7029 Informatik 2A

Prof. O. Nierstrasz

Sommersemester 1998

Table of Contents

1. Informatik 2A — Software Engineering

Lecturer: Prof. Oscar Nierstrasz
Schützenmattstr. 14/103, Tel. 631.4618

Secretary: Frau I. Huber, Tel. 631.4692

Assistants: Jean-Guy Schneider, Tobias Röthlisberger, Lukas Steiger

WWW: http://www.iam.unibe.ch/~scg

Principle Texts:

- *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.
- *Unified Modeling Language — Notation Guide*, version 1.1, Rational Software Corporation, 1997.
- *Designing Object-Oriented Software*, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990.
- *Object-Oriented Software Construction*, Second ed., B. Meyer, Prentice Hall, 1997.
- *The Mythical Man-Month*, F. Brooks, Addison-Wesley, 1975.

Course Overview

1.	27.03	Introduction: Modularity and Software Engineering
2.	03.04	The Software Lifecycle
	10.04	*Good Friday — no lecture*
3.	17.04	Modelling Objects and Classes
4.	24.04	Modelling Behaviour
5.	01.05	Responsibility-Driven Design
6.	08.04	Detailed Design
7.	15.05	Software Validation
8.	22.05	Design by Contract
9.	29.05	Design Patterns
10.	05.06	Project Management
11.	12.06	Software Tools
12.	19.06	4GLs: Delphi — *guest lecture*
	26.06	*Final exam*

Introduction

- ❑ What is Software Engineering?
- ❑ Problems with the Classical Software Lifecycle
  - ☞ chronically inaccurate cost estimates
  - ☞ low productivity
  - ☞ inflexible software products
- ❑ Modularity as the key to good Software Engineering practice

What is Software Engineering?

A naive view:

Problem Specification $\xrightarrow{\textit{coding}}$ 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?

Software Engineering is much more than just programming!

> *The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*
>
> *— F. Bauer, 1969*

Some Software Myths

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

Reality: Poor up-front definition is the major cause of project failure.

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

Reality: Adding more people typically slows a project down.

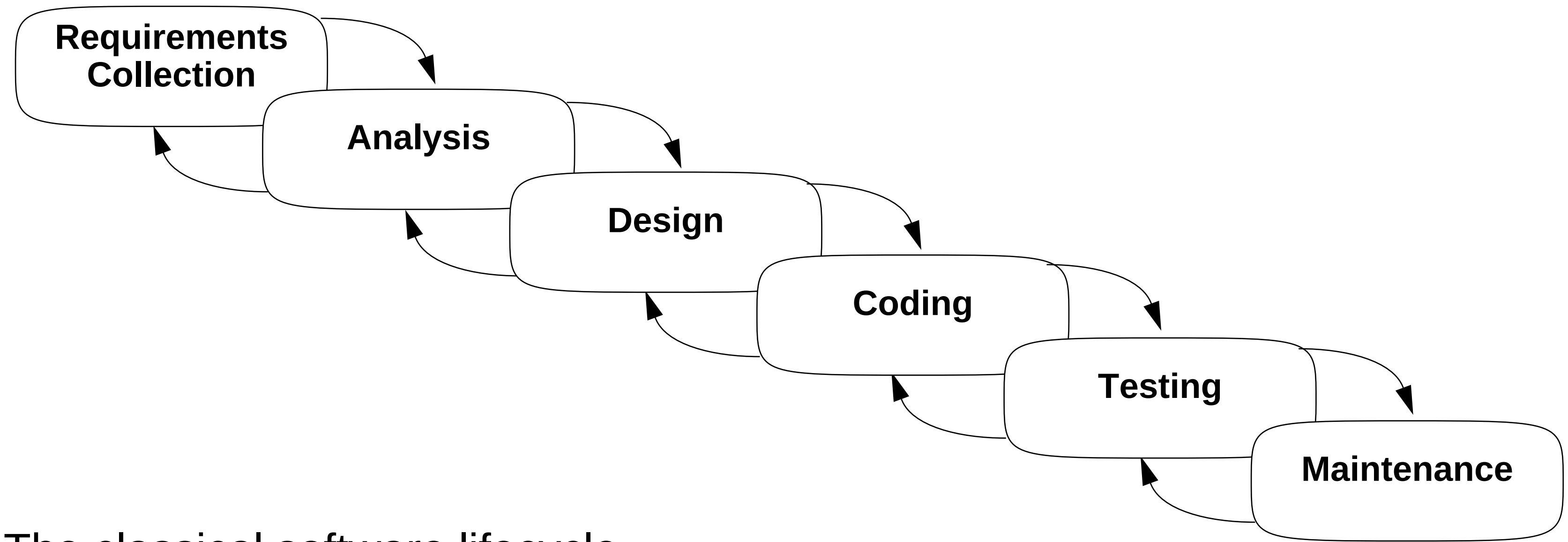
“**Myth:** *“The only deliverable for a successful project is the working program.”*

Reality: Documentation of all aspects of software development are needed to ensure maintainability.

Why software isn’t like hardware:

- “Software is developed or engineered, not manufactured in the classical sense
- Software doesn’t ‘wear out’
- Most software is custom-built rather than being assembled from components”

The Classical Software Lifecycle



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

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

The Software Crisis: Symptoms

The “software crisis” refers to the chronic inability of the software industry to develop reliable, flexible software systems that meet the constantly changing requirements of its ever expanding customer base ...

1. “Hardware sophistication has outpaced our ability to build software to tap hardware’s potential.
2. Our ability to build new programs cannot keep pace with the demand for new programs.
3. Our ability to maintain existing programs is threatened by poor design and inadequate resources.”

— *Pressman, SE, pp. 6-7*

The Software Crisis: Causes

Problems:

1. "Schedule and cost estimates are often grossly inaccurate."
2. "The 'productivity' of software people hasn't kept pace with the demand for their services."
3. "The quality of software is sometimes less than adequate."

Causes:

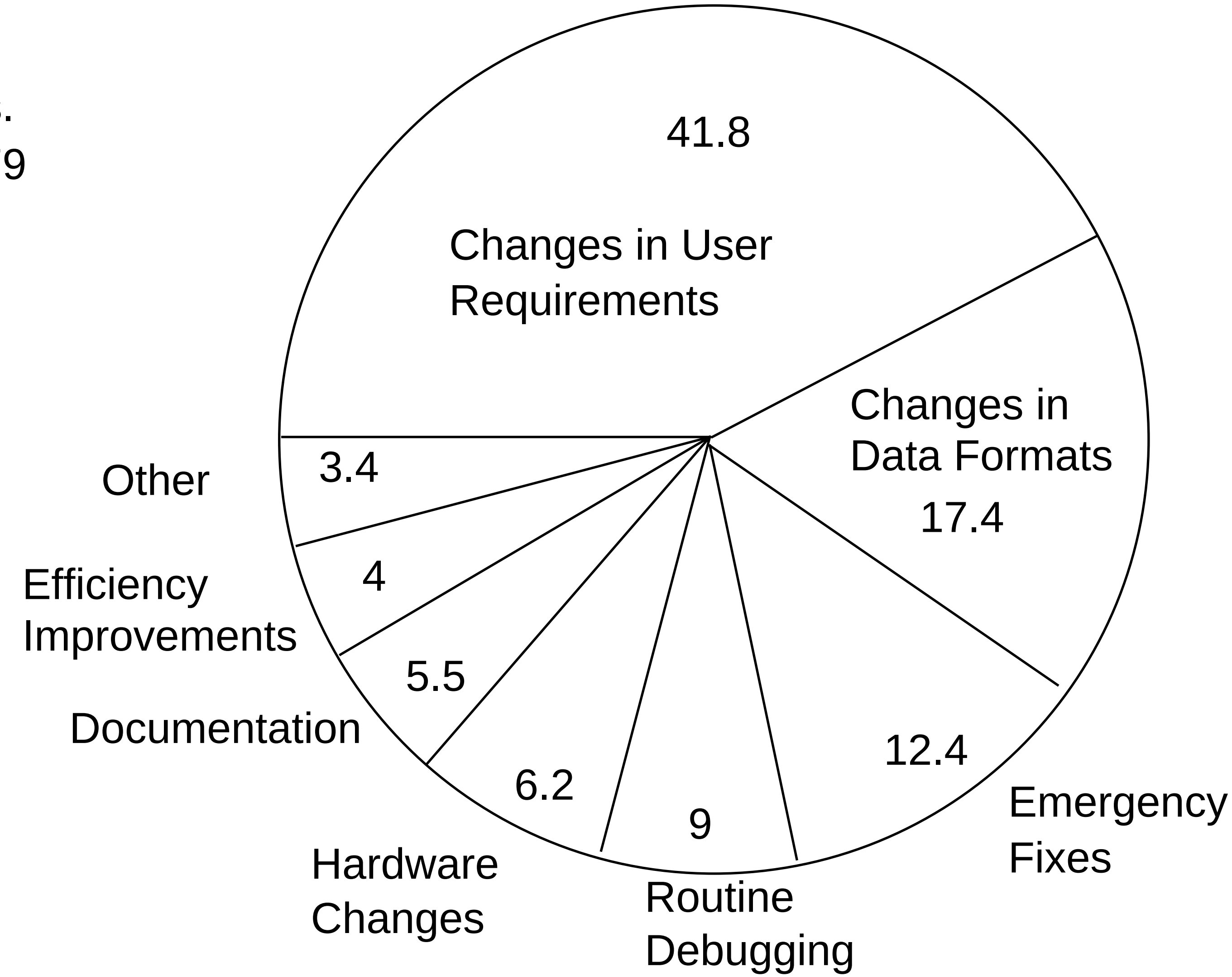
- ❑ Few reliable data on software process; poor predictability; weak basis to evaluate new tools, methods etc.
- ❑ Frequent customer dissatisfaction; inadequate formulation/understanding of requirements; poor communication between customer and developer
- ❑ Software quality is often suspect; quantitative measures of reliability and quality assurance are only now emerging
- ❑ Existing software can be hard to maintain ("legacy systems"); maintenance is typically more expensive than initial development

— Pressman, SE, pp. 17-18

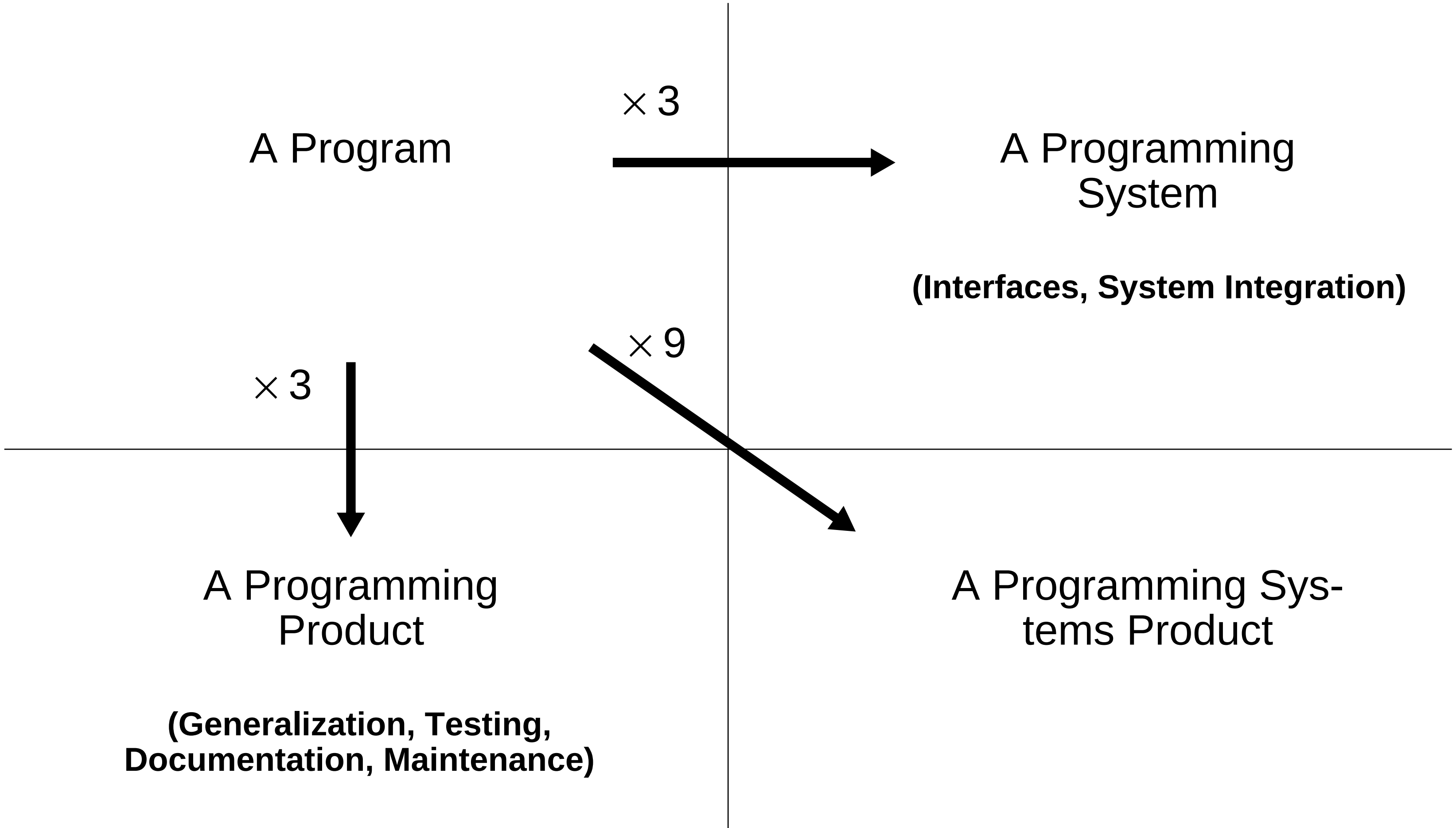
Maintenance

Breakdown of maintenance costs.

Source: Lientz 1979



Programs vs. Products



— Brooks, MMM, p. 5

Software Quality

Correctness	is the ability of software products to perform their exact tasks, as defined by their specifications
Robustness	is the ability of software systems to react appropriately to abnormal conditions
Extendibility	is the ease of adapting software products to changes of specification
Reusability	is the ability of software elements to serve for the construction of many different applications
Compatibility	is the ease of combining software elements with others
Efficiency	is the ability of a software system to place as few demands as possible on hardware resources ...
Portability	is the ease of transferring software products to various hardware and software environments
Ease of use	is the ease with which people of various backgrounds and qualifications can learn to use software products ...

— Meyer, OOSC, ch. 1

Criteria for Modularity

A design method supports modularity if:

Decomposability	it helps in the task of decomposing a software problem into a small number of less complex subproblems
Composability	it favours the production of software elements which may then be freely combined to produce new systems
Understandability	it helps produce software in which a human reader can understand each module without having to know the others
Continuity	a small change in the problem specification will trigger a change in just one module (or a small number of modules)
Protection	it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module (or at worst to a few neighbouring modules)

— *Meyer, OOSC, pp. 40-46*

Rules to ensure Modularity

We can ensure modularity if:

Direct Mapping	the modular structure of the software system should be compatible with the modular structure devised in modeling the problem domain
Few interfaces	every module should communicate with as few others as possible
Small interfaces (weak coupling)	if two modules communicate, they should exchange as little information as possible
Explicit interfaces	whenever two modules A and B communicate, this must be obvious from the text of A or B or both
Information hiding	the designer of every module must select a subset of the module's properties to be made available as the public interface to authors of client modules

— *Meyer, OOSC, pp. 46-53*

Principles of Modularity

The following principles guide modular software development:

Linguistic Modular Units	modules must correspond to syntactic units in the language used
Self Documentation	the designer of a module should strive to make all information about the module part of the module itself
Uniform Access	all services offered by a module should be available through a uniform notation (which does not betray whether they are implemented through storage or computation)
Open-Closed	modules should be both open and closed
Single Choice	whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list

— Meyer, OOSC, pp. 53-63

The Open-Closed Principle

- “A module is *open* if it is still available for extension.”

- “A module is *closed* if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (i.e., interface).

— Meyer, OOSC, pp. 57-58

The Role of Modularity

"Software maintenance, which consumes a large proportion of software costs, is penalized by the difficulty of implementing changes in software products, and by over-dependence of programs on the physical structure of the data they manipulate."

"Modularity is the key to achieving the aims of reusability and extendibility."

"Effective project management requires support for languages that are both open and closed. But classical approaches to design and programming do not permit this."

— Meyer, OOSC, 1st edn.

Component-Oriented Development

Generic	*Specific*
Domain models	Requirements specification
Prototyping tools	Analysis Model
Design patterns, generic architectures	Design
Frameworks, 4GLs	Coding
Automated testing tools ...	Testing
Generic architectures ...	Maintenance

Summary

You should know the answers to these questions:

- ❑ How does Software Engineering differ from programming?
- ❑ What are the phases of the classical software lifecycle?
- ❑ Why is the “waterfall” model unrealistic?
- ❑ Why is maintenance the most expensive phase of the software lifecycle?
- ❑ How does modularity enhance maintainability?

Can you answer the following questions?

- ✎ *How does Software Engineering differ from Engineering?*
- ✎ *What is the difference between Analysis and Design?*
- ✎ *How should requirements be specified?*
- ✎ *How does object-oriented programming support the goals of software engineering?*

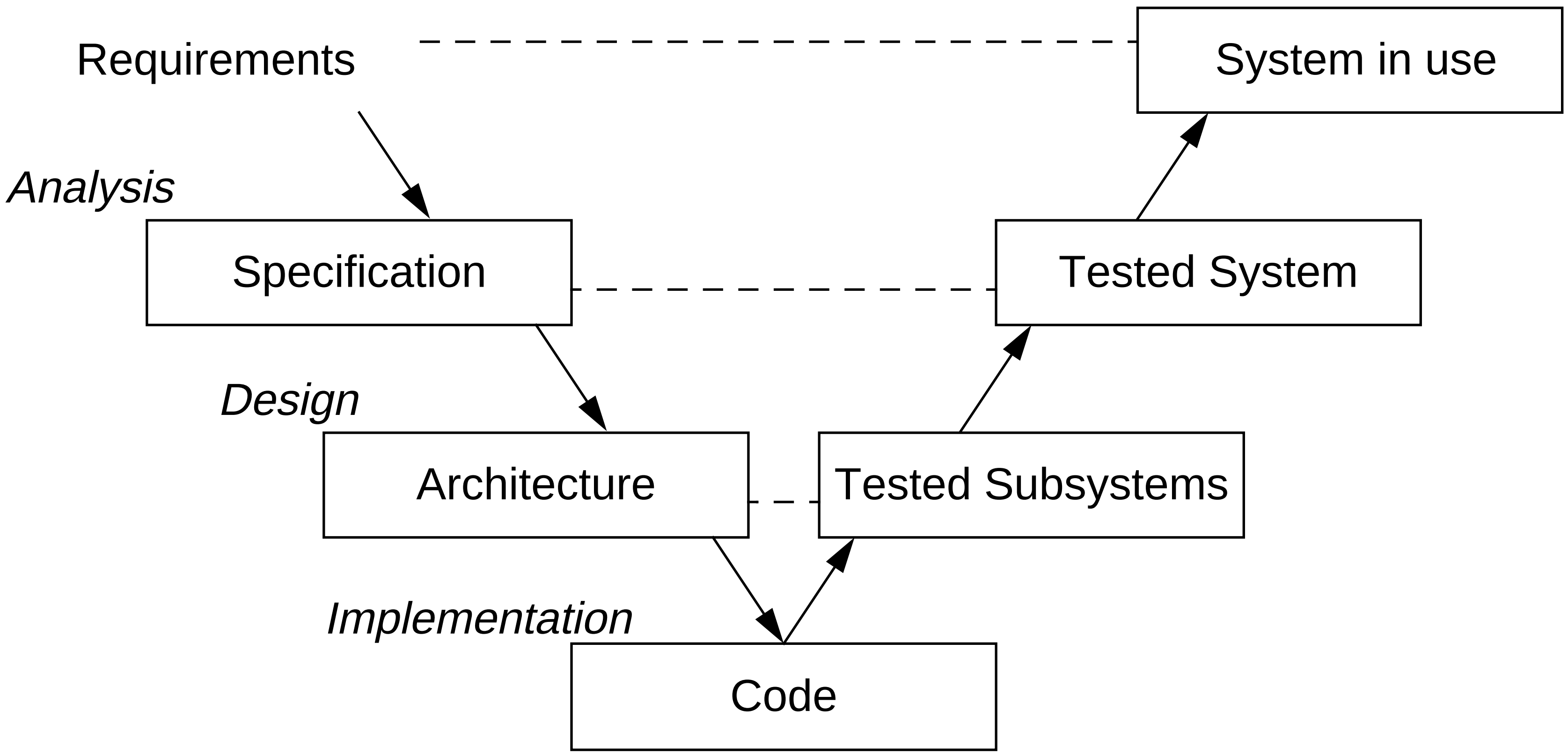
2. The Software Lifecycle

- Phases of Software Development
- Analysis vs. Design
- Iterative and incremental development
- Software architecture driven by functions or data?
- Object-oriented design

Sources:

- *Object-Oriented Development — The Fusion Method*, D. Coleman, et al., Prentice Hall, 1994.
- *Object-Oriented Software Construction*, 2d edn., B. Meyer, Prentice Hall, 1997.

Phases of Software Development



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

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 [cf. Fusion] results in models of the system which describe:

- *classes* of objects that exist in the system
- *relationships* between those classes
- *operations* that can be performed on the system
- allowable *sequences* of those operations

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

Design

Design is the process of specifying *how* the specified system behaviour will be realized from software components. The result is an *architecture document.*

Object-oriented design [cf. Fusion] 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

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.

Not Programming

Many critical aspects of software engineering in “real” projects are not directly related to programming:

- ☞ project planning: deliverables, manpower allocation, dependencies ...
- ☞ cost estimation, monitoring
- ☞ documentation
- ☞ configuration management
- ☞ validation and testing
- ☞ ...

Why use a Method?

Requirements checking:

- ❑ System modelling 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

Functions, Data and Continuity

Should we structure software architecture around functions or data?

Recall the criterion of *continuity:*

The *quality* of an architecture should not be measured only in terms of initial requirements, but in terms of how *robust* it is in the face of changing requirements.

- ☞ As a system evolves, the functions it performs tend to be the most volatile part. *Successful systems will be asked to perform new functions!* An architecture based extensively on initial functionality will not evolve as smoothly as the requirements.
- ☞ Even in the face of changing requirements and functionality, a system will tend to deal with the same kinds of data.
(Payroll programs manipulate employee records, tax information, etc.; window systems deal with windows, menus, icons, etc.)

The Top-Down Functional Approach

Traditional top-down design is based on stepwise refinement:

- ❑ Translate a C program to Motorola 68030 code
  - Read the program and produce a sequence of tokens
  - Parse the sequence of tokens into an abstract syntax tree
  - Decorate the tree with semantic information
  - Generate code from the decorated tree

Why it fails for long-term evolution:

- ☞ requirements are assumed to be complete and unchanging
- ☞ viewing a system as a single function is rarely appropriate
- ☞ data structure is easily neglected
- ☞ top-down development does not promote reusability

“Real systems have no top.”

