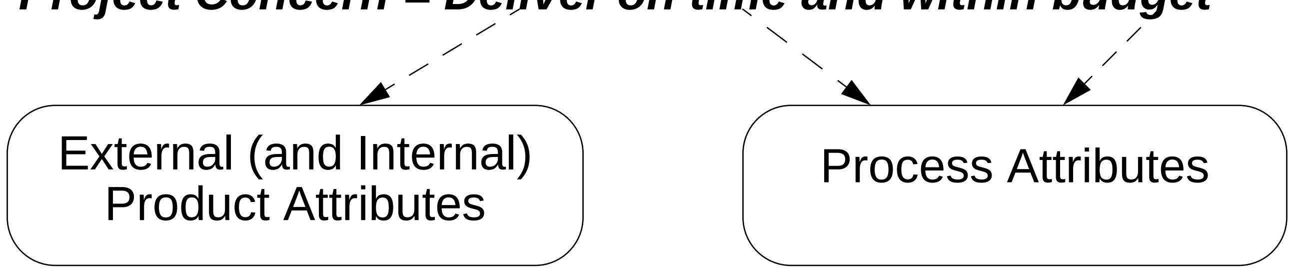


Visibility (Transparency, Glasnost)

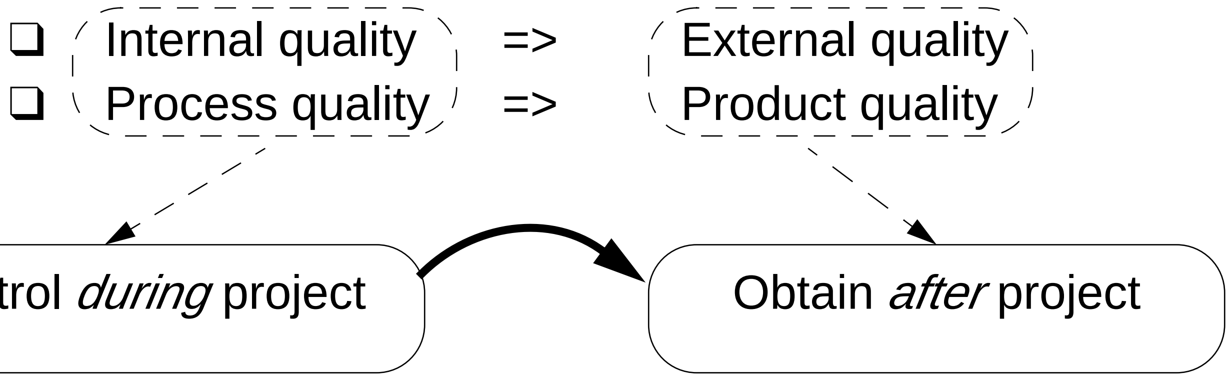
- ❑ Current process steps and project status is accessible
  - ☞ important for management; also deal with staff turn-over

Quality Control Assumption

Project Concern = Deliver on time and within budget



Assumptions:

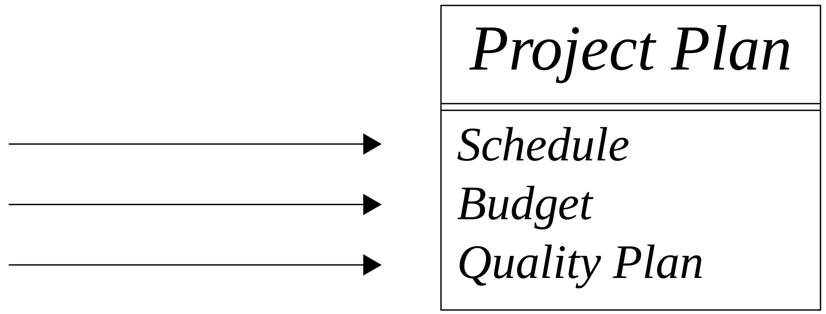


Otherwise, quality is mere coincidence!

The Quality Plan

Project Management:

- Plan Time
- Plan Money
- Plan Quality



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*.

Review type	*Principal purpose*
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.
Progress reviews	Driven by *budgets, plans and schedules* ❑ check whether project runs according to plan ❑ requires precise milestones ❑ both a process and a product review

- ❑ 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 and Minutes

Review meetings should:

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

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? ...

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 (II)

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? ...

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? ...

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.