Why Use a Bottom-Up Data-driven Design?

Compatibility:

- ❑ Subsystems can be easily combined only if they agree on the common data structures.

Reusability:

- ❑ Component reuse is inherently bottom-up: opportunities for reuse can be recognized by understanding how data are used.

Continuity:

- ❑ Over time, data structures — viewed abstractly — are the most stable parts of a system.

What is Object-Oriented Design?

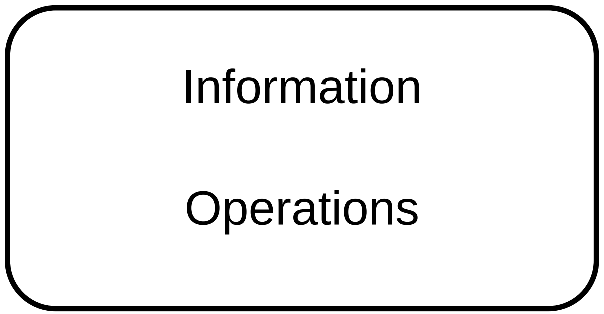
Object-oriented [design] is the method which bases the architecture of any software system on the objects it manipulates (rather than “the” function it is meant to ensure).

Ask not first what the system does: ask what it does it to!

— Meyer, OOSC, p. 116

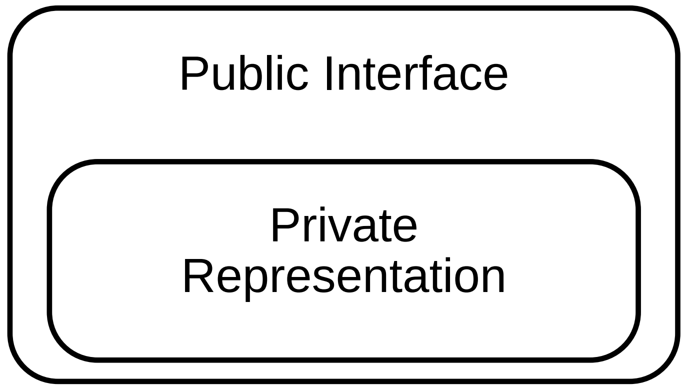
Encapsulation and Information Hiding

Encapsulation is the bundling together of related entities. Objects encapsulate information and the operations that may be performed with the information.



First abstract related functionality and information, and encapsulate them in an object.

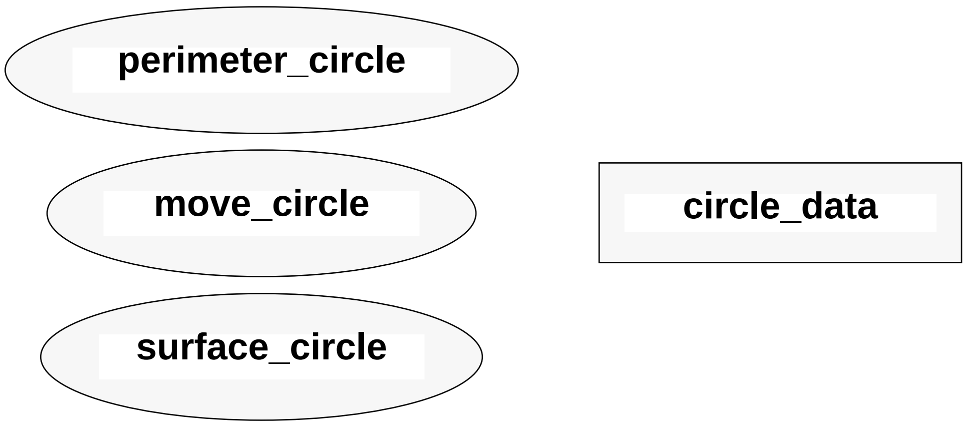
Information hiding distinguishes the *ability* to perform some action from the specific steps taken to do so. Objects reveal these abilities through a public interface.



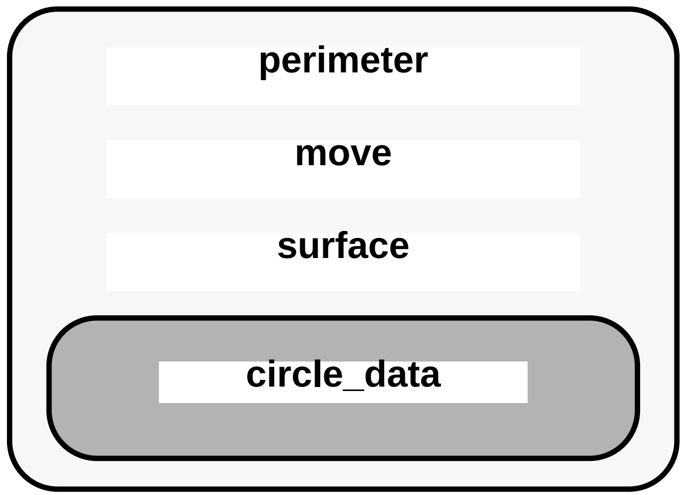
Then decide what functionality and information is required by other objects and hide the rest.

Example: Circle Class

CONVENTIONAL APPROACH



OBJECT ORIENTED APPROACH



The Promise of Object-Orientation

Data abstraction:

- Clients are protected from variations in implementation

Compatibility:

- Software components can be defined with plug-compatible interfaces

Decomposition:

- Groups of related classes form natural units for software development

Reuse:

- Classes are a convenient way to bundle methods and data as reusable software

Extensibility:

- software frameworks can be extended by inheritance
- classes form loosely coupled structures that are easier to modify

Maintenance:

- classes and inheritance limit the effects of changes

Problems with Object-Orientation

Focus on code:

- ❑ too much emphasis on language; too little on development process

Difficult to find the objects:

- ❑ software objects are not “real” objects; many ways to partition

Function-oriented methods are not appropriate:

- ❑ focus on specific needs rather than domain modelling

Management changes:

- ❑ different roles are required; more emphasis on reuse

Transition is risky:

- ❑ object-orientation requires a major “paradigm shift”

Object-Oriented Methods

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*.

Unified Modeling Language

The “Unified Modeling Language” (UML) is an attempt to unify the Booch and OMT object-oriented analysis and design methods. The modelling concepts and notation are bound to become an industry standard for documenting object-oriented models.

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

and others ...

Summary

You should know the answers to these questions:

- ❑ Why is feedback needed between software development phases?
- ❑ What is the difference between analysis and design?
- ❑ Why plan to iterate? Why develop incrementally?
- ❑ Why should requirements and analysis models be feature-oriented?
- ❑ Why should design and implementation models be data/object-oriented?
- ❑ 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?
- ❑ Why is a common notation useful for specifying analysis and design models?

Can you answer the following questions?

- ✎ *Why do requirements change?*
- ✎ *How can you validate that an analysis model captures users’ real needs?*
- ✎ *When does analysis stop and design start?*
- ✎ *When can implementation start?*
- ✎ *What kinds of projects call for object-oriented methods? Which don’t?*

3. Modelling Objects and Classes

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

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.

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

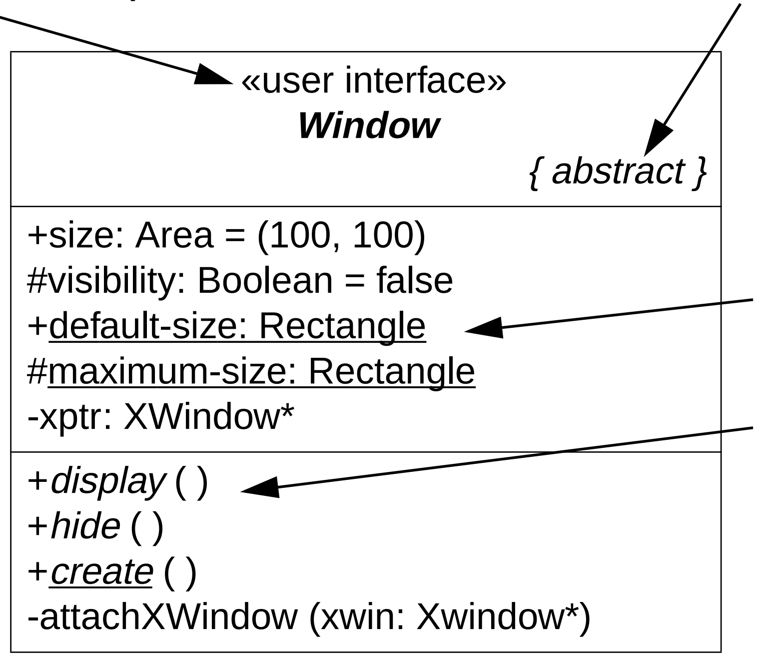
Attributes and operations are also collectively called *features*.

Visibility and Scope of Features

Stereotype
(what "kind" of class is it?)

User-defined properties
(e.g., abstract, readonly, owner = "Pingu")

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



<u>underlined</u> attributes have class scope

italic attributes are abstract

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

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

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

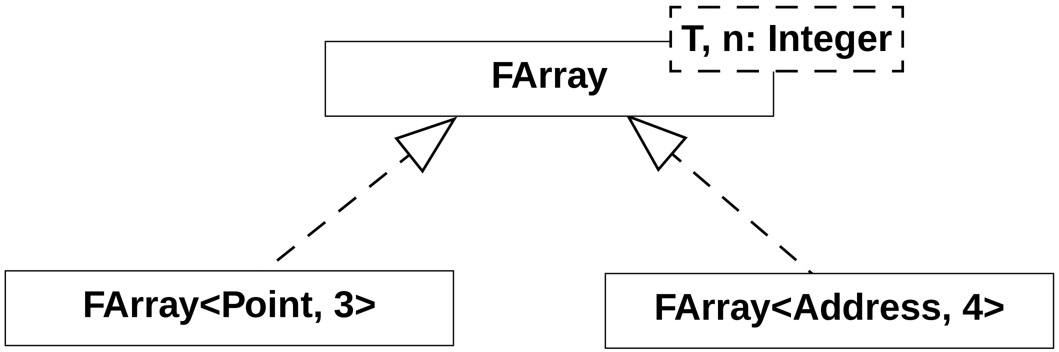
Aggregation
i.e., "consists of"

"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**).

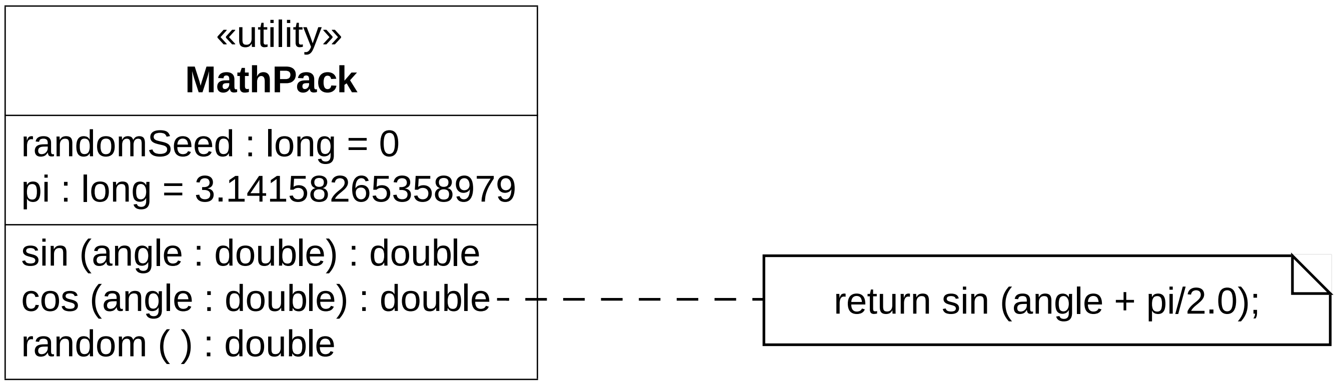


Instantiation of a class from a template can be shown by a dashed arrow.

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

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.

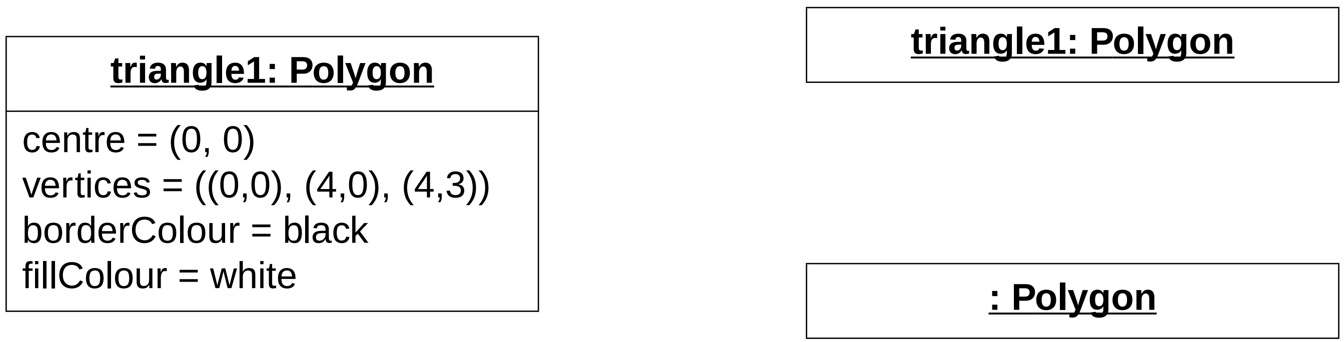


NB: A utiliy’s attributes are already interpreted as being in class scope, so it is redundant to underline them.

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

Objects

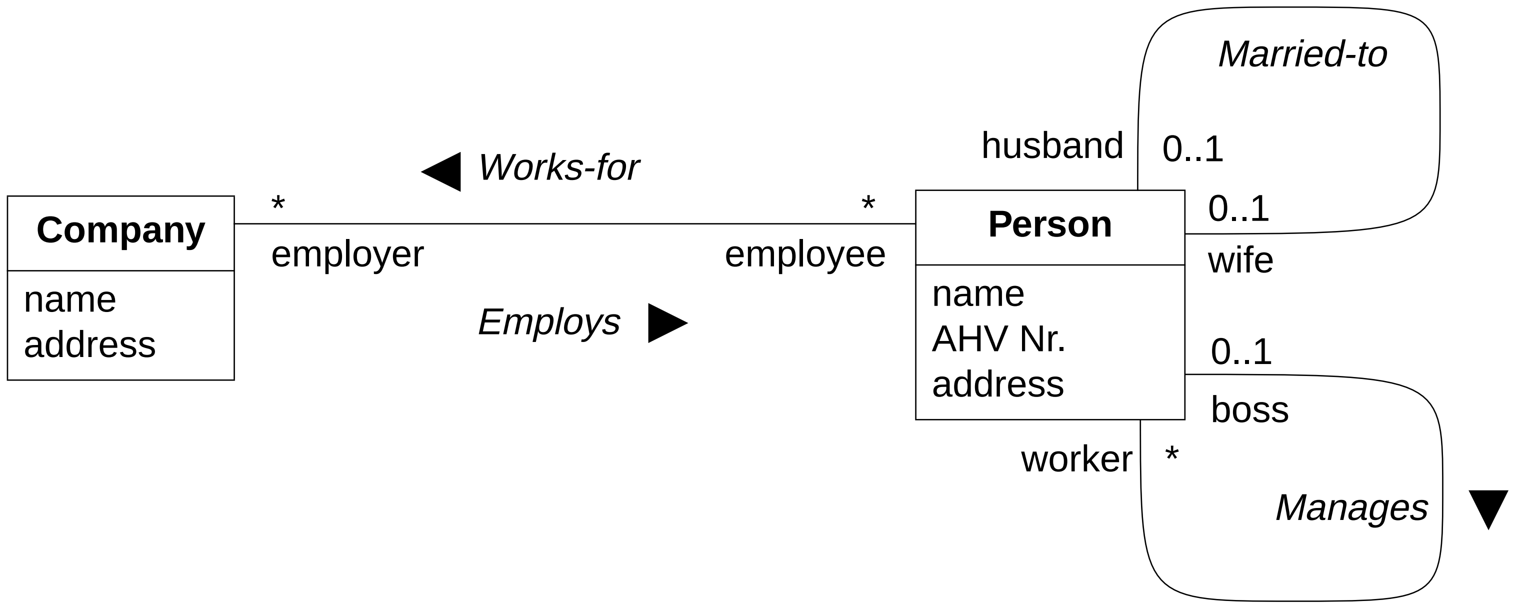
Objects are shown as rectangles with their name and type underlined in one compartment, and attribute values, optionally, in a second compartment.



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

Associations

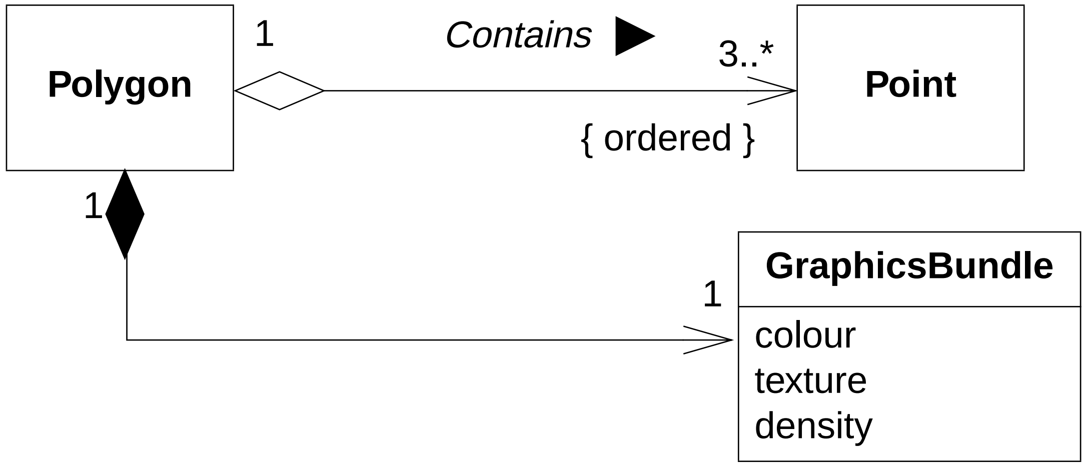
Associations represent structural relationships between objects of different classes.



- 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:



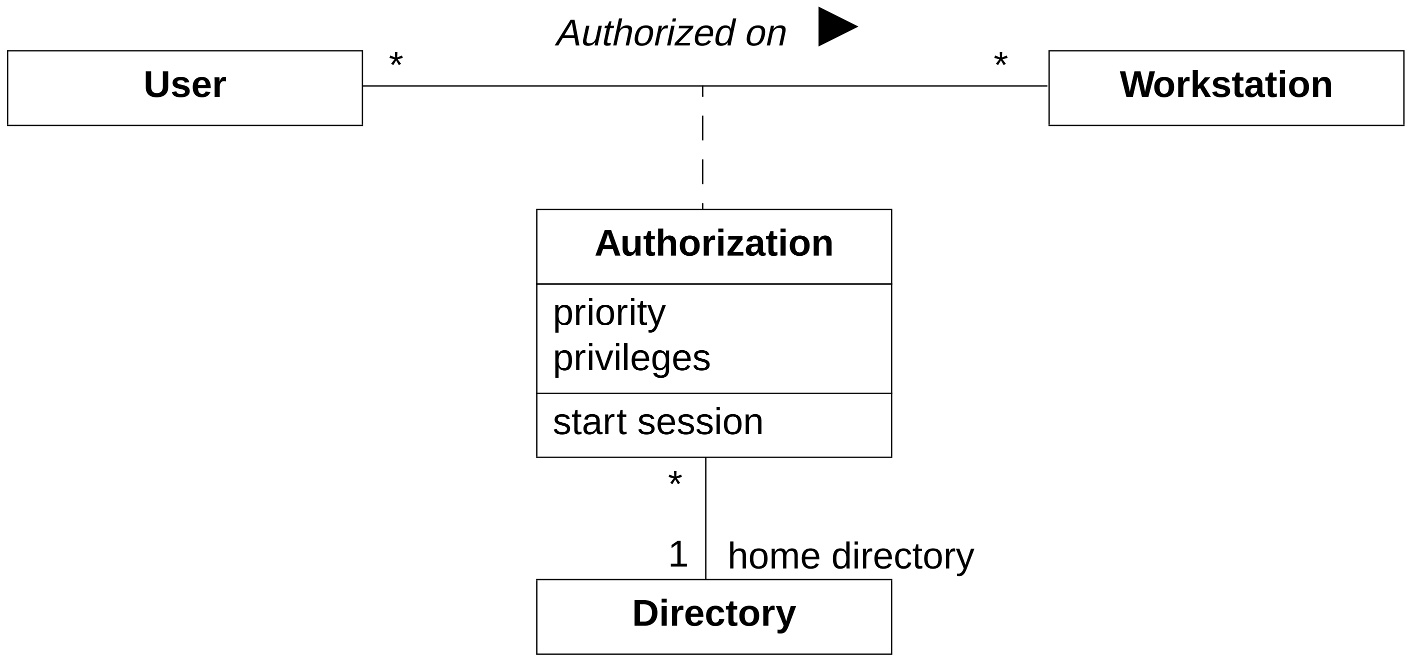
A hollow diamond indicates a reference; a solid diamond an implementation.

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

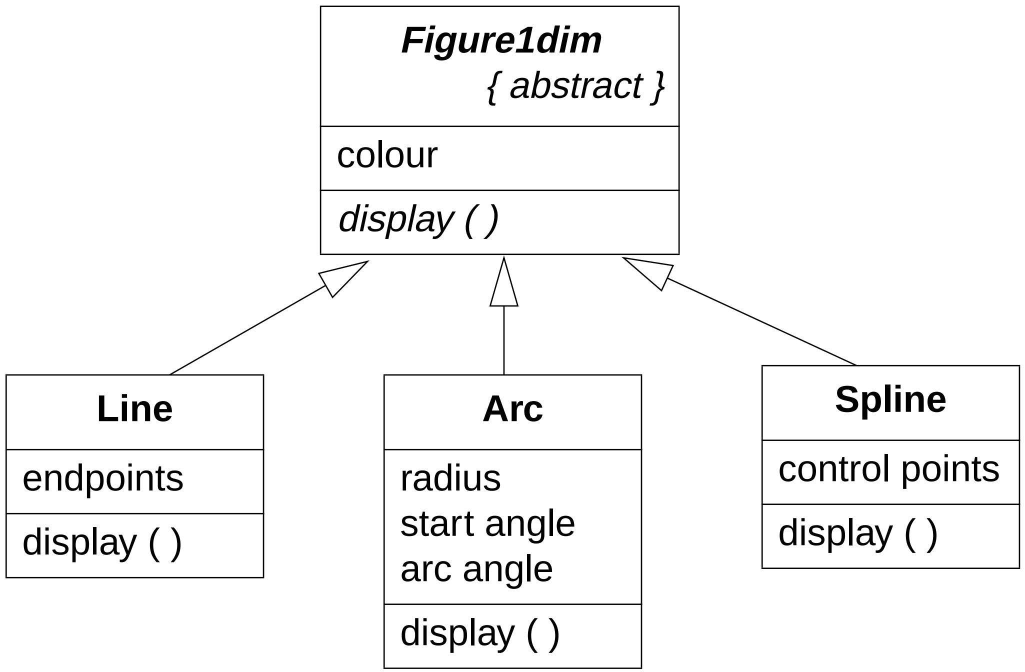
A qualified association uses a special qualifier value to identify the object at the other end of the association:

"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:

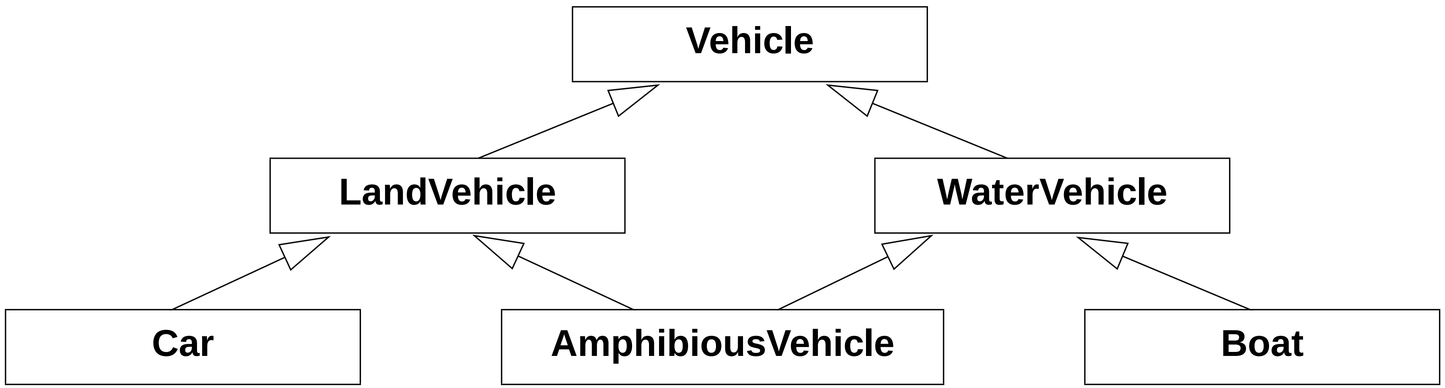
- related classes may share interfaces, data structures or behaviour
  - geometric objects ...

Polymorphism:

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

Multiple Inheritance

A class may inherit features from multiple superclasses:



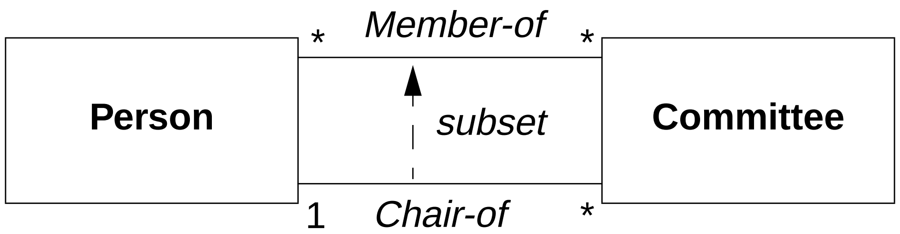
In Eiffel, features inherited from common parents are *shared* unless they have been renamed along one of the inheritance paths. Such features are considered *replicated*.

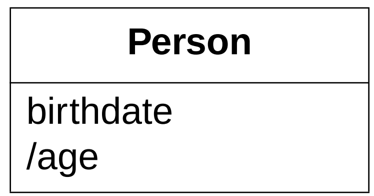
Other languages may adopt other rules to resolve inheritance conflicts.

Constraints

Constraints are restrictions on values attached to classes or associations.

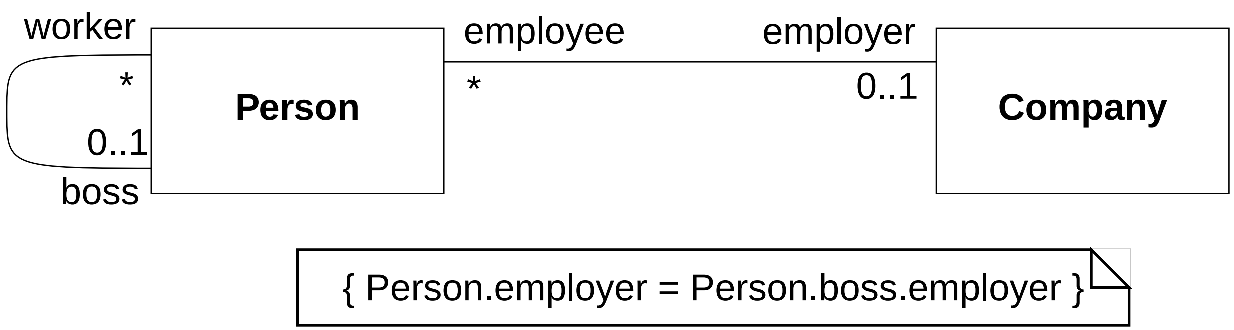
- Binary constraints may be shown as dashed lines between elements
- Derived values and associations can be marked with a “/”





{ age = currentDate - birthdate }

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



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 <u>data dictionary</u> 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?*

4. Modelling 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.

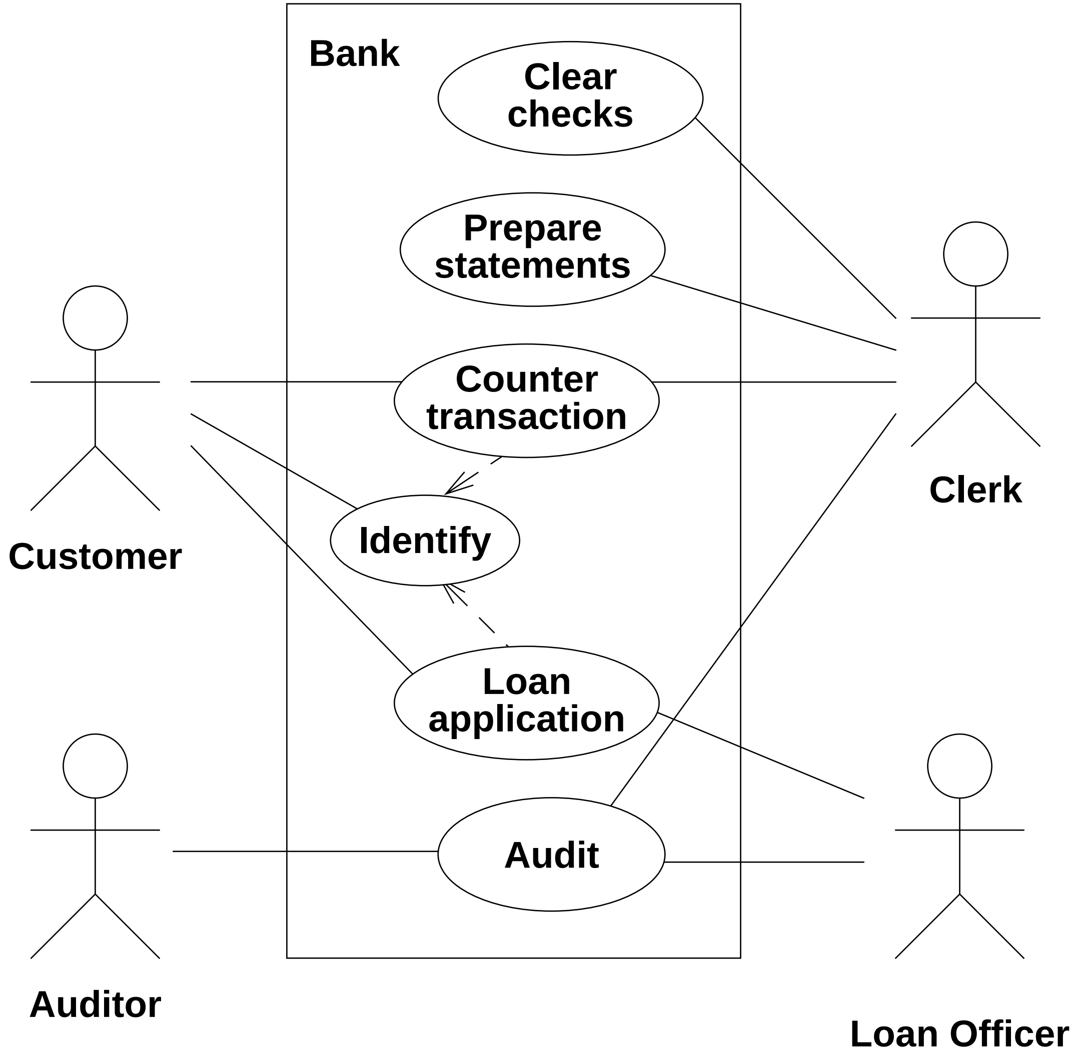
Use Case Diagrams

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.



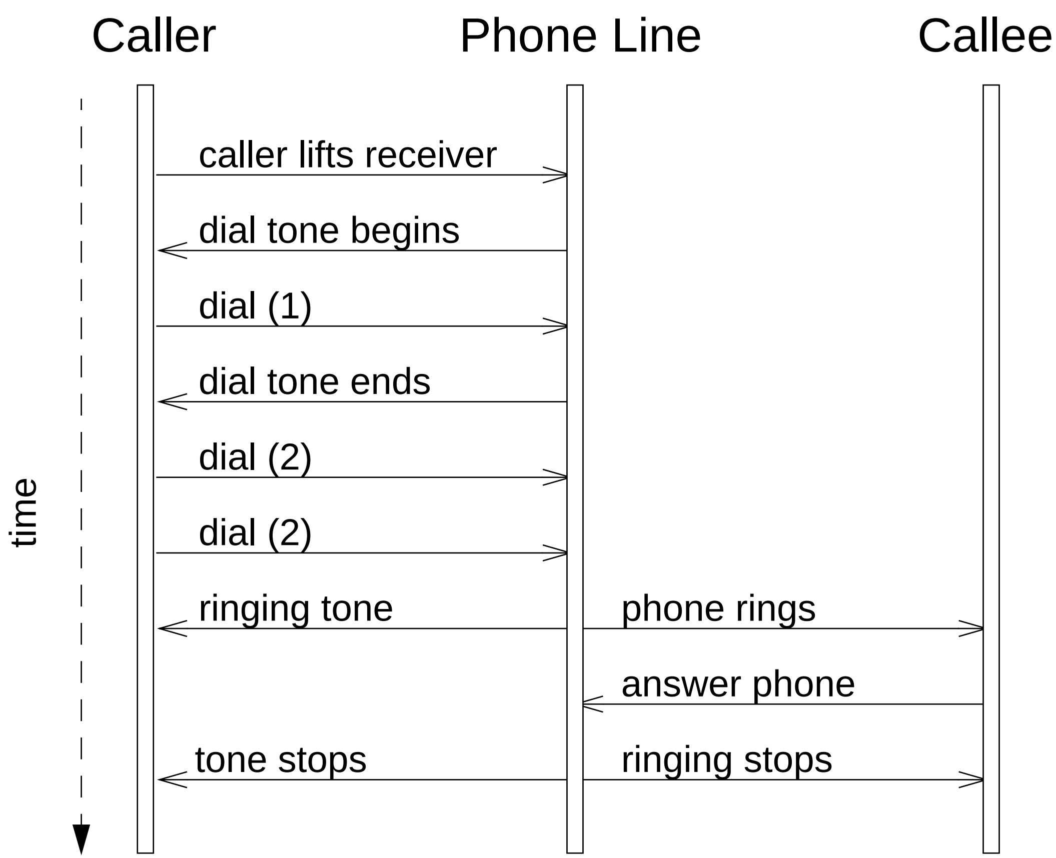
Sequence Diagrams

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

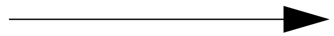
Objects (not classes!) are shown as vertical bars.

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!

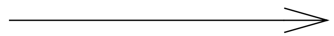


UML Message Flow Notation



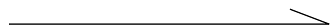
Filled solid arrowhead

procedure call or other nested control flow



Stick arrowhead

flat, sequential control flow (usually asynchronous)

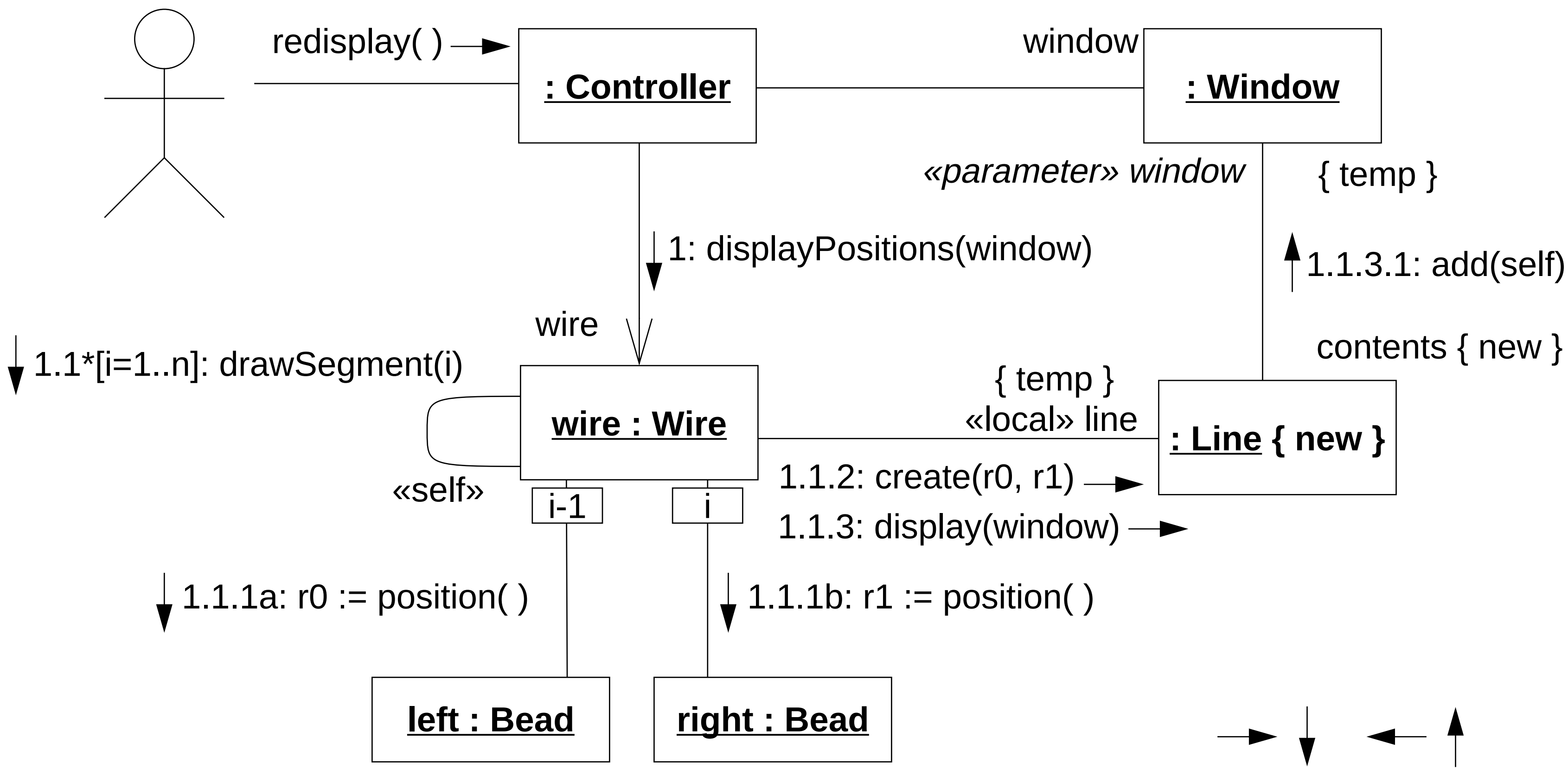


Half-stick arrowhead

asynchronous control flow between objects within a procedural sequence

Collaboration Diagrams

Collaboration diagrams depict scenarios as flows of messages between objects:



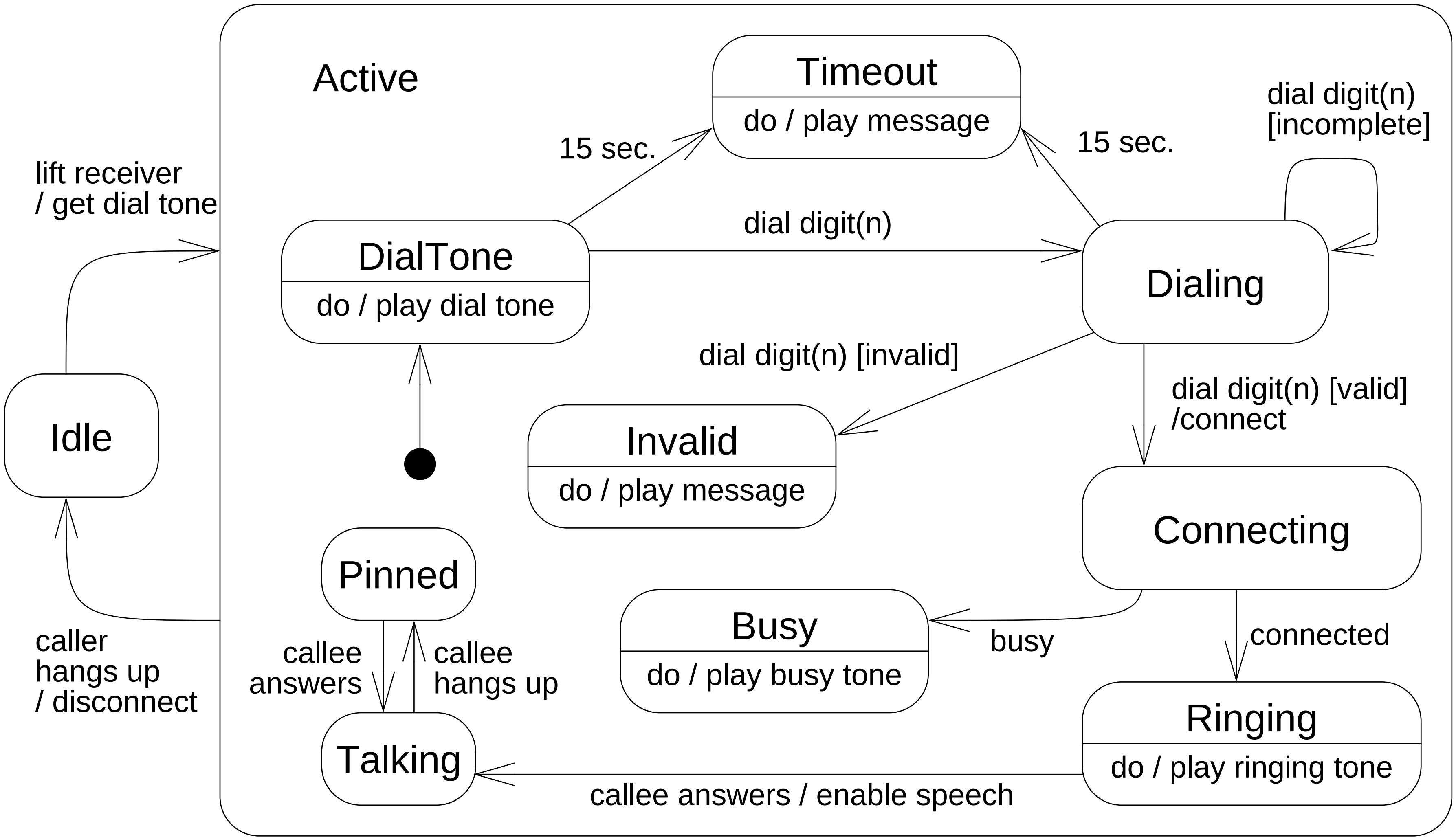
Message Labels

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.