Product standards	*Process standards*
Design review form	Design review conduct
Document naming standards	Submission of documents
Procedure header format	Version release process
Java programming style standard	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Problems

- 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.

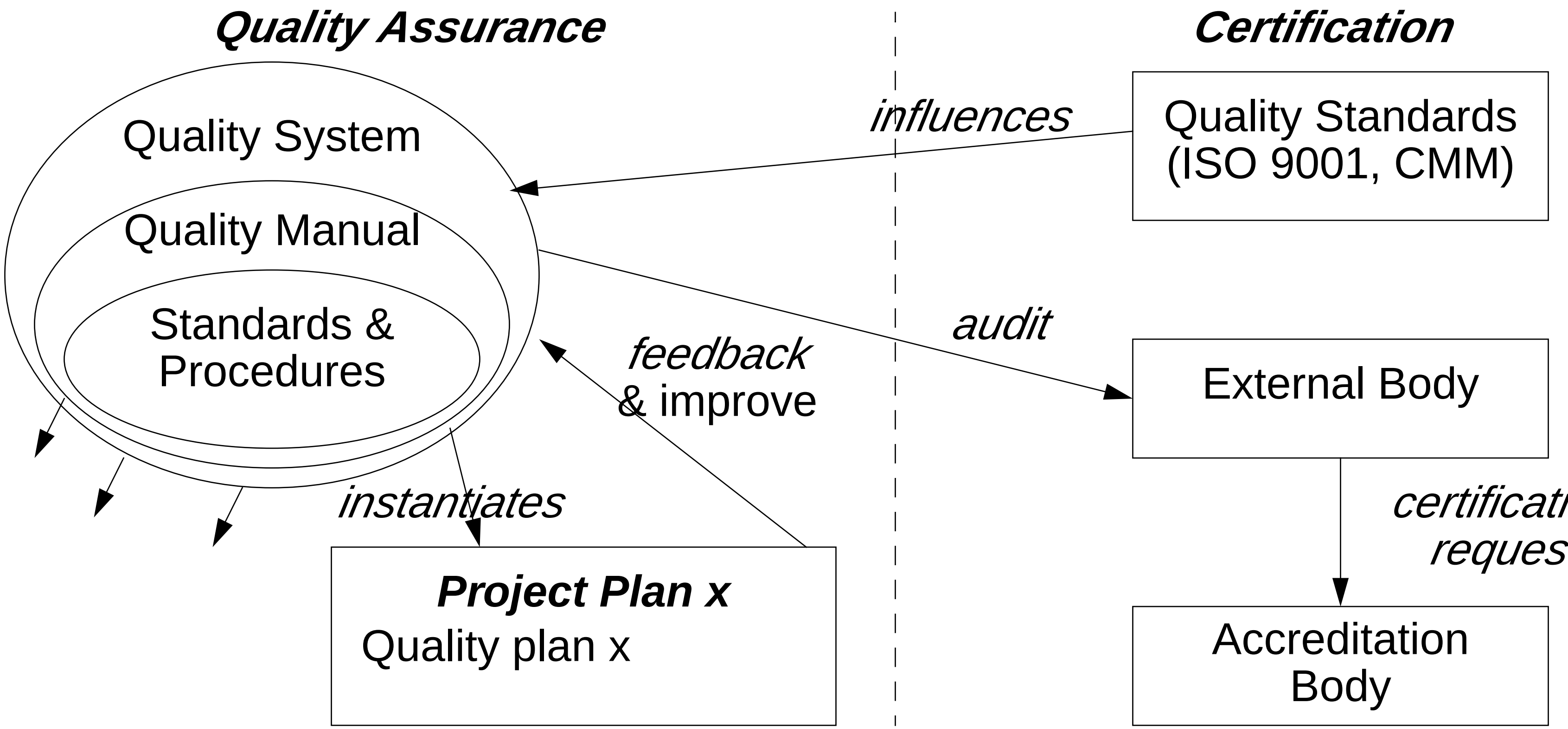
10.3 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

Quality System

When starting a project, the project will include a Quality Plan

- Ideally, such a plan should be an instance of the organization’s *Quality System*



Certain customers require an externally reviewed quality system

- An organization may request to *certify* its 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/

ISO 9001

Describes quality standards and procedures for developing products *of any kind*:

Management responsibility	Quality system
Control of non-conforming products	Design control
Handling, storage, packaging and delivery	Purchasing
Purchaser-supplied products	Product identification and traceability
Process control	Inspection and testing
Inspection and test equipment	Inspection and test status
Contract review	Corrective action
Document control	Quality records
Internal quality audits	Training
Servicing	Statistical techniques

Capability Maturity Model (CMM)

Assess how well contractors *manage software processes* (says little on projects).
(See [Somm96a] 31.4 The SEI Process maturity model)

Level 5: Optimizing
Improvement is fed back into QA process

Quantitative data is necessary for improvement!