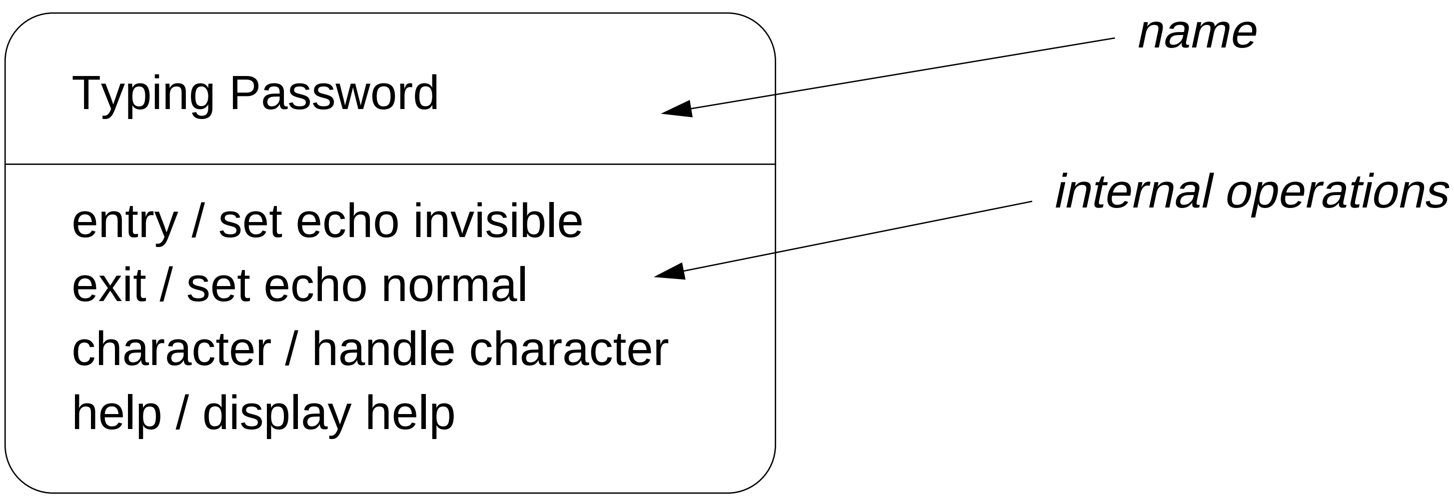
An *event* is a one-way (asynchronous) communication from one object to another:

- 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]
  ^target.sendEvent (arguments)
  / operation (arguments)
- *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

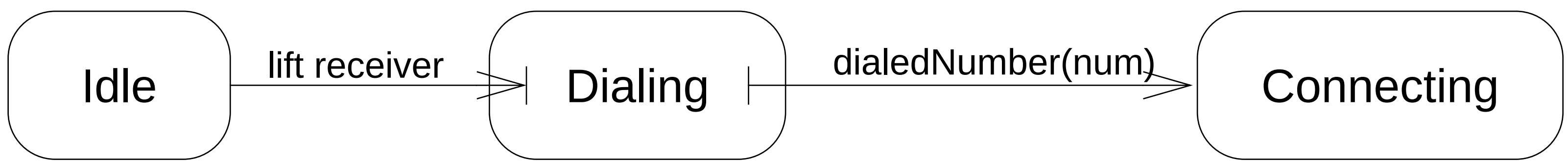
Activities:

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

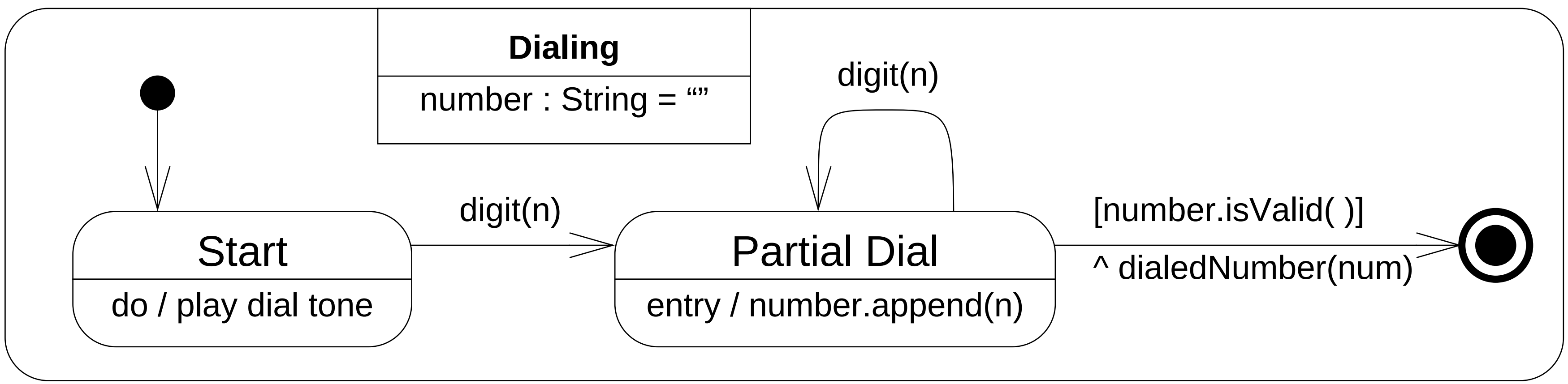
Composite States

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

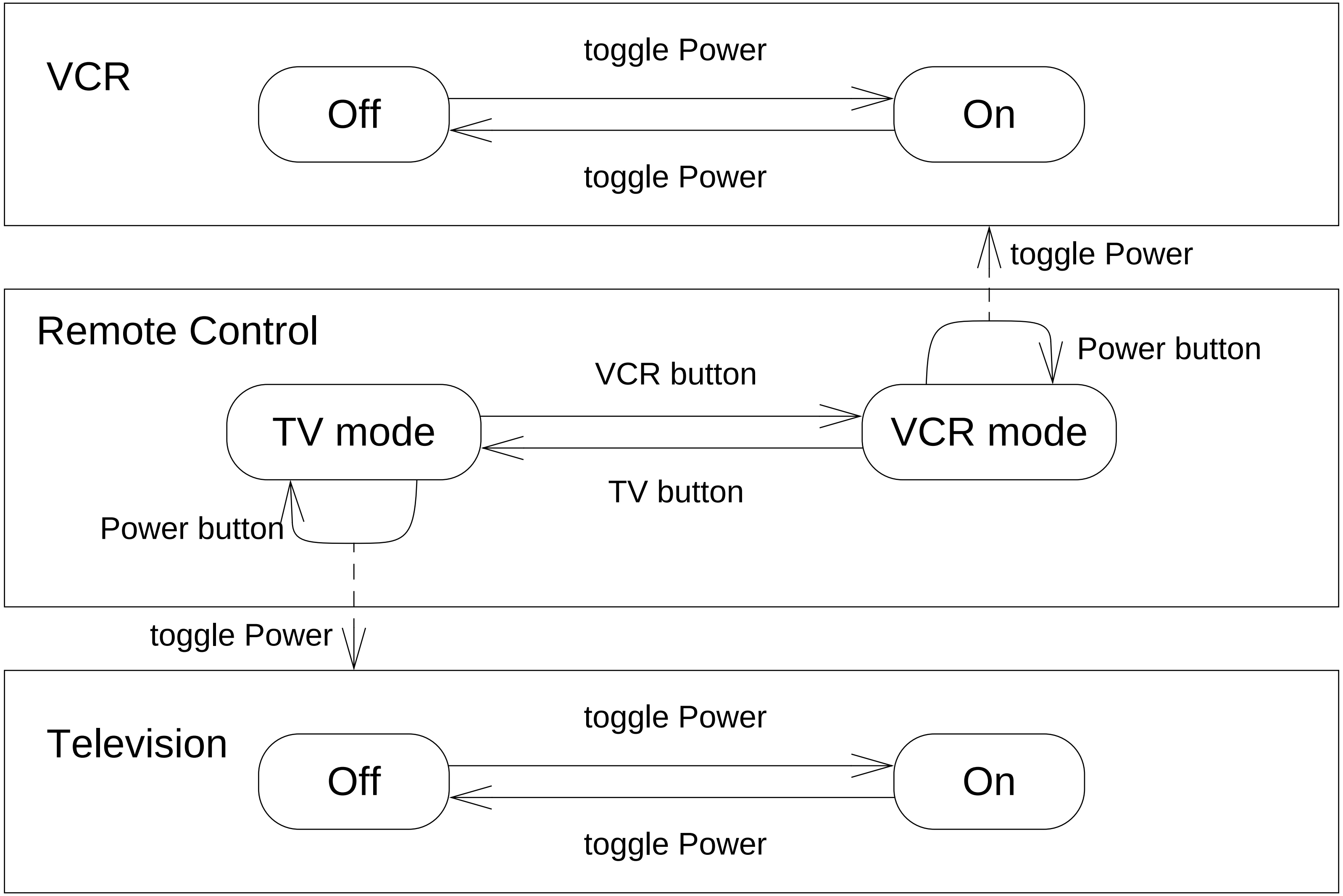
To indicate the presence of internal states, “stubbed transitions” may be used in the high-level view:



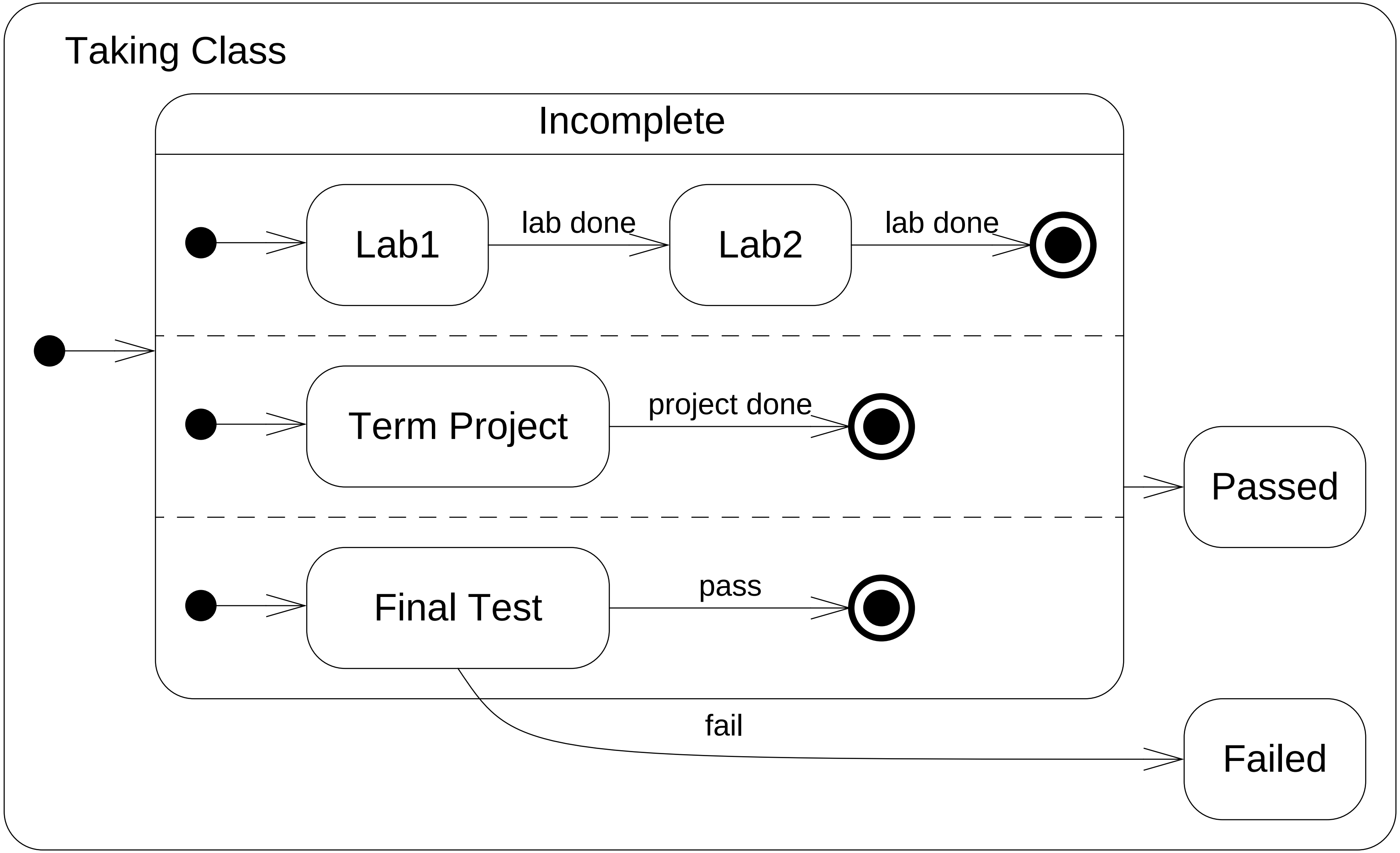
Starting and termination substates are shown as black spots and “bulls-eyes”:



Sending Events between Objects



Concurrent Substates



Branching and Merging

Entering concurrent states:

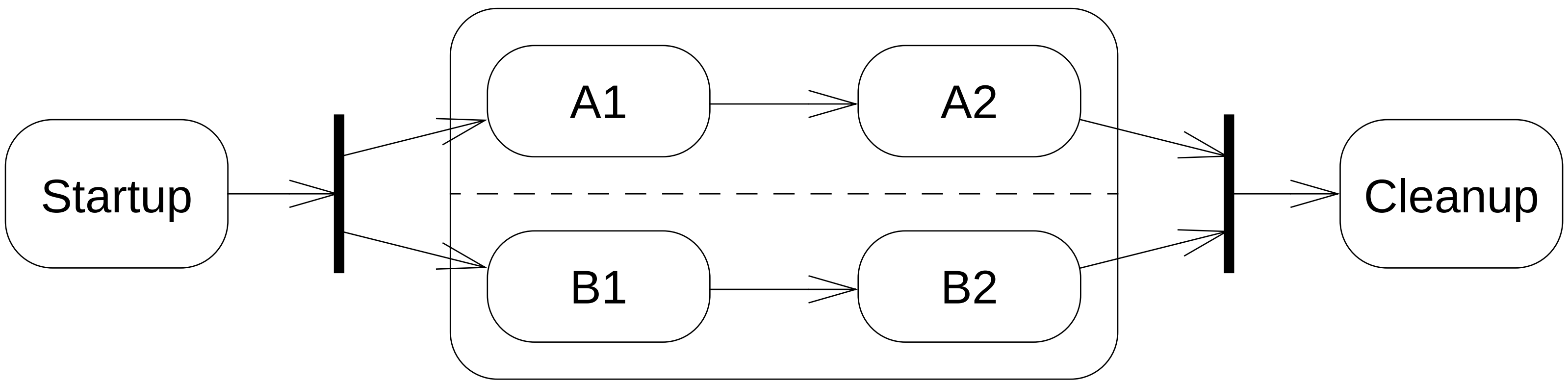
Entering a state with concurrent substates means that *each* of the substates is entered concurrently (one logical thread per substate).

Leaving concurrent states:

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

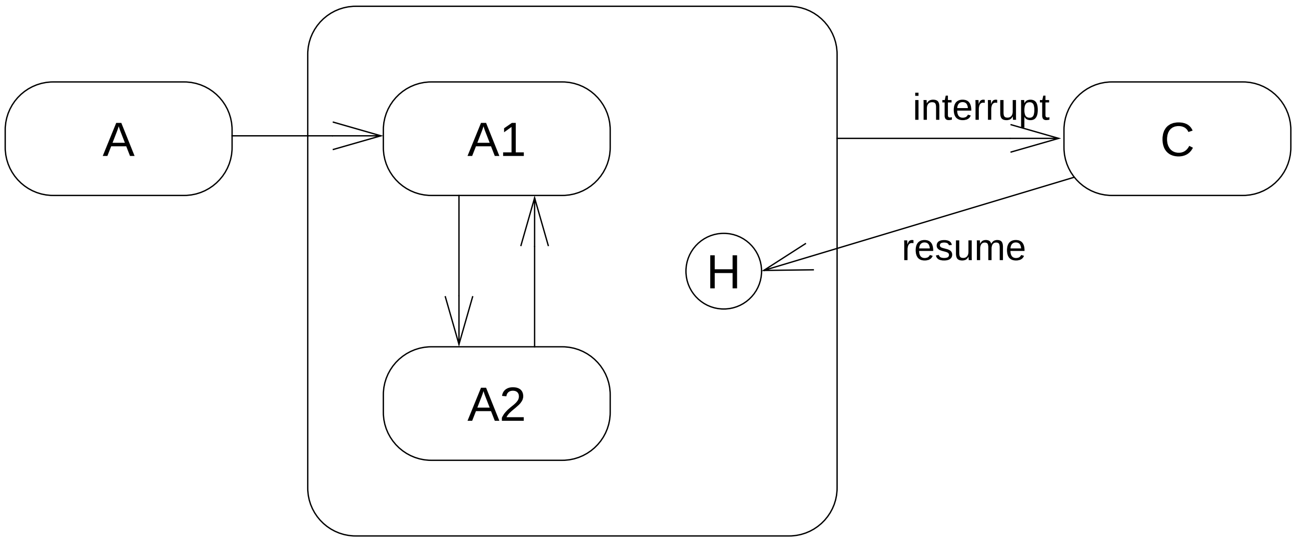
An unlabelled transition out of the overall state waits for all substates to terminate.

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



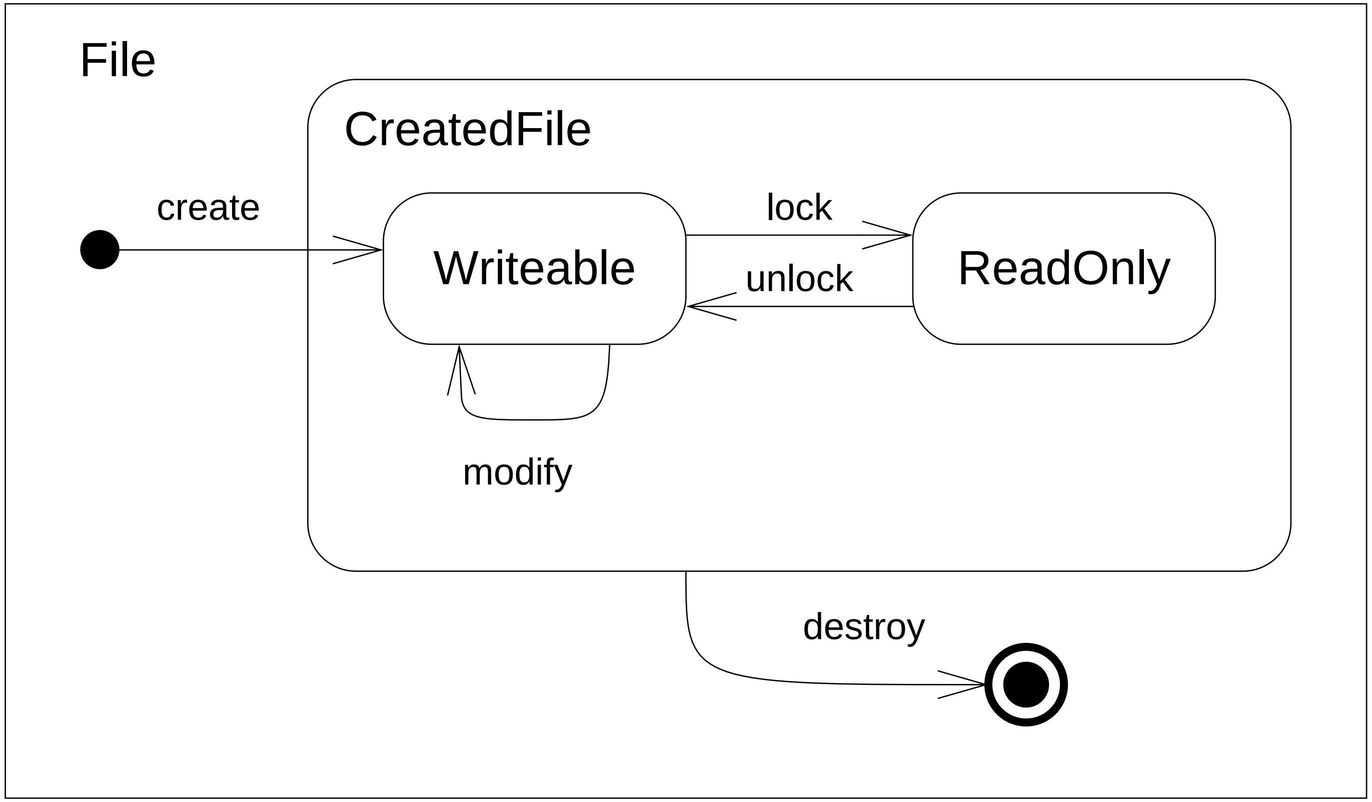
History Indicator

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?*

5. 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.

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.

<u>Class Selection Rationale (I)</u>

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~~
- ☞ <u>Ellipse Element</u>, <u>Line Element</u>, <u>Rectangle Element</u> ⇒ ~~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.

Class Cards

Use class 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

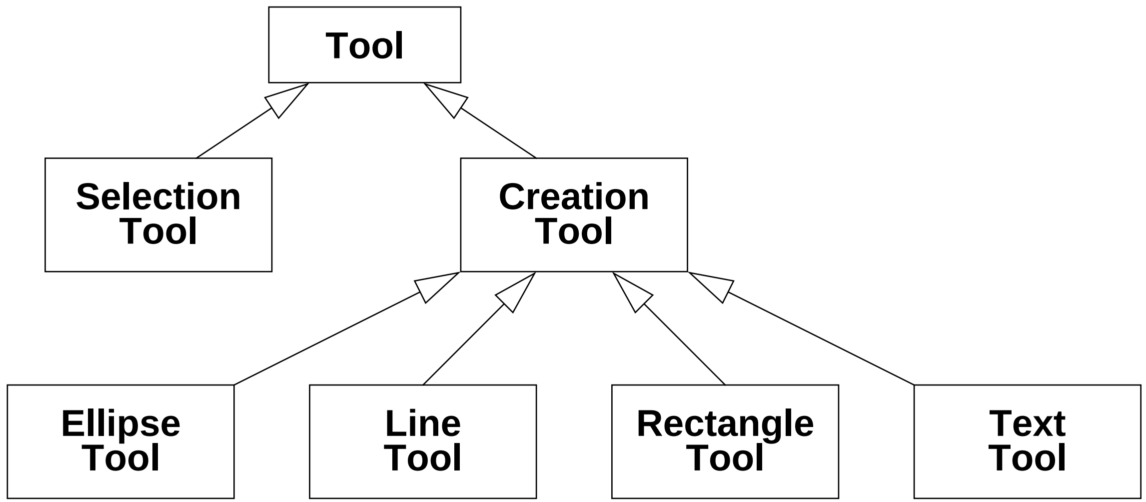
- ☞ compact, easy to manipulate, easy to modify or discard!
- ☞ easy to arrange, reorganize
- ☞ easy to retrieve discarded classes

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 $\Rightarrow$ roles $\Rightarrow$ responsibilities
  - recorded purposes on class cards $\Rightarrow$ 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?*

6. Detailed Design

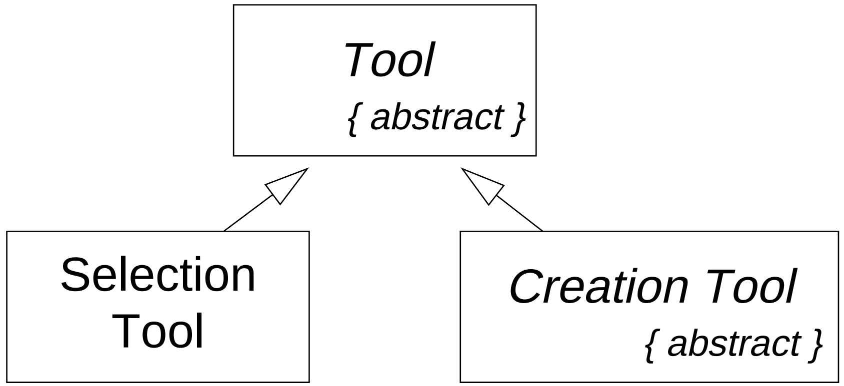
Overview:

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

Source:

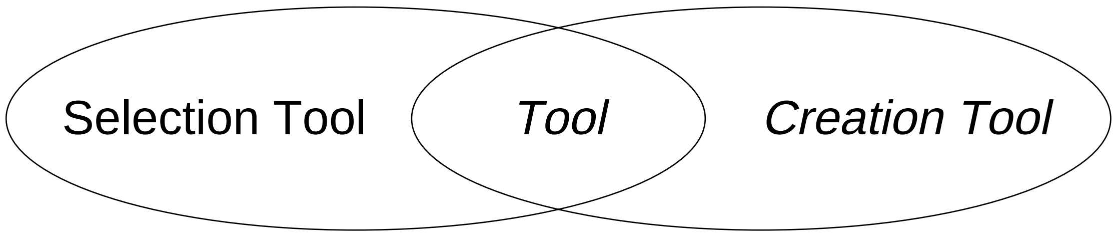
- *Designing Object-Oriented Software*, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990

Sharing Responsibilities



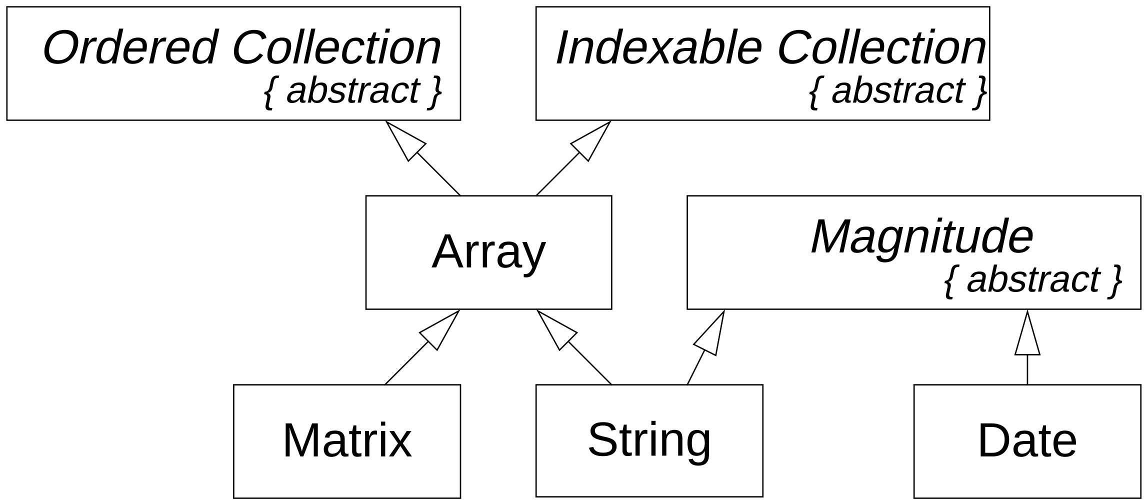
Concrete classes may be both instantiated and inherited from.
Abstract classes may only be inherited from. *Note on class cards and on class diagram.*

Venn Diagrams can be used to visualize shared responsibilities:



(Warning: not part of Unified Notation!)