Level 4: Managed
QA Process + quantitative data collection

Level 3: Defined
QA process is defined and institutionalized

Level 2: Repeatable
Formal QA procedures in place

Quality depends on individual project managers!

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

Quality depends on individuals!

Summary

You should know the answers to these questions:

- ❑ 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 by scheduled?*
- ✎ *Would you trust software developed by an ISO 9000 certified company? And if it were CMM?*

12. Software Metrics

Overview:

- ❑ Measurement Theory
- ❑ GQM Paradigm
- ❑ Quantitative Quality Model
- ❑ Sample Quality Metrics

Sources:

- ❑ Software Engineering, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.
- ❑ Software Engineering — A Practitioner's Approach, R. Pressman, Mc-Graw Hill, Third Edn., 1994.
- ❑ Norman E. Fenton, Shari l. Pfleeger, "Software Metrics: A rigorous & Practical Approach", Thompson Computer Press, 1996.

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

Measurement quantifies concepts

☞ understand, control and improve

Date	Measurement	Comment
2000 BC	Rankings “hotter than”	By touching objects, people could compare temperature
1600 AD	Thermometer “hotter than”	A separate device is able to compare temperature
1720 AD	Fahrenheit scale	Quantification allows to log temperature, study trends, predict phenomena (weather forecasting), ...
1742 AD	Celsius scale	
1854 AD	Kelvin scale	Absolute zero allows for more precise descriptions of physical phenomena

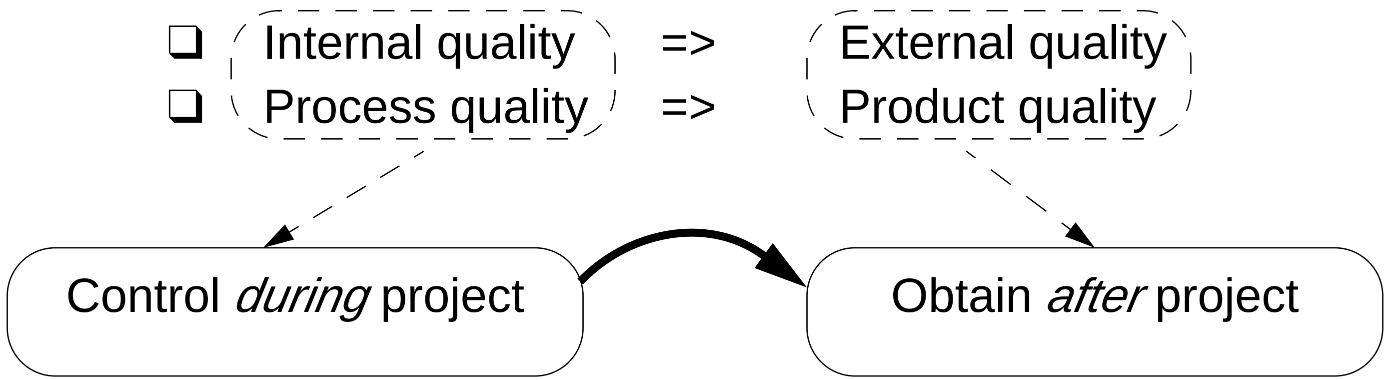
Why Software Metrics

Effort (and Cost) Estimation

- Measure early in the life-cycle to deduce later production efforts

Quality Assessment and Improvement

- Control software quality attributes during development
- Compare (and improve) software production processes
- Remember *Quality Assumptions*



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 *(# words / # sentences) + (percentage of words >= 3 syllables)
  - number of person-days required to implement a use-case
- According to measurement theory, Metric is an incorrect name for Measure
  - a Metric m is a function measuring distance between two objects
    such that m(x,x) = 0; m(x,y) = m(y,x); m(x,z) <= m(x,y) + m(y,z)

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

Possible Problems

Example: Compare productivity of programmers in lines of code per time unit.