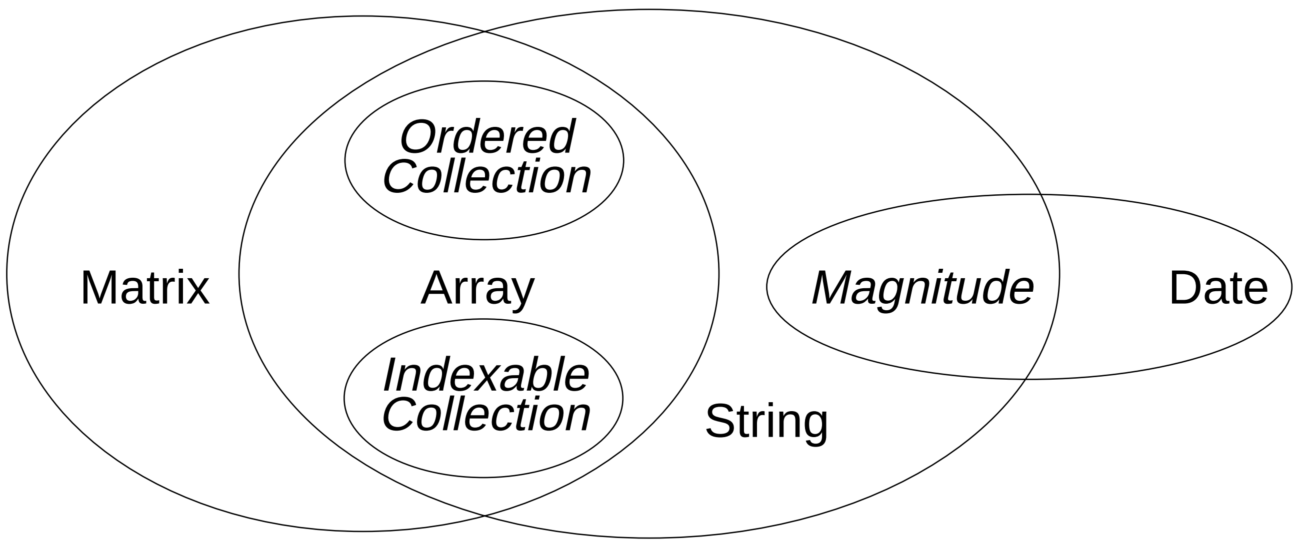
Multiple Inheritance



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:

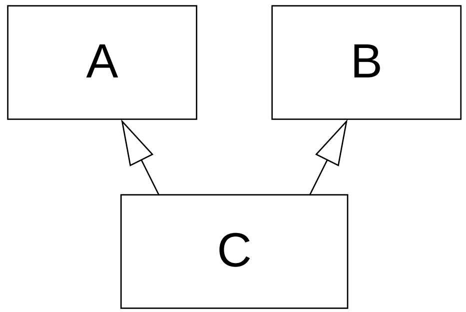
- ☞ Eliminate by introducing common abstract superclass: *abstract classes should support responsibilities in an implementation-independent way*

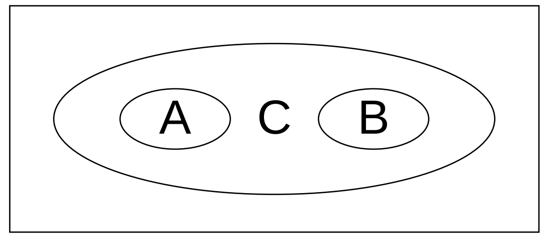
Eliminate classes that do not add functionality:

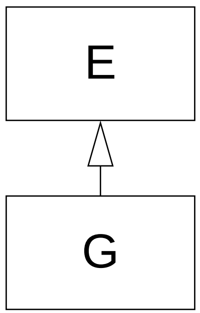
- ☞ Classes should either add new responsibilities, or a particular way of implementing inherited ones

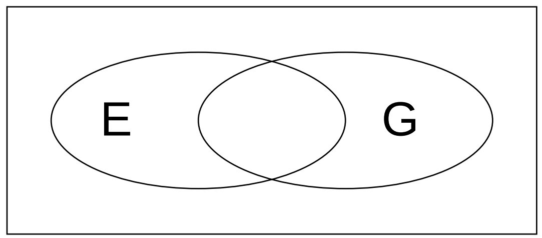
Building Kind-Of Hierarchies

Correctly Formed Subclass Responsibilities







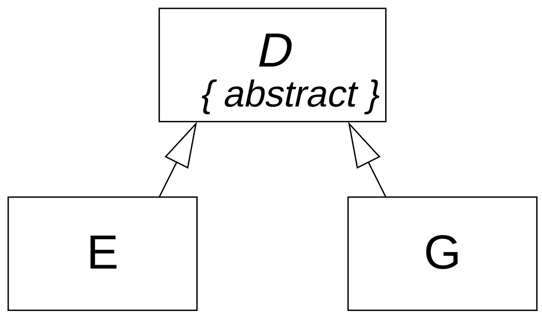


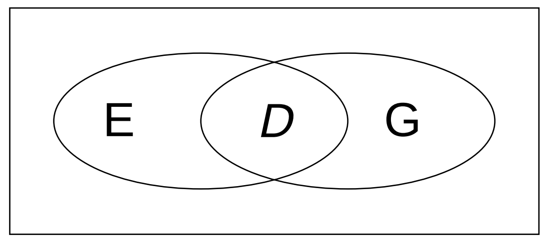
Incorrect Subclass/Superclass Relationships

Subclasses should assume *all* superclass responsibilities

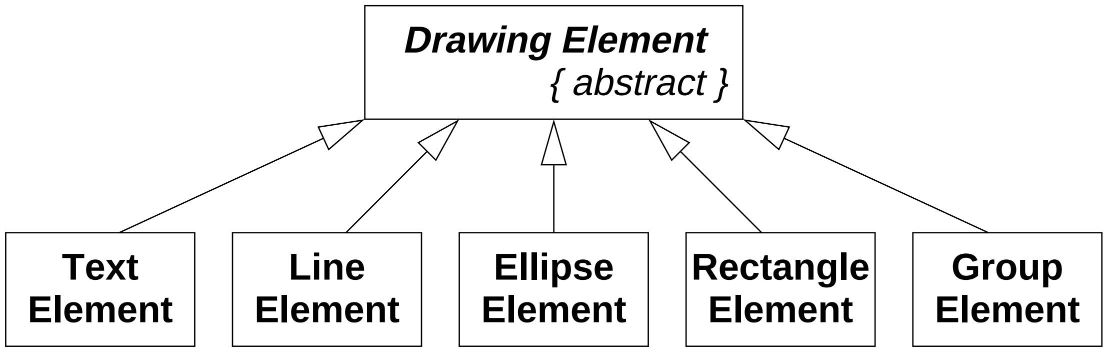
Revised Inheritance Relationships

Introduce abstract superclasses to encapsulate common responsibilities



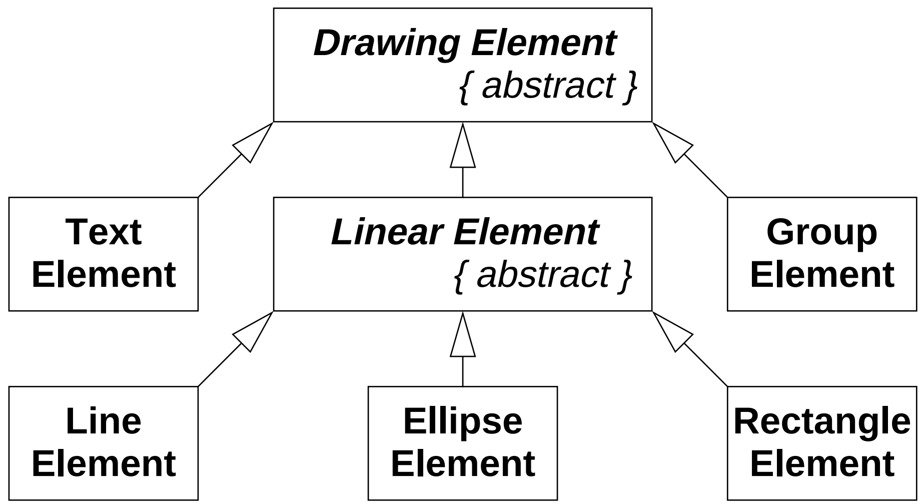


Refactoring Responsibilities



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?*

7. 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.

– alternative 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

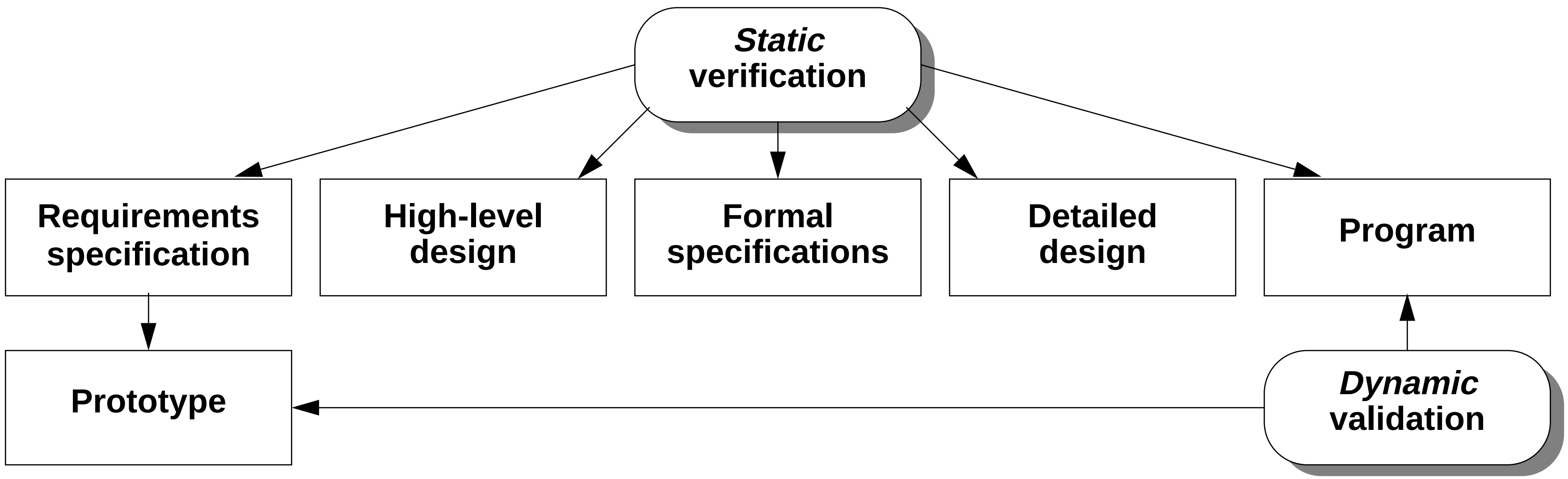
Verification and Validation

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* ...

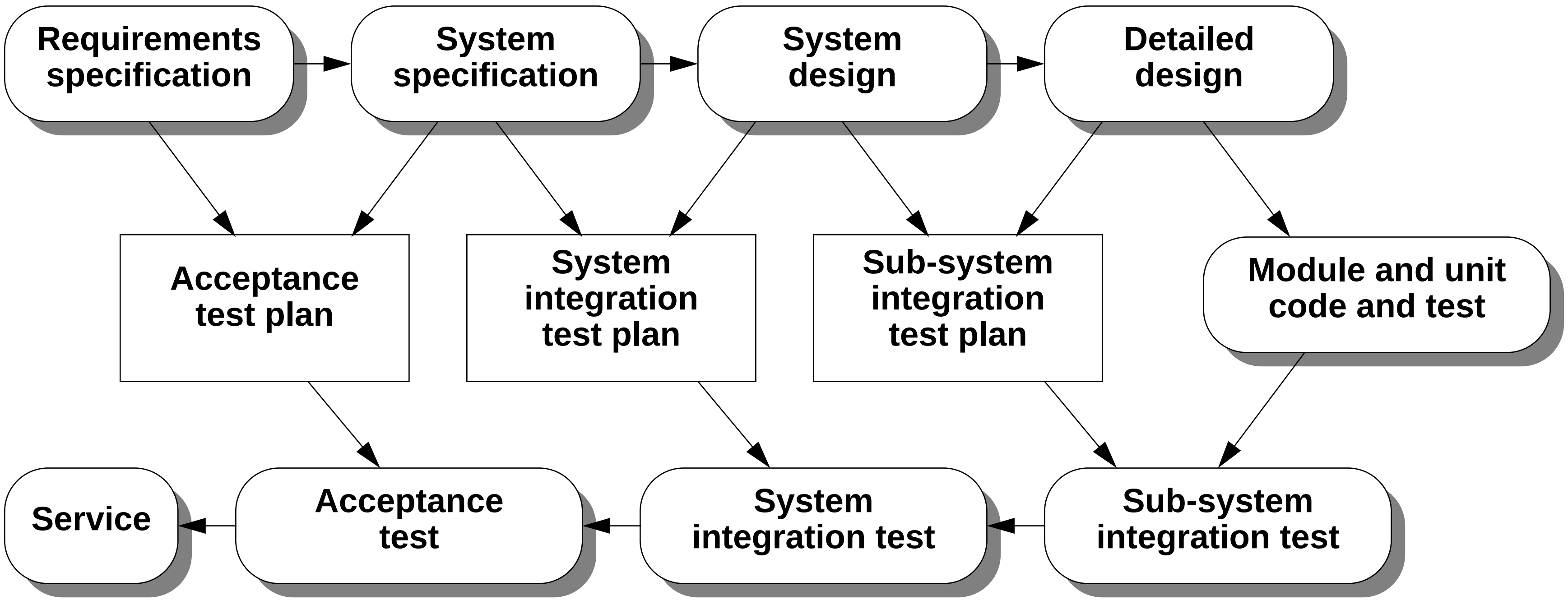
The Testing Process

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.

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:

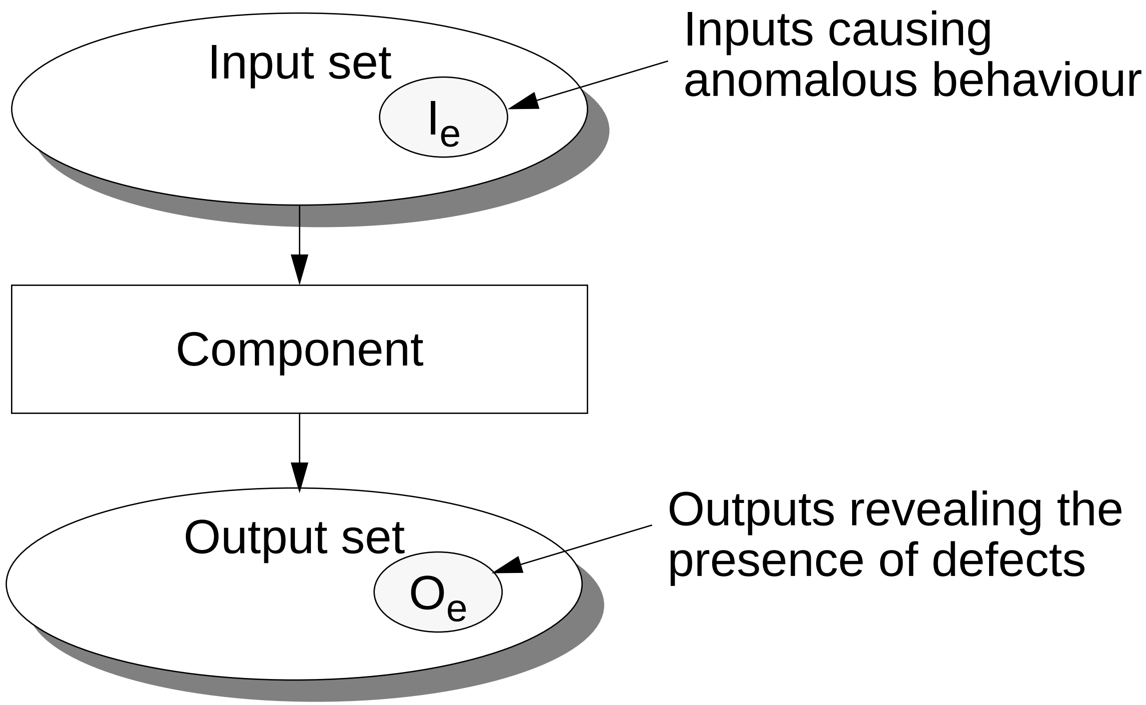
1. "Testing a system's capabilities is more important than testing its components."
   - ☞ Choose test cases that will identify situations that may prevent users from doing their job.

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.

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:

```
feature {ANY}
   find (key: INTEGER) : BOOLEAN is ...
feature {NONE}
   elements : ARRAY [INTEGER] -- sorted
```

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)

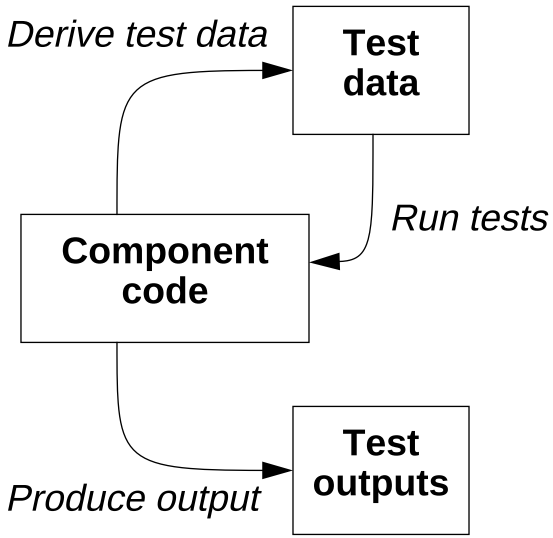
Test Cases and Test Data

Generate test data that cover all meaningful equivalence partitions.

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



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

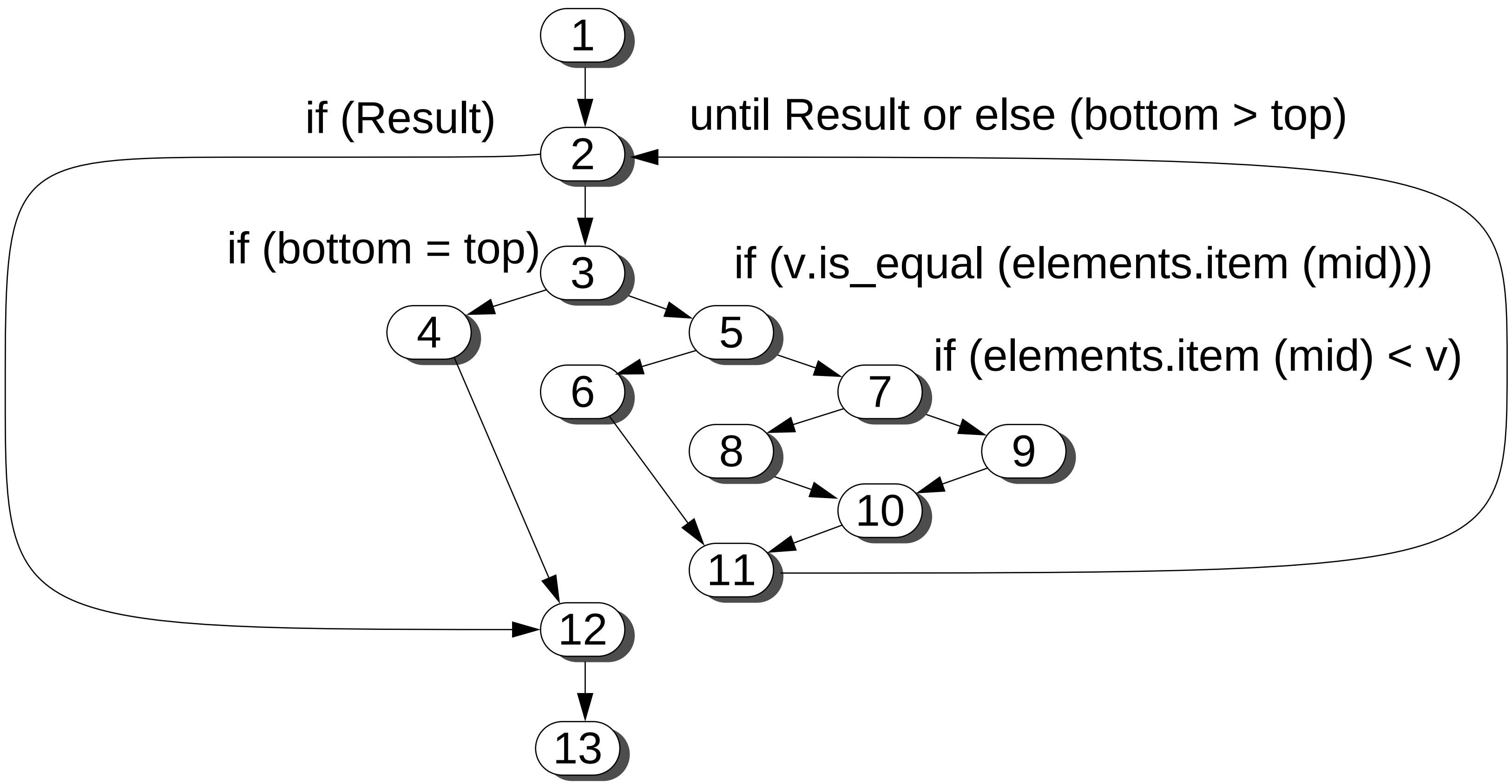
Binary Search Method

```
find (v: INTEGER) : BOOLEAN is
   -- find v in sorted array elements (an instance variable) by binary search
require        not_empty: not (empty) -- i.e., not(upper<lower)
local          bottom, top, mid : INTEGER
do
   from        bottom := lower     -- lower index of elements array
               top := upper        -- upper index of elements array
               last_index := (bottom + top) // 2
               Result := v.is_equal (elements.item (last_index))
   invariant   bottom <= top
   variant     top - bottom
   until       Result or else (bottom > top)
   loop
      mid := (bottom + top) // 2
      if (v.is_equal (elements.item (mid))) then
         Result := True
         last_index := mid
      else
         if (elements.item (mid) < v) then
            bottom := mid + 1
         else
            top := mid - 1
         end  -- if
      end  -- if
   end  -- loop
ensure         (Result = True) implies v.is_equal (elements.item (last_index))
end  -- find
```

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.

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

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?*

8. Design by Contract

Overview:

- Assertions
- Programming by Contract: Pre- and Post-conditions
- Class invariants and correctness
- Functions and side-effects
- Disciplined Exceptions

Source:

- *Object-Oriented Software Construction*, Second Edn., B. Meyer, Prentice Hall, 1997.

Assertions

An assertion is a property over values of program entities:

```
class STACK [T]

feature { ANY }
    numElements : INTEGER
    empty : BOOLEAN is do ... end
    full : BOOLEAN is do ... end

    pop is
        require
            not empty
        do ...
        ensure
            not full
            numElements = old numElements - 1
        end

    top : T is
        require
            not empty
        do ... end

    push (x : T) is
        require
            not full
        do ...
        ensure
            not empty
            top = x
            numElements = old numElements + 1
        end

end -- class STACK
```

Assertions are used to specify conditions which should hold at various points during program execution.

Programming by Contract

By associating **require** *pre* and **ensure** *post* to a routine *r*, a class establishes the *contract* with its clients:

> “If you promise to call *r* with *pre* satisfied, then I, in return, promise to deliver a final state in which *post* is satisfied.”

- ☞ The *precondition* binds *clients:* it defines the conditions under which a call to the routine is legitimate.
- ☞ The *postcondition*, in return, binds the *class:* it defines the conditions that must be ensured by the routine on return.

	Obligations	*Benefits*
Client Programmer	Only call `push(x)` on a non-full stack	Get `x` added as a new stack top on return (`top` yields `x`, `numElements` increased by 1)
Module Implementor	Make sure that `x` is pushed on top of the stack	No need to treat cases in which the stack is already full

Checking Preconditions

```
sqrt (x : REAL) : REAL is
   -- square root of x
   require
      x >= 0
   do ...
```

What happens if a precondition is not satisfied?

If the client fails to satisfy the precondition to a contract, the object is under no obligation to provide anything in return

- ❑ Objects should *not* check preconditions; they are the responsibility of *clients* that make requests
- ❑ Redundant checking is not only inefficient but needlessly complicates code.
- ❑ In practice, however, objects *must* check preconditions as a guard against programming errors!

✔ *Rigorous use of preconditions promotes readability, maintainability and clear assignment of responsibilities.*

Example — the STACK Class

```
class STACK [T]

creation { ANY }
   make

feature { NONE }

   contents : ARRAY [T]

   maxSize : INTEGER

   make(n :INTEGER) is
      do
         if n>0
         then
            maxSize := n
            !!contents.make(1,n)
         end
      end

feature { ANY}

   numElements : INTEGER

   empty : BOOLEAN is
      do
         Result := (numElements = 0)
      end

   full : BOOLEAN is
      do
         Result := (numElements = maxSize)
      end
```

STACK Operations ...

```
pop is
    require
        not empty
    do
        numElements := numElements - 1
    ensure
        not full
        numElements = old numElements - 1
    end

top : T is
    require
        not empty
    do
        Result := contents @ numElements
    end

push (x : T) is
    require
        not full
    do
        numElements := numElements + 1
        contents.put (x, numElements)
    ensure
        not empty
        top = x
        numElements = old numElements + 1
    end

end -- class STACK
```

Class Invariants

What are valid "stable" states of an instance of Stack?

```
class STACK [T]
   ...
feature { NONE }
   contents : ARRAY [T]
   maxSize : INTEGER
   ...
feature { ANY}
   numElements : INTEGER
   ...
end -- class STACK
```

Need:

```
invariant
   0 <= numElements; numElements <= maxSize
```

Using the Stack

```
class MAIN

creation {ANY}
    make

feature {NONE}

    myStack : STACK [INTEGER]

    make is
        do
            io.putstring ("Making stack%N")
            !!myStack.make(5)
            trypush(10)
            trypush(20)
            trypop
            trypop
            trypop          -- empty stack
            trypush(30)
            trypush(40)
            trypush(50)
            trypush(60)
            trypush(70)
            trypush(80)     -- full stack
        end
```

```
trypop is
    -- try to pop a value from myStack
    -- if an error occurs,
    -- print a message and continue

    local
        top : INTEGER
        error : BOOLEAN     -- initially False

    do
        if not error
        then
            io.putstring ("Popping ")
            top := myStack.top
            io.putint(top)
            io.putstring ("%N")
            myStack.pop
            printsize
        end

    rescue
        io.putstring ("ERROR: stack is empty%N")
        error := True
        retry
    end
```

Using the STACK ...

```
trypush (n : INTEGER) is
   -- try to push a value onto myStack; if an error occurs, print a message and continue
   local
      error : BOOLEAN     -- initially False
   do
      if not error
      then
         io.putstring ("Pushing ")
         io.putint(n)
         io.putstring ("%N")
         myStack.push(n)
         printsize
      end
   rescue
      io.putstring ("ERROR: stack is full%N")
      error := True
      retry
   end

printsize is
   local
      n : INTEGER
   do
      n := myStack.numElements
      io.putstring ("Stack has ")
      io.putint(n)
      io.putstring (" elements%N")
   end

end -- class MAIN
```

Class Correctness

Invariant rule: An assertion I is a *correct class invariant* for a class C if and only if:

- ☞ the create procedure of C, when applied to arguments satisfying its precondition in a state where attributes have their default values, yields a state satisfying I; and
- ☞ every exported routine of the class, when applied to arguments and a state satisfying both I and the routine's precondition, yields a state satisfying I.

Note:

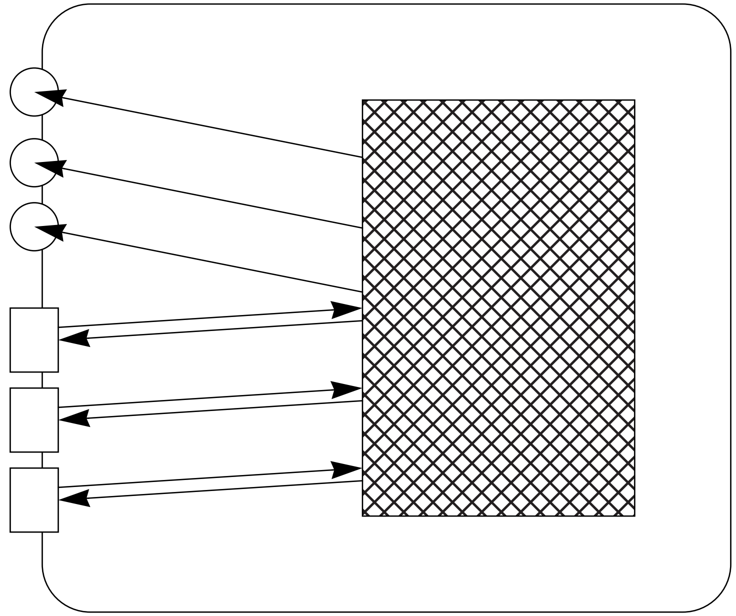
- ❑ Every class is considered to have a create procedure.
- ❑ The state of an object is defined by its attributes.
- ❑ The precondition of a routine may involve initial state and arguments.
- ❑ The postcondition may only involve the initial and final states, the arguments and the Result.
- ❑ The invariant may only involve the state.

Side Effects in Functions

Objects as Machines

Queries monitor state without altering it.

Commands alter the state of an object.

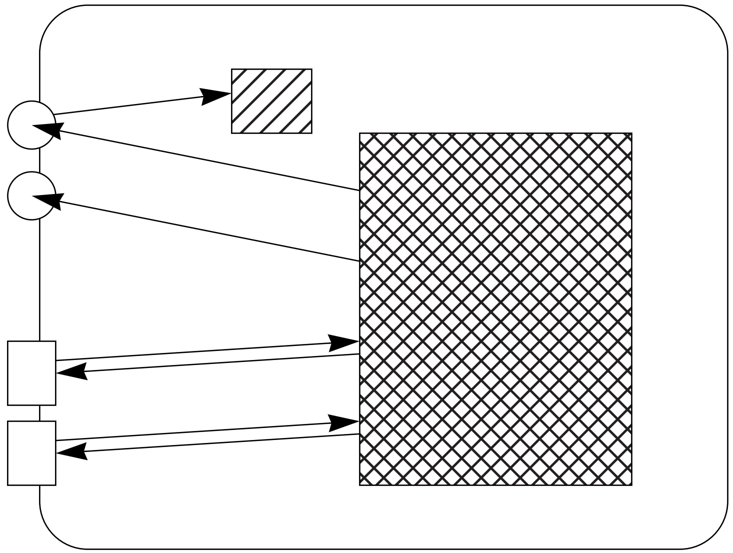


Recommended style:

Functions should be free of *visible* side-effects, so they can be used safely as queries (for example, in assertions).

Procedures should implement commands, and should *not* return results.

Legitimate Side Effects



Functions should not modify the *visible* (abstract) state of an object, but sometimes it is convenient for them to change the hidden (concrete) representation:

- ☞ caching computed queries
- ☞ switching between alternative representations
- ☞ garbage collection ...

Using Assertions

Assertions have four principle applications:

- ❑ Help in writing correct software
- ❑ Documentation aid
- ❑ Debugging tool
- ❑ Support for software fault tolerance

Correctness:

- ☞ specifying pre- and post-conditions and invariants is a conceptual aid to developing correct software in the first place
- ☞ assertions can be used to *prove* software correct

Documentation:

- ☞ concise and unambiguous specification of contract to clients of a module

Exceptions

Assertions can be checked and exceptions caught at run-time:

- ☞ debugging
- ☞ failure recovery
- ☞ fault tolerance

Three levels of checking:

1. no checking
2. checking pre-conditions only (the default)
3. checking all assertions

Disciplined Exceptions

An *exception* is the occurrence of an abnormal condition during the execution of a software element.

A *failure* is the inability of a software element to satisfy its purpose.

An *error* is the presence in the software of some element not satisfying its specification.

When an assertion is violated at run-time, an exception is raised.

There are only two reasonable courses of action:

1. clean up the environment and report *failure* to the client (“organized panic”)
2. attempt to change the conditions that led to failure and *retry*

It is not acceptable to return control to the client without special notification.

✔ *If it is not possible to run your program without raising an exception, then you are abusing the exception-handling mechanism!*

Rescue and Retry

```
routine is
   require ...
   local ...
   do
      body
   ensure ...
   rescue
      rescue clause
      if ...
      then retry
      end
   end
```

successful termination

failure

A routine execution *fails* (in Eiffel) if an exception occurs during its execution and the routine terminates by executing its rescue code.

Rescue rule: The rescue clause must be correct with respect to the precondition **true** and (except for a branch ending in a **retry**) to the postcondition given by the class invariant.

Summary

You should know the answers to these questions:

- ❑ What is an assertion?
- ❑ How are contracts are formalized by pre- and post-conditions?
- ❑ What is a class invariant and how can it be specified?
- ❑ What are assertions useful for?
- ❑ How can exceptions be used to improve program robustness?
- ❑ What situations may cause an exception to be raised?
- ❑ What kind of activity should you perform in a `rescue` clause?

Can you answer the following questions?

- ✎ *How would you apply disciplined exceptions in C++?*
- ✎ *How about in a language with no exception handling mechanism?*
- ✎ *How do you know if you have correctly specified the class invariant?*

9. Design Patterns

Overview:

- What are (not) Design Patterns?
- How are they specified?
- Common OO Design Techniques
- Example: the Template Method pattern
- What problems do Design Patterns solve?

Source:

- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns — Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, MA, 1995
- Douglas C. Schmidt, “Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software,” *Communications of the ACM*, Vol. 38, No. 10, Oct. 1995
- Christopher Alexander, et al., *A Pattern Language — Towns · Buildings · Construction*, Oxford University Press, 1977

What are Design Patterns?

Patterns were first systematically catalogued in the domain of architecture:

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*
>
> *Alexander, et al., A Pattern Language*

Software design patterns document standard solutions to common design problems:

> *"Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively."*
>
> *Gamma, et al., Design Patterns*

What Design Patterns are not ...

Algorithms are not design patterns

- ☞ algorithms solve computation problems, not design problems
- ☞ *merge-sort* is an algorithm; *divide and conquer* is a design pattern

Software components are not design patterns

- ☞ design patterns describe a *way* of solving a problem
- ☞ design patterns document pros and cons of different implementations
- ☞ software components may be implemented using design patterns

Frameworks are not design patterns

- ☞ a framework implements a generic software architecture using an object-oriented language
- ☞ a design pattern documents the solution to a *specific* design problem
- ☞ a framework may use and be documented with design patterns
- ☞ like frameworks, design patterns are drawn from experience with multiple applications solving related problems

How are Design Patterns Specified?

1. **Pattern Name and Classification:** should convey *essence* of pattern
   - ☞ *Also Known As:* other common names
2. **The Problem Forces:** describes when to apply the pattern
   - ☞ *Intent:* short statement of rationale and intended use
   - ☞ *Motivation:* a problem scenario and example solution
   - ☞ *Applicability:* in which situations can the pattern be applied
3. **The Solution:** abstract description of design elements
   - ☞ *Structure:* class and scenario diagrams
   - ☞ *Participants:* participating classes/objects and their responsibilities
   - ☞ *Collaborations:* how participants carry out responsibilities
4. **The Consequences:** results and trade-offs of applying the pattern
   - ☞ *Implementation:* pitfalls, hints, techniques, language issues
   - ☞ *Sample Code:* illustrative examples in C++, Smalltalk etc.
   - ☞ *Known Uses:* examples of the pattern found in real systems
   - ☞ *Related Patterns:* competing and supporting patterns

Common Design Techniques

Design patterns make use of many common design techniques:

- Class vs. Interface inheritance
  - Class inheritance supports sharing of implementation
  - Interface inheritance supports polymorphism
- Program to an interface, not an implementation!
  - Increase flexibility by declaring variables of abstract, not concrete classes
  - Localize knowledge concerning which concrete classes to instantiate
- Inheritance vs. Object Composition
  - Inheritance occurs statically, and exposes parent class implementation
  - Object composition occurs dynamically, and increases run-time flexibility
- Delegation vs. Inheritance
  - An object can “implement” a service by delegating it to another object
  - Delegation increases flexibility by allowing behaviour to change at run-time

Improving Design Flexibility

Many design problems are concerned with achieving flexibility:

- Varying which *classes* are instantiated
  - ☞ Create objects indirectly by delegating to a “Factory” or “Prototype” object
- Varying which *operations* are performed at run-time
  - ☞ Use polymorphism and delegation to dynamically select operations
- Varying hardware or software *platform*
  - ☞ Use polymorphism to hide implementation details from clients
- Varying object *representations* and implementations
  - ☞ Encapsulate dependencies to prevent changes from cascading
- Varying *algorithms*
  - ☞ Use polymorphism to substitute or parameterize algorithms
- *Decoupling* objects
  - ☞ Use object composition and delegation to avoid tight coupling
- Extending functionality in arbitrary ways
  - ☞ Prefer object composition and delegation to inheritance
- Adapting existing classes
  - ☞ Use object composition and delegation to hide and adapt them

Example: Template Method

Adapted from “Design Patterns,” Gamma, et al., pp. 325-330.

Name

Template Method

Intent

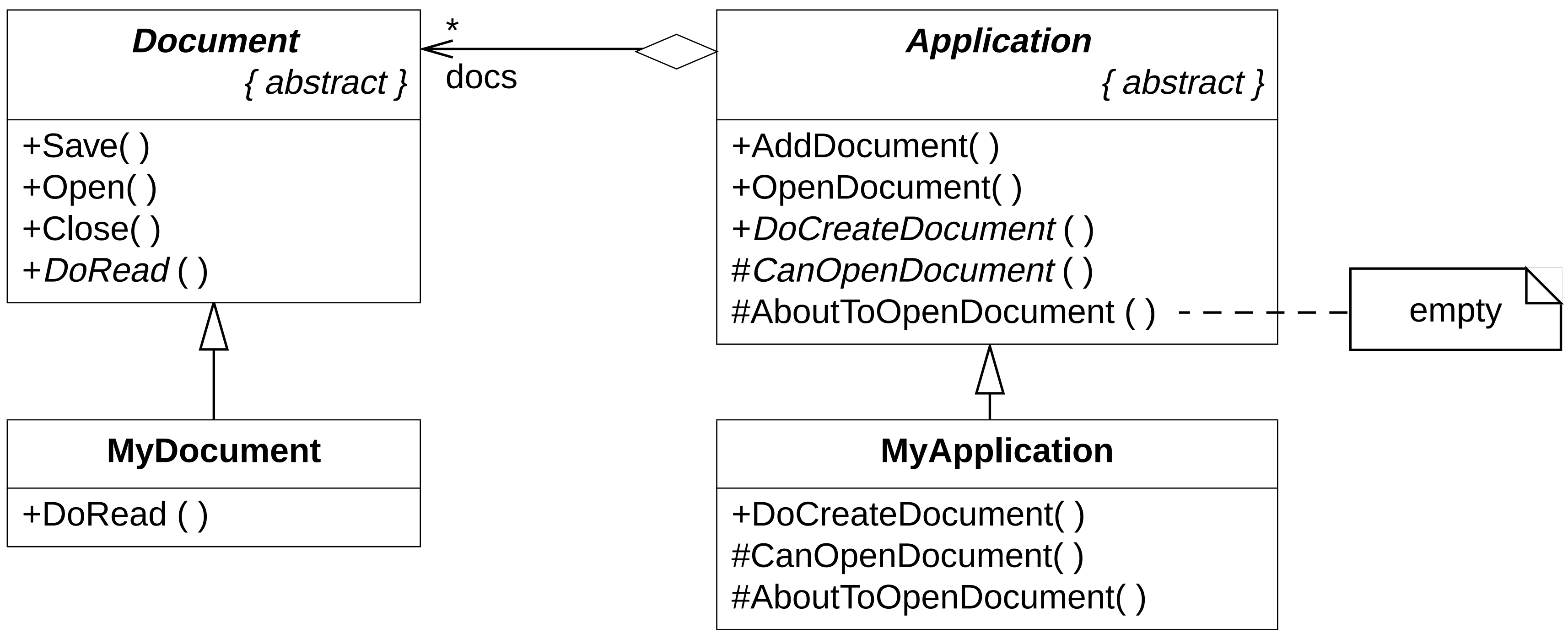
“Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.”

Template Method — Motivation

Motivation

An application framework provides `Application` and `Document` classes. `Application` is responsible for opening existing documents stored in an external format. An open document is represented by a `Document` instance.

An application built with the framework should subclass `Application` and `Document` for specific kinds of documents.



Template Method — Motivation ...

The abstract Application class defines the algorithm for opening and reading a document in its `OpenDocument` operation:

```
void Application::OpenDocument (const char* name)
{
   if (!CanOpenDocument(name)) {      // can the document be opened?
      return;
   }
   Document* doc = DoCreateDocument(name);
   if (doc) {                         // successful creation
      _docs->AddDocument(doc);
      AboutToOpenDocument(doc);       // warn Application subclass instances
      doc->Open();
      doc->DoRead();
   }
}
```

`OpenDocument` is a *template method*, since it defines an algorithm in terms of abstract operations that subclasses override to provide concrete behaviour. Subclasses must provide the logic for `CanOpenDocument` and `DoCreateDocument`. If special actions are needed to prepare for opening documents, they may be specified by overriding `AboutToOpenDocument`.

Template Method — Applicability

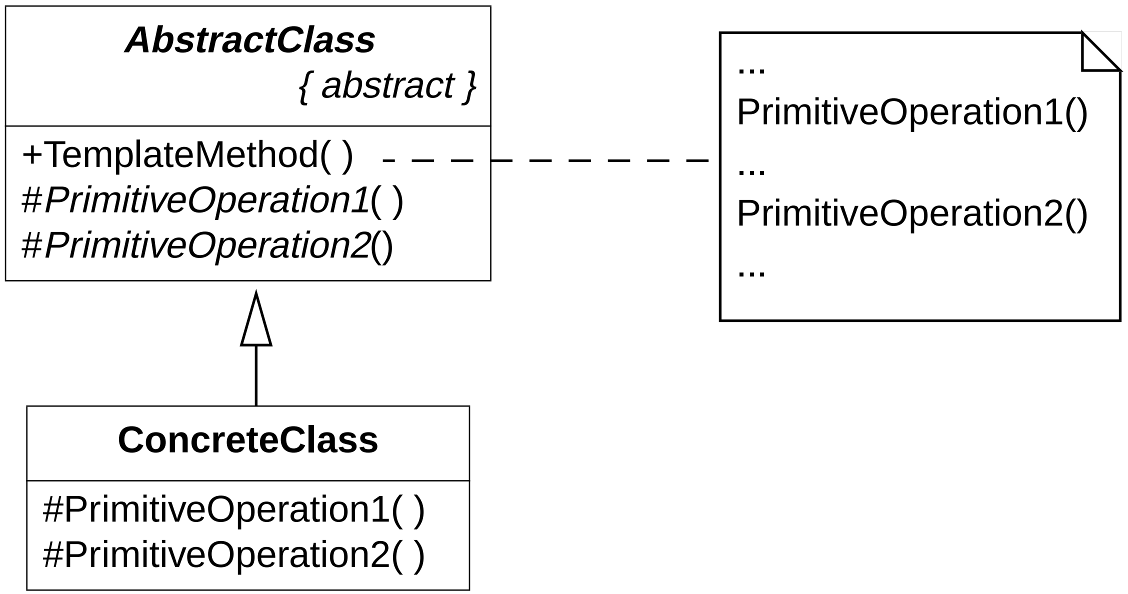
Applicability

The Template Method should be used:

- ❑ to implement the non-varying parts of an algorithm once and allow subclasses to implement the parts that may vary
- ❑ to refactor common behaviour among subclasses into a common superclass
  [This is a good example of “refactoring to generalize”.]
- ❑ to control subclass extensions. You can define a template method that calls “hook” operations at specific points, thereby permitting extensions only at those points.

Template Method — Structure

Structure



Template Method — Participants

Participants

- `AbstractClass` (e.g., `Application`)
  - declares abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  - defines a template method that implements the skeleton of an algorithm. The template method calls the primitive operations as well as operations defined in `AbstractClass` or elsewhere.
- `ConcreteClass` (e.g., `MyApplication`)
  - implements the primitives operations to carry out subclass-specific steps of the algorithm

Collaborations

- `ConcreteClass` relies on `AbstractClass` to implement the non-varying steps of the algorithm

Template Method — Consequences

Consequences

Template methods are a fundamental technique for *factoring out common behaviour* in class libraries.

They lead to an *inverted control structure* since a parent classes calls the operations of a subclass and not the other way around.