- ❑ Do we use the same units to compare? [***"Preciseness" of Measurement***]
  - ☞ What is a "line of code"? What is the "time unit"?
- ❑ Is the context the same?
  - ☞ Were programmers familiar with the language? [***"Preciseness"***]
- ❑ Is "code size" really what we want to produce? [***Representation Condition***]
  - ☞ What about code quality?
- ❑ How do we want to interpret results? [***Scale and Scale Types***]
  - ☞ Average productivity of a programmer?
    Programmer X is more productive than Y?
    Programmer X is twice as productive as Y?
- ❑ What do we want to do with the results? [***GQM-paradigm***]
  - ☞ Do you reward "productive" programmers?
    Do you compare productivity of software processes?

Metrics theory will help us to answer these questions...

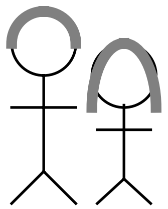
Empirical Relations

Observe true/false relationships between (attributes of) real world entities

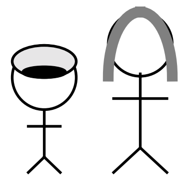
Empirical relations are *complete*, i.e. defined for all possible combinations

Examples: empirical relationships between height attributes of persons

"is taller than" binary relationship

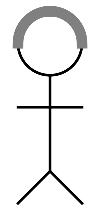


Frank "is taller than" Laura

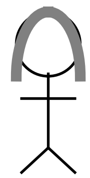


Joe "is not taller than" Laura

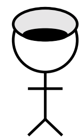
"is tall" unary relationship



Frank "is tall"

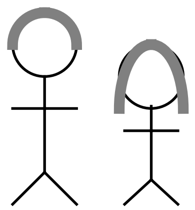


Laura "is tall"

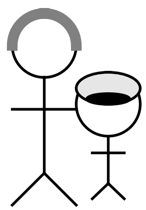


Joe "is not tall"

"is much taller than" binary relationship

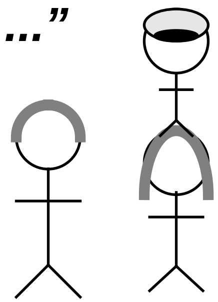


Frank "is not much taller than" Laura



Frank "is much taller than" Joe

"... is higher than ... + ..." ternary relationship



Frank "is not higher than" Joe on Laura's shoulders

Measurement Mapping

Measure & Measurement

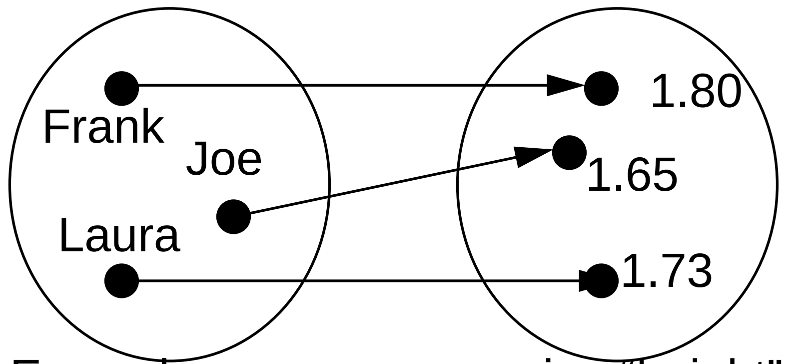
A *measure* is a function mapping

- 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.



Example: measure mapping “height” attribute of person on a number representing “height in meters”.

Purpose

Manipulate symbol(s) in the range => draw conclusions about attribute(s) in the domain

Preciseness

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

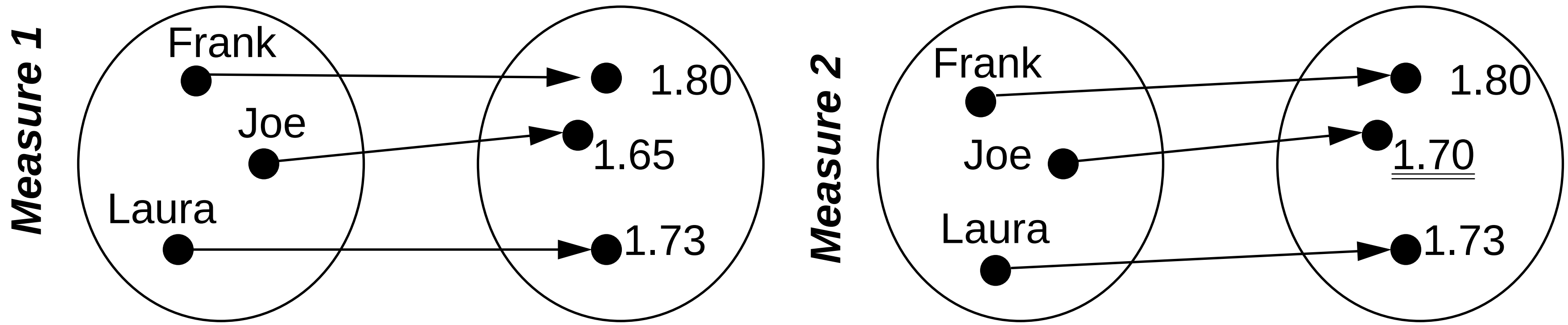
- *domain*: do we measure people’s height or width?
- *range*: do we measure height in centimetres or inches?
- *mapping rules*: do we allow shoes to be worn?

Representation Condition

To be *valid*, a measure must satisfy the ***representation condition***

- empirical relations (in domain) <=> mathematical relations (in range)

In general, the more empirical relations, the more difficult it is to find a valid measure.



Empirical Relation		***Measure 1***		***Measure 2***	
is-taller-than		***x > y***		***x > y***	
Frank, Laura	true	1.80 > 1.73	true	1.80 > 1.73	true
Joe, Laura	false	1.65 > 1.73	false	1.70 > 1.73	false
is-much-taller-than		***x > y + .10***		***x > y + .10***	
Frank, Laura	false	1.80 > 1.73 + .10	false	1.80 > 1.73 + .10	false
Frank, Joe	true	1.80 > 1.65 + .10	true	*1.80 > 1.70 + .10*	*false*

Scale

Scale

= the symbols in the range of a measure + the permitted manipulations

- ☞ When choosing among valid measures, we prefer a richer scale (i.e., one where we can apply more manipulations)
- ☞ Classify scales according to permitted manipulations => Scale Type

Typical Manipulations on Scales

- ❑ *Mapping*: Transform each symbol in one set into a symbol in another set
  - ☞ {false, true} -> {0, 1}
- ❑ *Arithmetic*: Add, Substract, Multiply, Divide
  - ☞ It will take us twice as long to implement use-case X than use-case Y
- ❑ *Statistics*: Averages, Standard Deviation, ...
  - ☞ The average yearly temperature in Helsinki is 4.4^{o}C

Scale Types

Name	***Characteristics*** ***Permitted Manipulations***	***Example*** ***Forbidden Manipulations***
Nominal	- n different symbols - no ordering	{true, false} {design error, implementation error}
	- all one-to-one transformations	- no magnitude, no ordering - no median, percentile
Ordinal	- n different symbols - ordering is implied	{trivial, simple, moderate, complex} {superior, equal, inferior}
	- order preserving transformations - median, percentile	- no arithmetic - no average, no deviation
Interval	Difference between any pair is preserved by measure	Degrees in Celsius or Fahrenheit
	- Addition (+), Substraction (-) - Averages, Standard Deviation - Mapping have the form M = aM' + b	- Multiplication (*), Division (/) not ("20^{o}C is twice as hot as 10^{o}C" is forbidden as expression)
Ratio	Difference and ratios between any pair is preserved by measure. There is an absolute zero.	Degrees in Kelvin Length, size, ...
	- All arithmetic - Mappings have the form M = aM'	nihil