Template methods tend to call one of several kinds of operations:

- ❑ concrete operations (on client classes)
- ❑ concrete `AbstractClass` operations
- ❑ primitive operations (i.e., declared abstract in `AbstractClass`)
- ❑ factory methods (i.e., abstract operations for creating objects)
- ❑ hook operations that subclasses can extend

It’s important for template methods to specify which operations are hooks (*may* be overridden) and which are abstract operations (*must* be overridden).

Template Method — Consequences ...

A subclass can *extend* a parent class operation's behaviour by overriding the operation and calling the parent operation explicitly:

```
void DerivedClass::Operation() {
   ParentClass::Operation();
   // DerivedClass extended behaviour ...
}
```

Unfortunately it's easy to forget to call the parent operation. We can transform such an operation into a template method to give the parent control over how subclasses extend it:

```
void ParentClass::Operation() {
   // ParentClass behaviour ...
   HookOperation();
}
```

`HookOperation` does nothing in `ParentClass`:

```
void ParentClass::HookOperation() { }
```

Subclasses just override `HookOperation` to extend the behaviour of `Operation`:

```
void DerivedClass::HookOperation() {
   // derived class extension ...
}
```

Template Method — Implementation

Implementation

Three implementation issues are worth noting:

1. *Using C++ access control.* In C++, the primitive operations can be declared *protected* members. This ensures that they are *only* called by the template method. Primitive operations that *must* be overridden are declared *pure virtual*. The template method itself should not be overridden, so it can be declared non-virtual.
2. *Minimizing primitive operations.* You should minimize the number of primitive operations that a subclass must override to flesh out the algorithm of the template method. The more operations that need overriding, the more tedious things get for clients.
3. *Naming conventions.* You can identify the operations that should be overridden by adding a prefix to their names. For example, the MacApp framework for Macintosh applications prefixes primitive method names with “Do-”: e.g., “DoCreateDocument”, “DoRead”, and so on.

NB: “pure virtual” = “deferred” in Eiffel. In Eiffel all operations are “virtual”.

Template Method — Sample Code

Sample Code

This example, from NeXT's AppKit, shows how a parent class can enforce an invariant for its subclasses. The class `View` supports drawing on the screen. It enforces the invariant that its subclasses can draw into a view only after it becomes the "focus," which requires certain drawing state (for example, colours and fonts) to be set up properly.

The `Display` template method sets up this state. `View` defines two concrete operations, `SetFocus` and `ResetFocus`, that set up and clean up the drawing state, respectively. The `DoDisplay` hook operation performs the actual drawing.

```
void View::Display () {          // template method
   SetFocus();                   // set up drawing state
   DoDisplay();                  // hook operation to override in subclasses
   ResetFocus();                 // release drawing state
}
```

To maintain the invariant, the `View`'s clients always call `Display`, and `View` subclasses always override `DoDisplay`.

`DoDisplay` does nothing in `View`, and is overridden in subclasses.

Template Method — Known Uses

Known Uses

Template methods are so fundamental that they can be found in almost every abstract class. Wirfs-Brock et al. provide a good overview and discussion of template methods.

Related Patterns

Factory Methods are often called by template methods. In the Motivation example, the factory method `DoCreateDocument` is called by the template method `OpenDocument`.

Strategy: Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.

Sample Design Patterns

The following design patterns are typical of those found in *Gamma, et al.*

Creational Patterns

Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Structural Patterns

Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Behavioural Patterns

Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm with changing the algorithm's structure.

What Problems do Design Patterns Solve?

Patterns document design experience:

- Patterns enable widespread reuse of software architecture
- Patterns improve communication within and across software development teams
- Patterns explicitly capture knowledge that experienced developers already understand implicitly
- Useful patterns arise from practical experience
- Patterns help ease the transition to object-oriented technology
- Patterns facilitate training of new developers
- Patterns help to transcend “programming language-centric” viewpoints

Schmidt, CACM Oct 1995

Summary

You should know the answers to these questions:

- ❑ How can you recognize a design pattern?
- ❑ How does a design pattern differ from a piece of software?
- ❑ What is the structure of a design pattern?
- ❑ How does object composition promote flexibility?
- ❑ Why is delegation more flexible than inheritance?
- ❑ When should you use Template Method in your program design?
- ❑ How does Template Method promote software reuse?

Can you answer the following questions?

- ✎ *How would you use Template Method in an Eiffel program? Pascal?*
- ✎ *Is “Binary Search” a design pattern?*
- ✎ *“What about Window System”? “Dynamic Array”? “File Lock”?*
- ✎ *Is it a good idea to invent new design patterns?*

10. Project Management

Overview:

- ❑ Software Management
- ❑ Introducing Object-Oriented Technology
- ❑ Object Lessons

Sources:

- ❑ *Software Engineering*, I. Sommerville, Addison-Wesley, Fifth Edn., 1996.
- ❑ "Succeeding with Objects," K. Rubin, CHOOSE Conference 94 tutorial notes.
- ❑ "Transition Management Strategies," M. Lenzi, OOP 94 tutorial notes.
- ❑ "Strategies for Managing O-O Cultural Change," A. Bowles, OOP 94 tutorial notes.
- ❑ *Object Lessons*, T. Love, SIGS Books, 1993

Recommended Reading:

- ❑ *The Mythical Man-Month*, F. Brooks, Addison-Wesley, 1975
- ❑ *Succeeding with Objects: Decision Frameworks for Project Management*, A. Goldberg and K. Rubin, Addison-Wesley, 1995

Software Management

- ❑ The Software Process:
  - ☞ How is software developed?
- ❑ The Management Process:
  - ☞ How is development organized and monitored?
- ❑ Group Working:
  - ☞ How are software teams structured?
- ❑ Planning and Scheduling:
  - ☞ How are projects planned?

Software Teams

- Programming teams should not be too large (max. 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

- Chief programmer teams (see e.g. Brooks):
  - *chief programmer* is experienced & highly qualified: takes full responsibility for design, programming, testing and installation of system
  - skilled *backup programmer* (deputy) keeps track of CP’s programmer and develops test cases to verify the work
  - *librarian* manages all information associated with project
  - other experts may include: *project administrator*, *toolsmith* (produces supporting software tools), *documentation editor* (prepares doc. written by CP & BP), *language/system expert*, *tester* (develops test cases), and *support programmers* (code from detailed specs by CP)

Planning and Scheduling

- Project Milestones:
  - Milestones are reports delivered at end-points of software process activities: e.g., feasibility study $\rightarrow$ feasibility report; requirements specification $\rightarrow$ req. spec. document; ...
  - Should be scheduled roughly every 2-3 weeks

- Project Scheduling:
  - Planning and estimation are iterative and schedules must be monitored and revised during the project
  - Schedules should account for anticipated *and* unanticipated problems
  - Requirements analysis and design takes roughly twice as long as coding
  - Dependencies between project tasks must be documented (total time depends on longest path in activity graph)

Ten Golden Rules for Using Objects

- ❑ Choose a small but real project without tight timescale
- ❑ Take care with your selection of both tools and suppliers
- ❑ Invest in up-front staff training
- ❑ Establish an infrastructure to support all OO projects
- ❑ Use the mentor model for on-the-job training
- ❑ Spend longer thinking about your design than you are used to
- ❑ Prototyping is essential at all stages of the project
- ❑ Choose your programming language for practicality, not fashion
- ❑ Adopt a more democratic project team organization
- ❑ Put your strongest people in charge of your class library

Transitioning Projects

Why adopt OO Technology? How to introduce it?

Determine *goals* and *objectives*; set up a structure for *decision making* in which decisions are *traceable* back to these goals and objectives.

Set *realistic expectations* for how object-oriented technology can help you to achieve your software development goals and objectives.

Assess your *current situation* and set up process or resource improvement projects:

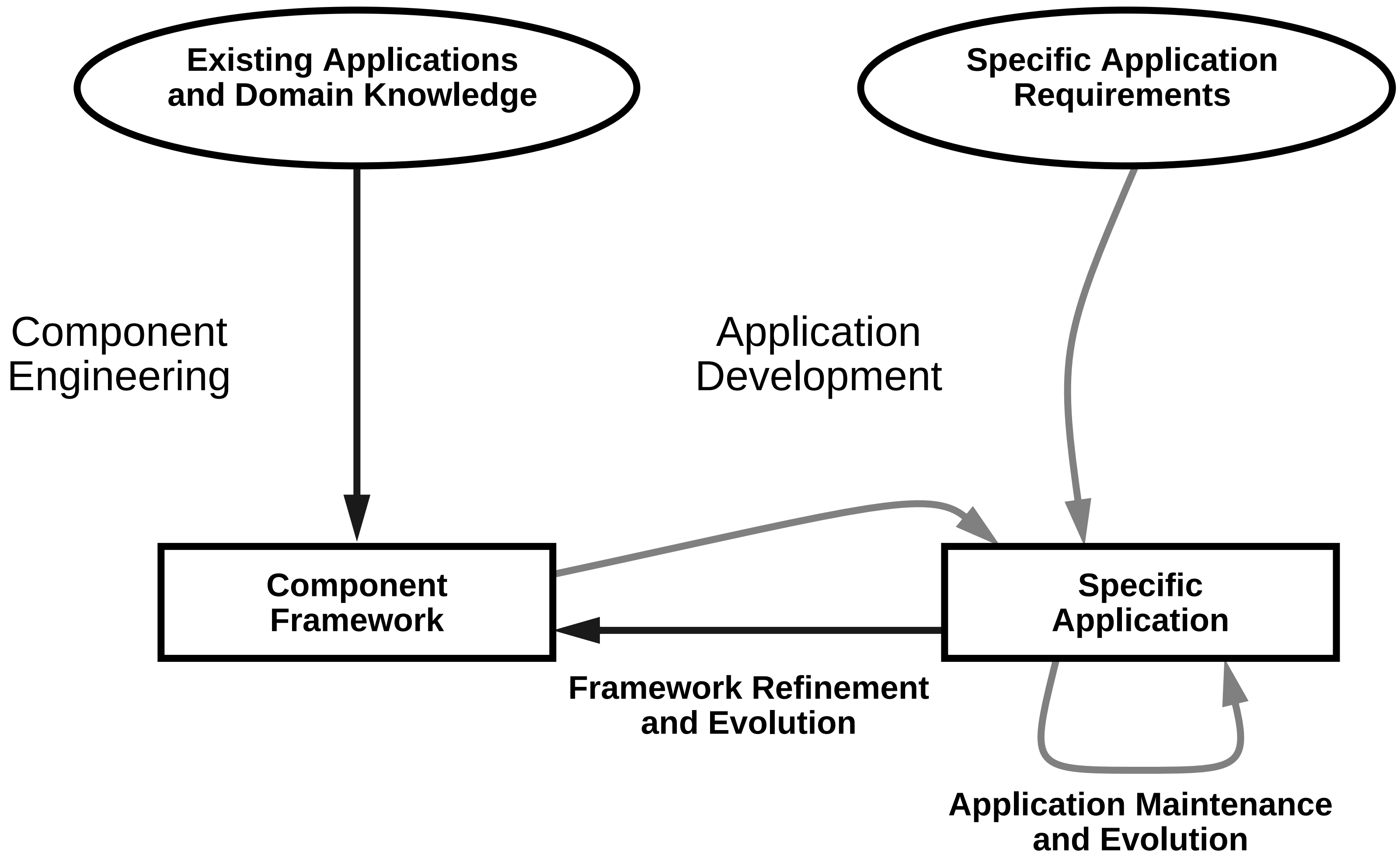
- ❑ Select product *process model*
- ❑ Set up *project plan* and control
- ❑ Select *reuse process* model
- ❑ Select *team* structure
- ❑ Select software *development environment*
- ❑ Set up *training* plan
- ❑ Set up *software measurement* program

Product Process Model

***Incremental* decision-making, development, testing and integration produce effective project results.**

- Iterative development:
  - Controlled reworking of parts of a system to remove mistakes or make improvements based on user feedback
  - "We get things wrong before we get them right"
- Incremental development:
  - Partition systems and develop at different times or rates
  - Test and integrate as each partition completes
  - Make progress in small steps to get earlier customer feedback
  - Obtain better quality testing by integrating partitions as early as possible
- Prototyping:
  - Creating a scaled-down model of some or all of the system
  - Benefit by "buying" information before making key decisions

Reuse-based Life Cycle



Project Plan and Control

Planning and execution are interleaved activities whereby partial plans are set, carried out, and the results used to do further planning.

Identify required milestones, major system capabilities, tasks and cost of each task.

Uncertainties in OOD:

- ❑ Iterative development: how many iterations?
- ❑ Incremental development: how will evaluation of completed partitions affect work on yet-to-be completed partitions?
- ❑ Prototyping: used to resolve what questions?

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

Reuse Process Model

Reusable assets are *strategic* products of the organization.

Set up a structure in which to plan and manage the process of *acquiring*, *distributing* and *maintaining* reusable assets throughout the organization.

Acquiring Reusable Assets:

- ☞ Give focus: collecting everything is not useful
- ☞ Give direction: collecting redundant solutions is not useful
- ☞ Certification: documentation, testing, history, support
- ☞ Classification: representation, classification scheme, process

Distribution and Maintenance:

- ☞ Communicate availability
- ☞ Locate, retrieve, understand and use assets
- ☞ Update reusers when assets change

Expert Services Business Model

- Technology transfer through people who understand the reusable assets
- Virtual hallway through teams whose members temporarily join application teams
- Corporate funding to emphasize importance of reuse within organization

Training Plan

- ❑ Training takes 80-200 class hours:
  - ☞ Object basics
  - ☞ Analysis and design
  - ☞ Languages
- ❑ Learning takes 6-12 months:
  - ☞ On-the-job pilot projects
  - ☞ Mentoring is highly cost-effective
  - ☞ “Mistakes” are an invaluable asset

Software Measurement Program

- Proper Program:
  - plan for evaluation/measurement
  - measures from the start
  - team size, responsibility, experience level
  - key classes + support classes
    - methods/class
    - LOC/methods (avg 5, largest 25)
    - hierarchy nesting
    - comments/method
    - coupling/cohesion
- Number of classes, methods depends on:
  - size of application
  - data or process intensive application
  - maturity of model
  - available inventory of parts

First Project

Select the right pilot project application:

- Important but not time critical
- Add value to the business
- Be apolitical
- Have definable requirements
- 4-6 month duration
- Big enough

The Pilot Project Team

Select the right pilot project team:

- ❑ 5-6 of your best people
- ❑ Look for some good abstract thinkers
- ❑ Support learning, change and teamwork
- ❑ Train the team professionally
- ❑ Provide mentoring facilities
- ❑ Allocate time for re-work (get your models right!!!)
- ❑ Don’t impose anxiety and frustration
- ❑ Need to reward:
    - ☞ reuse
    - ☞ library additions
    - ☞ low defect rate
    - ☞ *not* lines of code!!!

Staffing

Concentrate on skills, not job titles

- Business Analysts: End-user requirements, prototypes, delivering applications
- Model builders: design business frameworks
- Component builders: review/extend classes into reusable components
- System Architects: facilitate reuse
- Coaches/Mentors: facilitate object introduction and implementation

Project team sizing:

- first few pilots: < 6 staff
- first major project: <9 staff
- scope projects: < 15 staff

Costs and Risks

Biggest cost is education: technical and non-technical

- ☞ Trend away from “big-bang” training and towards “just in time” training and mentoring/internships
- ☞ Mind-set does not change overnight: on-the-job training is critical

Dangers:

- ☞ Ignoring the cost of learning: conceptual material; team sport; technology and infrastructure
- ☞ Training people at the wrong time
- ☞ Training the wrong people

Problems and Challenges

Reusability problems:

- ❑ Some evidence of reuse (25%)
- ❑ No rewards for programmers
- ❑ Lack of standards
- ❑ Incompatible languages

Gains and Costs:

- ❑ Productivity gains of 3:1, *but*
  - ☞ Higher initial training costs
  - ☞ Immature tools

User needs:

- ❑ Industry wide standards
- ❑ Improved quality in basic tools
- ❑ CASE support for OO A&D
- ❑ Reusable class libraries
- ❑ Redesign of existing management structures and practices

Challenges

- Lack of standards: interoperability, class hierarchies, languages
- Tools & methods in flux
- Usefulness/availability of third party libraries

Object Lessons

- Prototyping: plan to throw one (two?) away; prototypes are not products
- Requirements and Design: both must be formally specified and reviewed with the customer to correct misunderstandings at the earliest possible stage
- Training: 6-12 months to train software engineers to OO productivity (if ever)
- Reusability: high programmer resistance; requires incentives and support
- Productivity: can vary by 50:1; match organization to available skills & talents
- Tools: devote 20% of project staff to toolsmiths (building, acquiring ...)
- Leading vs. Managing: team leaders should read & review all code produced by the team; managers should be able to read and understand all code produced by their organization
- 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:

- ❑ Why should programming teams have no more than about 8 members?
- ❑ What is the difference between *iterative* and *incremental* development?
- ❑ What is the role of prototyping in a project?
- ❑ What is meant by “plan to throw one away”?
- ❑ Why would you put your best people in charge of the class library (instead of say, programming or design)?
- ❑ What is *mentoring* and why is it important for introducing new methods?
- ❑ Why should managers need to understand code?

Can you answer the following questions?

- ✎ *Why does requirements analysis and design take longer than implementation?*
- ✎ *What are good examples of reusable assets? (Bad examples?)*
- ✎ *What is a good example of a first project using OO technology?*
- ✎ *What are good examples of Conway’s Law in action?*

11. Computer-Aided Software Engineering

Overview:

- ❑ What is CASE?
  - ☞ CASE tool functionality vs. process support
- ❑ Tools, Workbenches and Environments
  - ☞ Programming workbenches
  - ☞ Analysis and design workbenches
  - ☞ Testing workbenches
- ❑ Software Engineering Environments

Source:

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

What is CASE?

"Computer-aided Software Engineering" refers to automated support for the software engineering process. There are mainly 3 levels of CASE technology:

1. Production-process support technology:
   - ☞ includes support for process activities such as specification, design, implementation, testing etc. (mature, and wide-spread)
2. Process management technology:
   - ☞ includes tools to support process modelling and process management (few products available)
3. Meta-CASE technology:
   - ☞ tools for generating CASE tools (not widely adopted)

CASE Tool Functionality

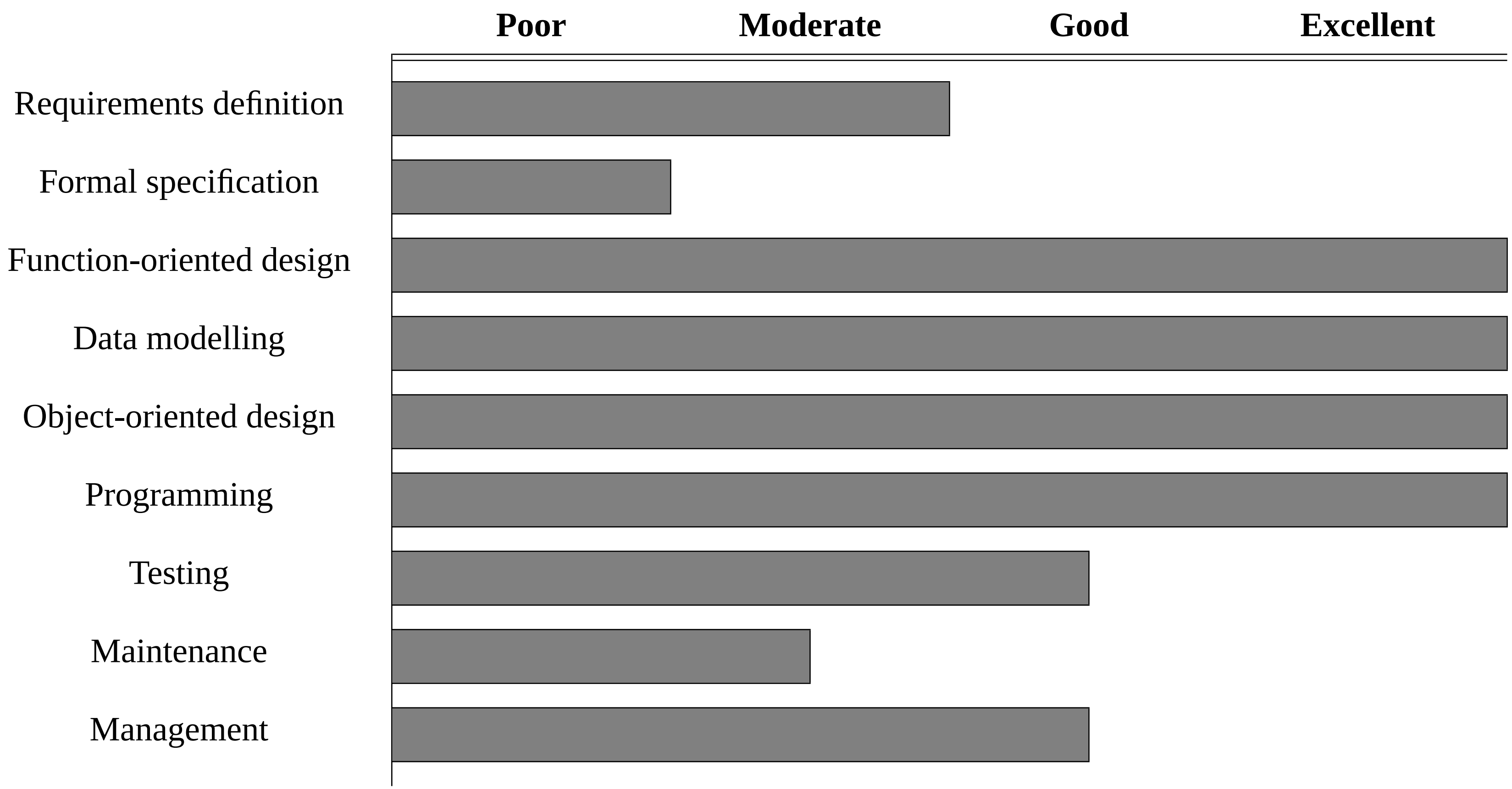
CASE tools can be classified by functionality or by their support for the software process.

Tool type	Examples
Management Tools	PERT tools, estimation tools
Editing tools	Text editors, diagram editors, word processors
Configuration management tools	Version management systems, change management systems
Prototyping tools	Very high-level languages, user interface generators
Method support tools	Design editors, data dictionaries, code generators
Language processing tools	Compilers, interpreters
Program analysis tools	Cross-reference generators, static analysers, dynamic analysers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Re-engineering tools	Cross-reference systems, program restructuring systems

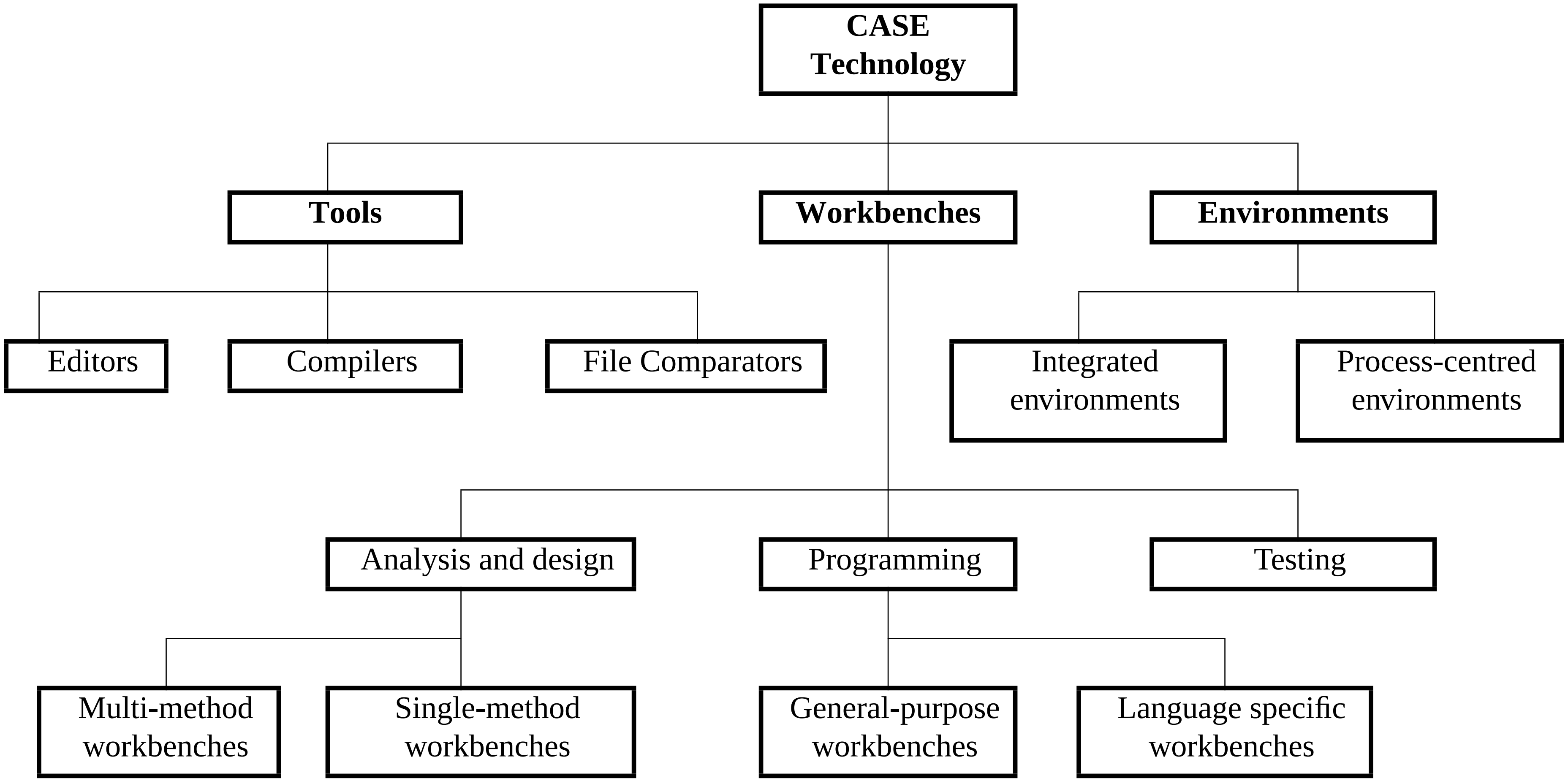
CASE Tool Process Support

Tools	Specification	Design	Implementation	Verification and Validation
Planning and Estimation	✓	✓	✓	✓
Text Editing	✓	✓	✓	✓
Document Preparation	✓	✓	✓	✓
Configuration Management	✓	✓	✓	✓
Prototyping	✓			✓
Diagram Editing	✓	✓		
Data Dictionary	✓	✓		
User Interface Management		✓	✓	
Method Support	✓	✓		
Language Processing			✓	✓
Program Analysis			✓	✓
Interactive Debugging			✓	✓
Program Transformation			✓	
Modelling and Simulation	✓			✓
Test Data Generation				✓

Quality of Tools Support



Tools, Workbenches and Environments



Integrated CASE

CASE systems can be integrated at various levels:(Wasserman 1990):

1. Platform integration
   ☞ Tools run on the same hardware/operating system platform

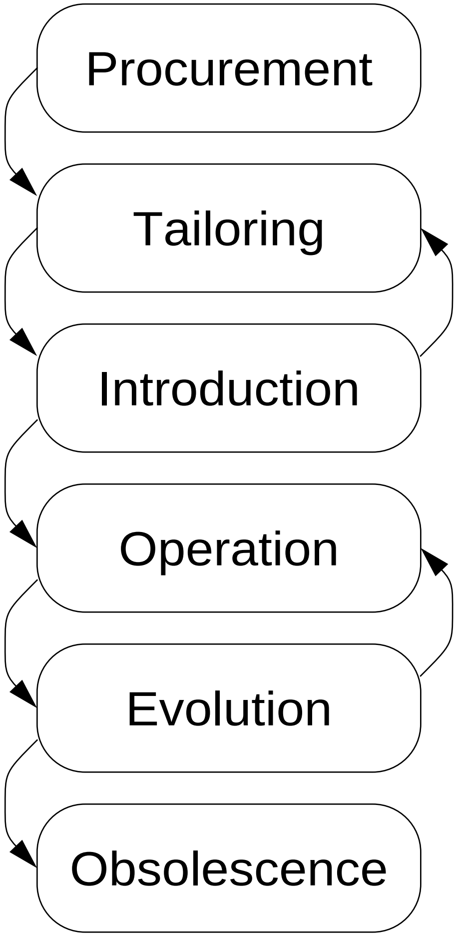
2. Data integration
   ☞ Tools operate using a shared data model

3. Presentation integration
   ☞ Tools offer a common user interface

4. Control integration
   ☞ Tools may activate and control the operation of other tools

5. Process integration
   ☞ Tool usage is guided by an explicit process model and process engine

The CASE life cycle



During CASE system *procurement*, current methods and standards, platform, application domain, security, and CASE system cost (including training and maintenance) must be considered.

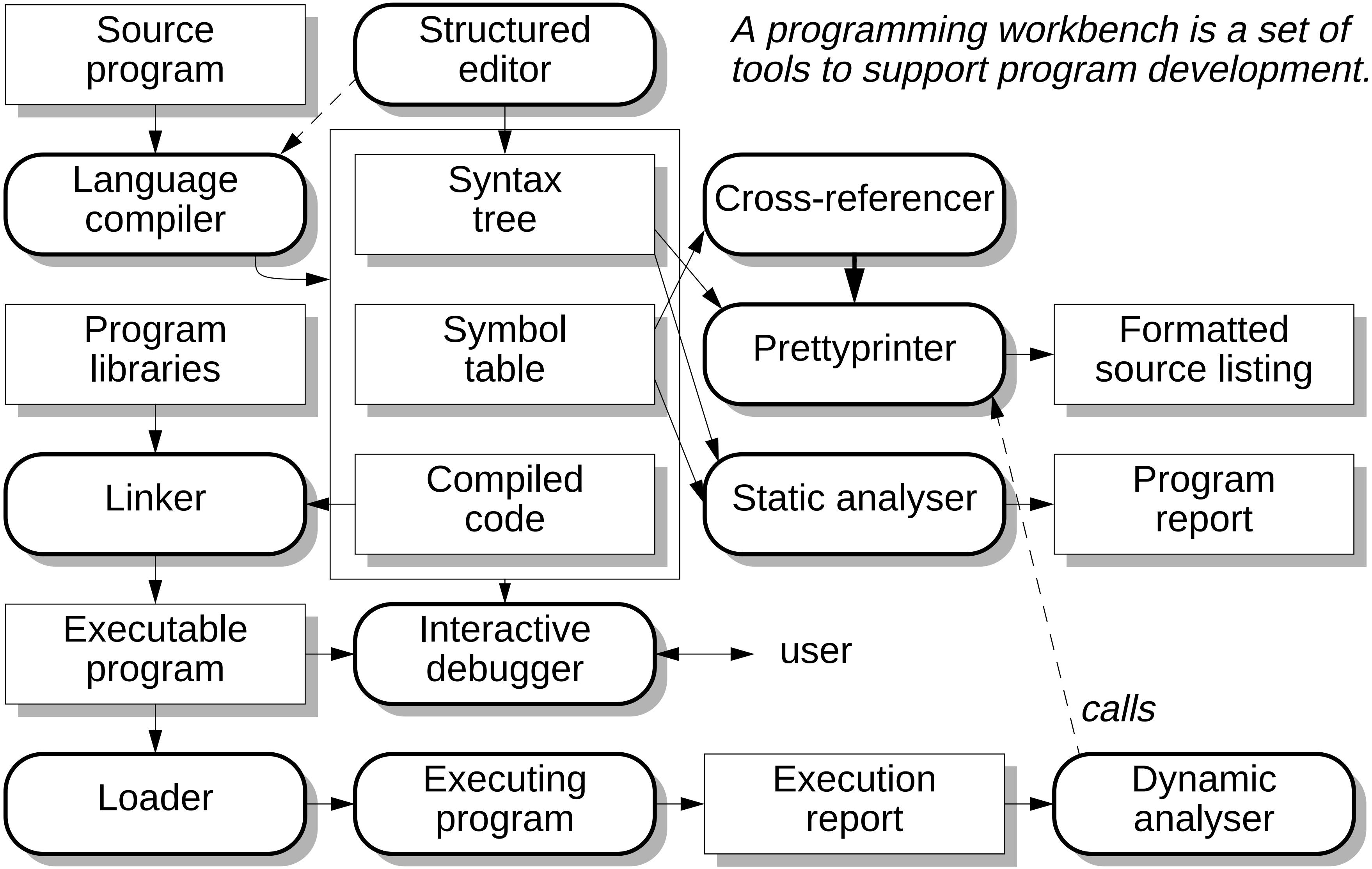
CASE system *tailoring* involves installation, process model definition, tool integration, and documentation of the installation.

Introduction can be risky due to user resistance (CASE systems restrict freedom by imposing discipline), inadequate training, or even management resistance (changing tools and procedures increases risks for individual projects).

An *obsolete* CASE system cannot simply be scrapped, but must be phased out over a transition period.

Programming Workbenches

A programming workbench is a set of tools to support program development.



Static Program Analysers

Static program analysers scan the source code to detect possible faults and anomalies:

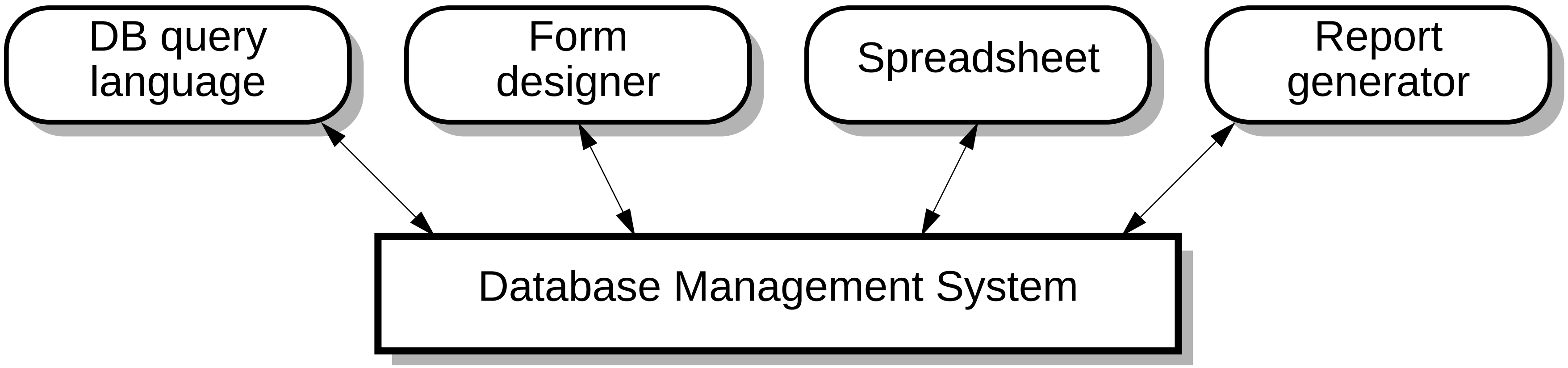
- ☞ Unreachable code
- ☞ Unconditional branches into loops
- ☞ Undeclared variables
- ☞ Variables used before initialization
- ☞ Variables declared and never used
- ☞ Variables written twice with no intervening assignment
- ☞ Parameter type mismatches
- ☞ Parameter number mismatches
- ☞ Uncalled functions and procedures
- ☞ Non-usage of function results
- ☞ Possible array bound violations
- ☞ Misuse of pointers

Stages of Static Analysis

1. Control flow analysis:
   - ☞ loops with multiple exit or entry points and unreachable code ...
2. Data use analysis:
   - ☞ use of uninitialized variables, declared but unused variables ...
3. Interface analysis:
   - ☞ consistency of procedure declarations and use, unused functions ...
4. Information flow analysis:
   - ☞ identifies dependencies of output variables on input
5. Path analysis:
   - ☞ identifies all possible paths through program

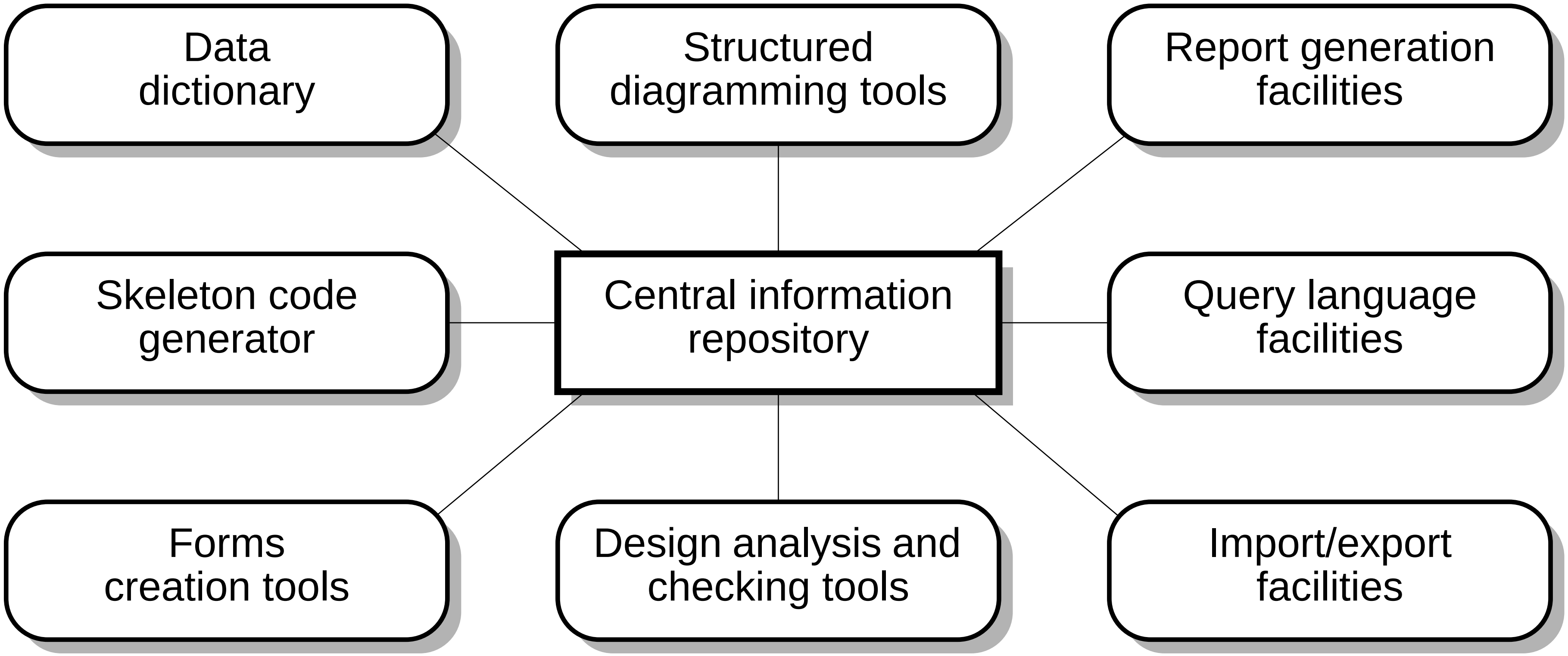
4GL Workbenches

A so-called “Fourth Generation Language” (4GL) is really a programming workbench for producing interactive applications that provide users with form and spreadsheet views on an underlying (relational) database.



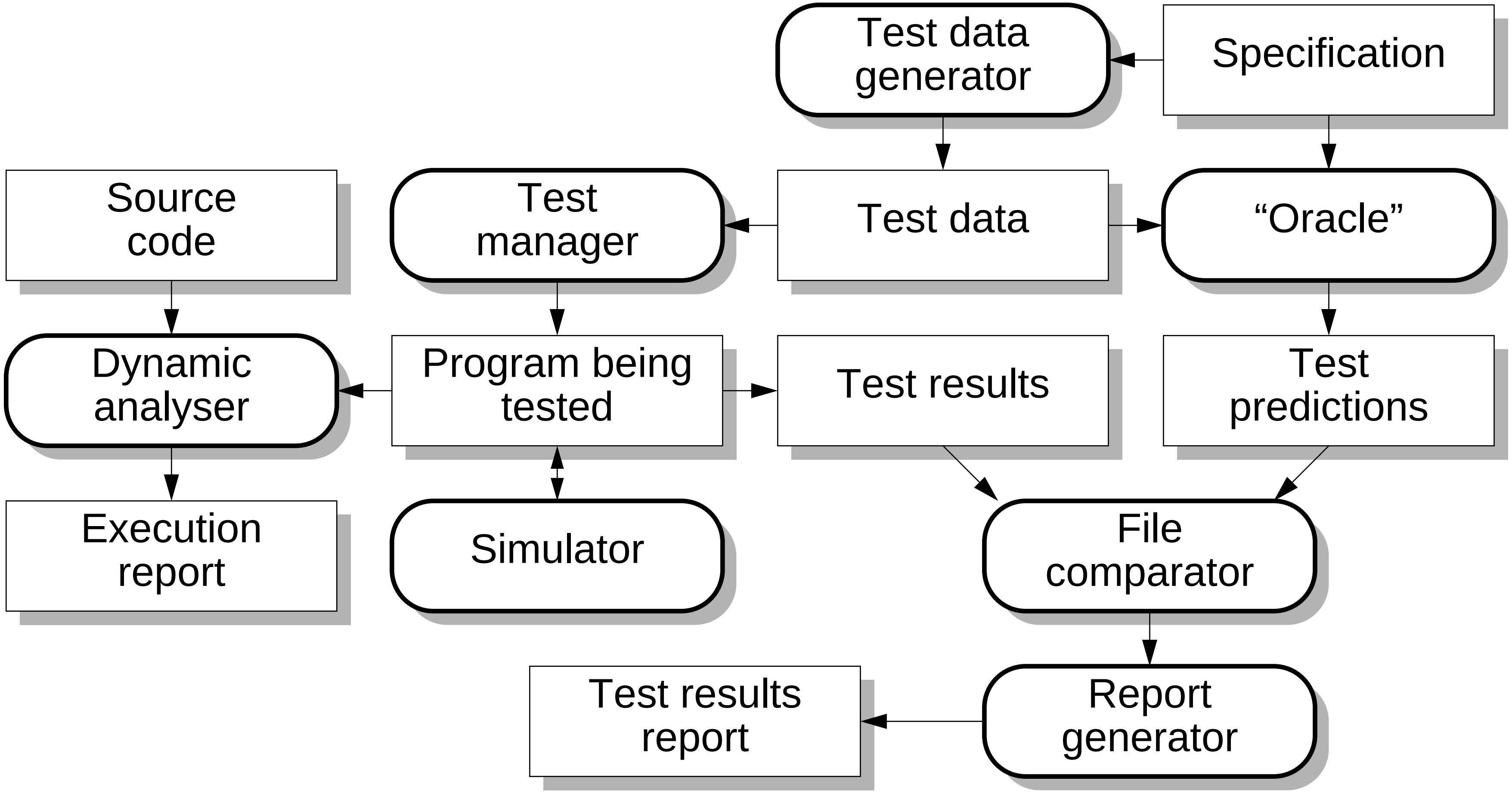
Analysis and Design Workbenches

Analysis and design workbenches support the *modelling* phases of the software process, usually by means of a graphical notation (e.g., dataflow, ER, UML etc.), and may or may not support a specific analysis and design method (e.g., JSD, Booch, etc.).



Testing Workbenches

Testing tends to be application and organization specific, so workbenches are typically developed in-house using standard tools.



Testing Tools

Test Data Generators:

- ☞ automatic generation of test inputs
- ☞ output analysis by “oracle” (i.e., prototype, parallel system, human)

File Comparators:

- ☞ automatically comparing old and new test results (e.g., UNIX “diff”)

Simulators:

- ☞ hardware — cost, availability, risk ...
- ☞ events — real-time, reproducibility, load ...

Dynamic Analysers:

- ☞ instrumentation statements are automatically added to program
- ☞ execution profiles are generated and analysed

Configuration Management Tools

Configuration management is concerned with the development of procedures and standards for managing an evolving software system product.

Tool examples:

Version Control — SCCS and RCS:

- ☞ check-out and check-in of components
- ☞ logging changes (who, where, when)
- ☞ changes converted to system “deltas” (can generate any version)
- ☞ “freezing” of versions as releases (possibly parallel ⇒ tree of versions)

System Building — Make:

- ☞ dependency specification
- ☞ rules for generation of intermediate files
- ☞ automatic re-generation of out-of-date files

Software Engineering Environments

A software engineering environment (SEE) is a set of hardware and software tools which can act in combination in an integrated way to provide support for the whole of the software process from initial specification through to testing and system delivery.

— Sommerville, 5th edn., p. 548

SEEs vs. CASEs:

- ❑ SEEs are fully integrated (all 5 levels)
- ❑ SEEs support development by teams and provide integrated configuration management
- ❑ SEEs support workbenches for a range of software development activities

Although there are presently no good examples of SEEs, the Portable Common Tool Environment (PCTE) has been widely adopted as a standard framework for SEEs ...

Summary

You should know the answers to these questions:

- ❑ What are the key features of a CASE environment?
- ❑ Which phases of the software lifecycle benefit from configuration management?
- ❑ In what different ways can CASE system be integrated?
- ❑ What are the risks in adopting a CASE system?
- ❑ What kinds of errors can be detected by static analysis?
- ❑ What is an “oracle” and how is it used?

Can you answer the following questions?

- ✎ *Why is the quality of tool support for project management not as good as for design and programming?*
- ✎ *Where does SNiFF+ fit into the CASE system classification?*
- ✎ *Is it better to use a single method A&D workbench or a multi-method one?*
- ✎ *Why is Meta-CASE technology not widely used?*
- ✎ *Why are there no good examples of SEEs in use?*

12. 4th Generation Systems — Delphi

Invited Lecture

Markus Lumpe