GQM

Goal - Question - Metrics approach

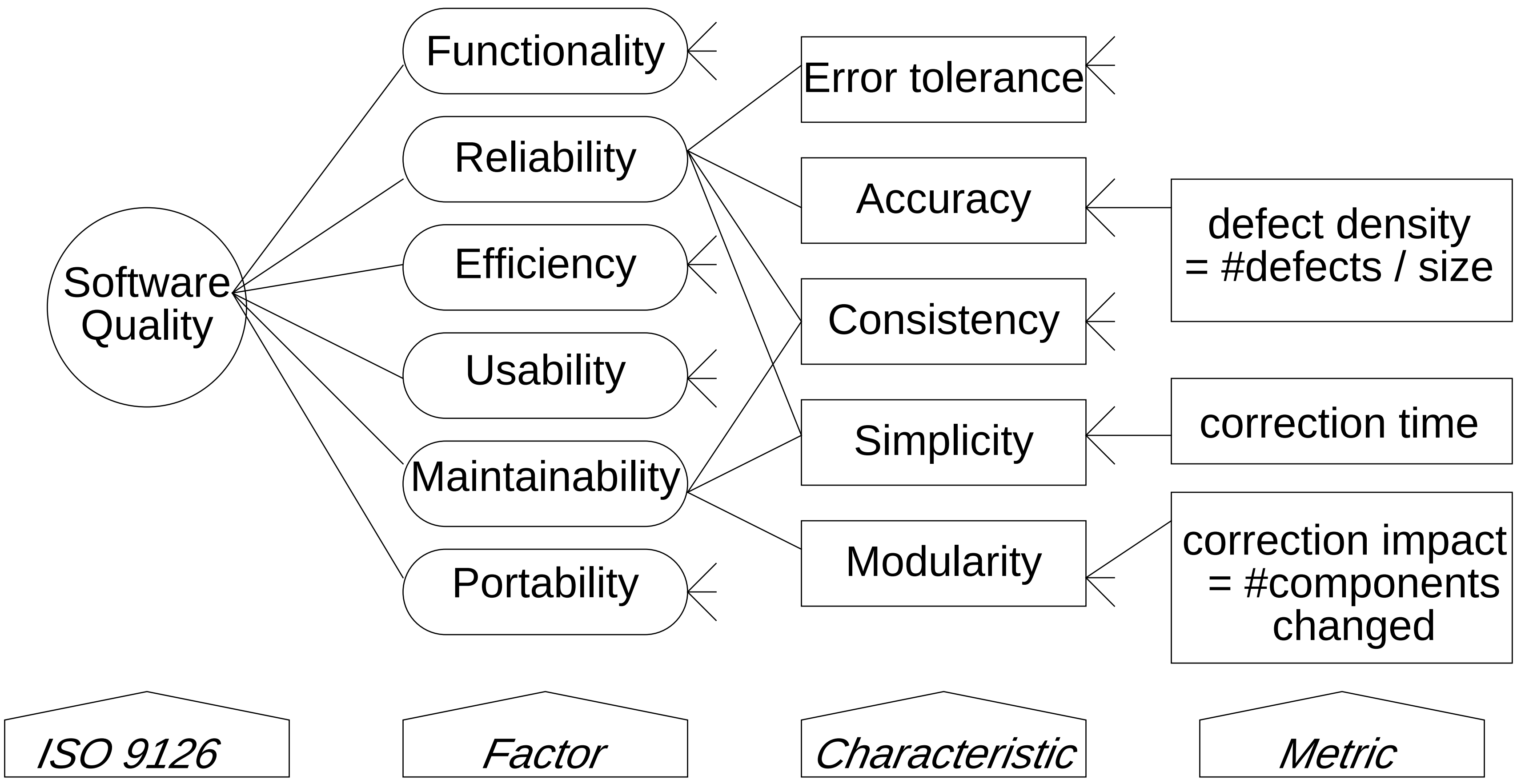
([Somm96a], [Press94a] all citing [Basili et al. 1984])

- 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 ...

Quantitative Quality Model

Quality according to ISO 9126 standard

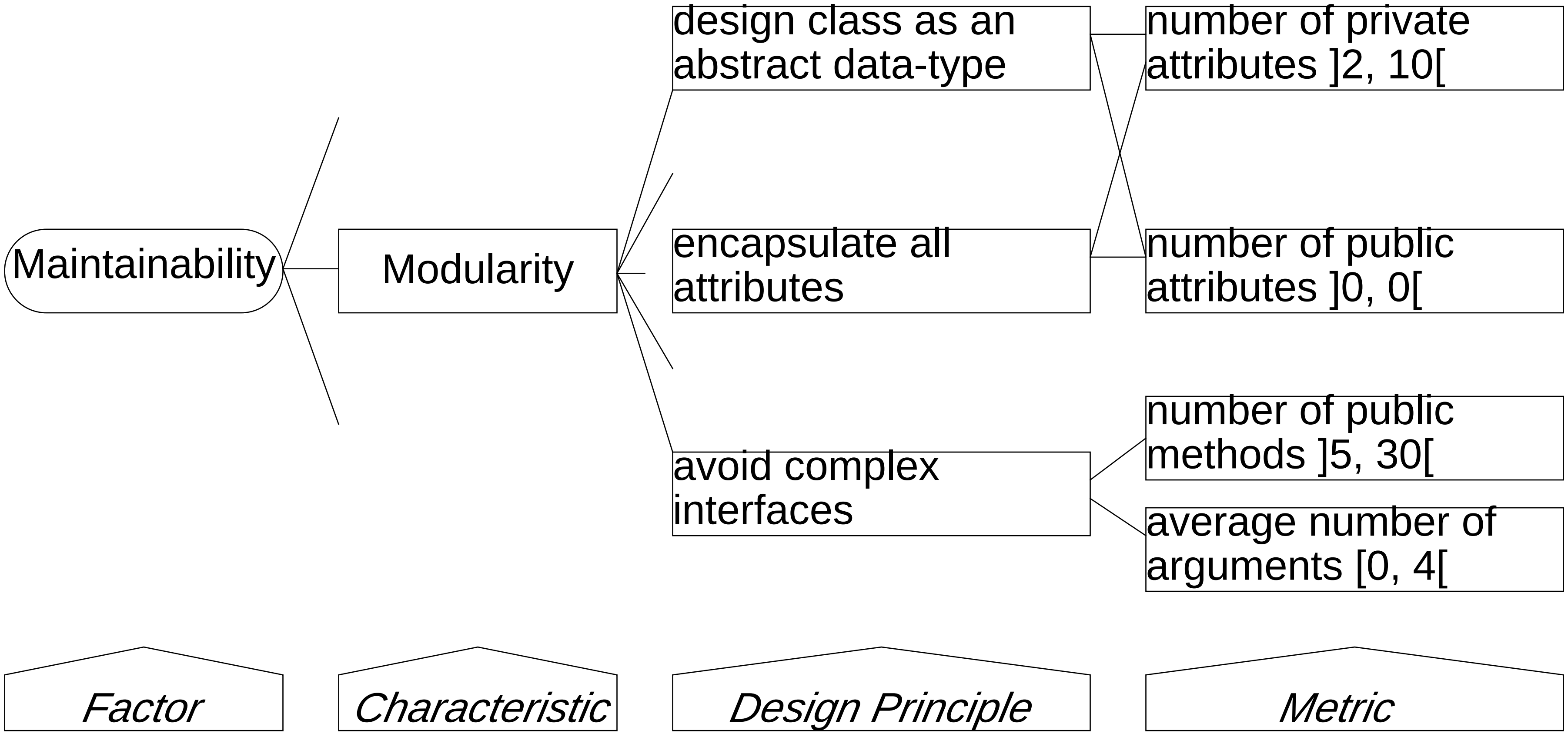
- Divide-and conquer approach via “hierarchical quality model”
- Leaves are simple metrics, measuring basic attributes



“Define your own” Quality Model

Define the quality model with the development team

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



Sample Size (and Inheritance) Metrics

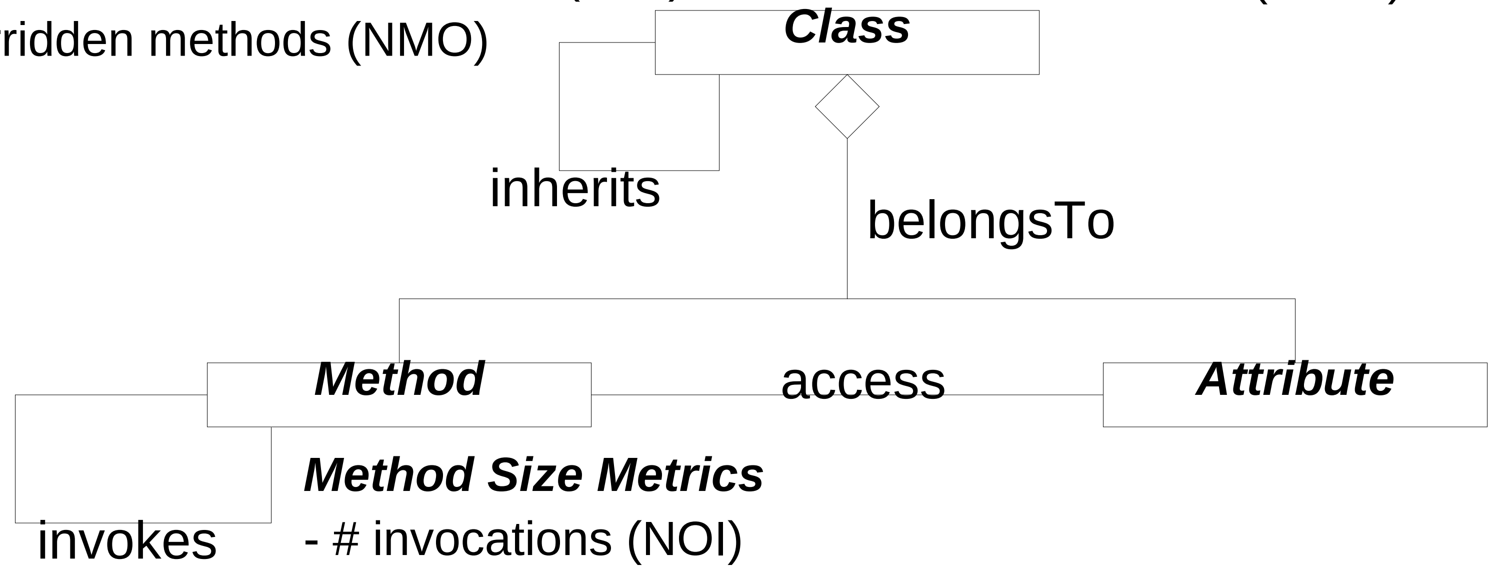
These are Internal Product Metrics

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)
- # Σ of method size (WMC)



Method Size Metrics

- # invocations (NOI)
- # statements (NOS)
- # lines of code (LOC)
- # arguments (NOA)

Sample Coupling & Cohesion Metrics

These are Internal Product 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

Beware

Researchers disagree whether coupling/cohesion methods are *valid*

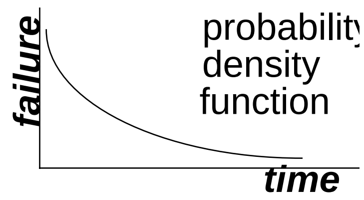
- ❑ Classes that are observed to be cohesive may have a high LCOM value
  - ☞ due to accessor methods
- ❑ Classes that are not much coupled may have high CBO value
  - ☞ no distinction between data, method or inheritance coupling

Sample External 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!

Reliability (Product Metric)



- mean time to failure = mean of probability density function PDF
  - MTTF (T) = $\int t\, f(t)\, dt$
  - for hardware, PDF is usually a negative exponential
    $$f(t) = \lambda e^{-\lambda t}$$
  - for software one must take into account the fact that repairs will influence the rest of the function => 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 External Quality Metrics (II)

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 you're data collection is sound!

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 for the units
  - categories of changes is subjective
  - time =?

    problem recognition time + administrative delay time +
    problem analysis time + change time + testing & reviewing time

Conclusion: Metrics for QA (I)

Question:

- ❑ Can internal product metrics reveal which components have good/poor quality?

Yes, but...

- ❑ Not reliable
  - – false positives: “bad” measurements, yet good quality
  - – false negatives: “good” measurements, yet poor quality
- ❑ Heavy Weight Approach
  - – Requires team to develop (customize?) a quantitative quality model
  - – Requires definition of thresholds (trial and error)
- ❑ Difficult to interpret
  - – Requires complex combinations of simple metrics

However...

- ❑ Cheap once you have the quality model and the thresholds
- ❑ Good focus (± 20% of components are selected for further inspection)
  Note: focus on the most complex components first

Conclusion: Metrics for QA (II)

Question:

- ❑ Can external product/process metrics reveal quality?

Yes, ...

- ❑ More reliably then internal product metrics

However...

- ❑ Requires a finished product or process
- ❑ It is hard to achieve preciseness
  - ☞ even if measured in same units
  - ☞ beware to compare results from one project to another

Summary

You should know the answers to these questions

- What are the possible problems of metrics usage in software engineering? How does the metrics theory address them?
- What kind of measurement scale would you need to say “A specification error is worse than a design error”? And what if we want to say “A specification error is twice as bad as a design error?”
- 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?
- Why is it so important to have “good” product size metrics?
- Why do we prefer measuring Internal Product Attributes instead of External Product Attributes during Quality Control? What is the main disadvantage of doing that?
- Why are coupling/cohesion metrics important? Why then are they so rarely used?

13. Outlook: Heavy vs. Light Methods

Soon in this theatre!