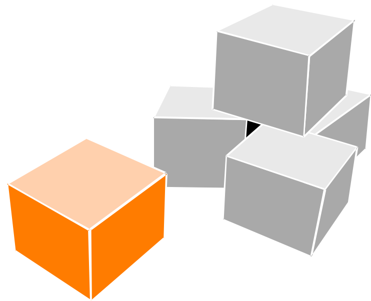


Software
Technology
Group
TU Darmstadt | FB Informatik

Software Engineering Design
3. Principles of Good Design

Prof. Dr. Mira Mezini
Dipl.-Inform. Christoph Bockisch
Dipl.-Ing. Michael Haupt

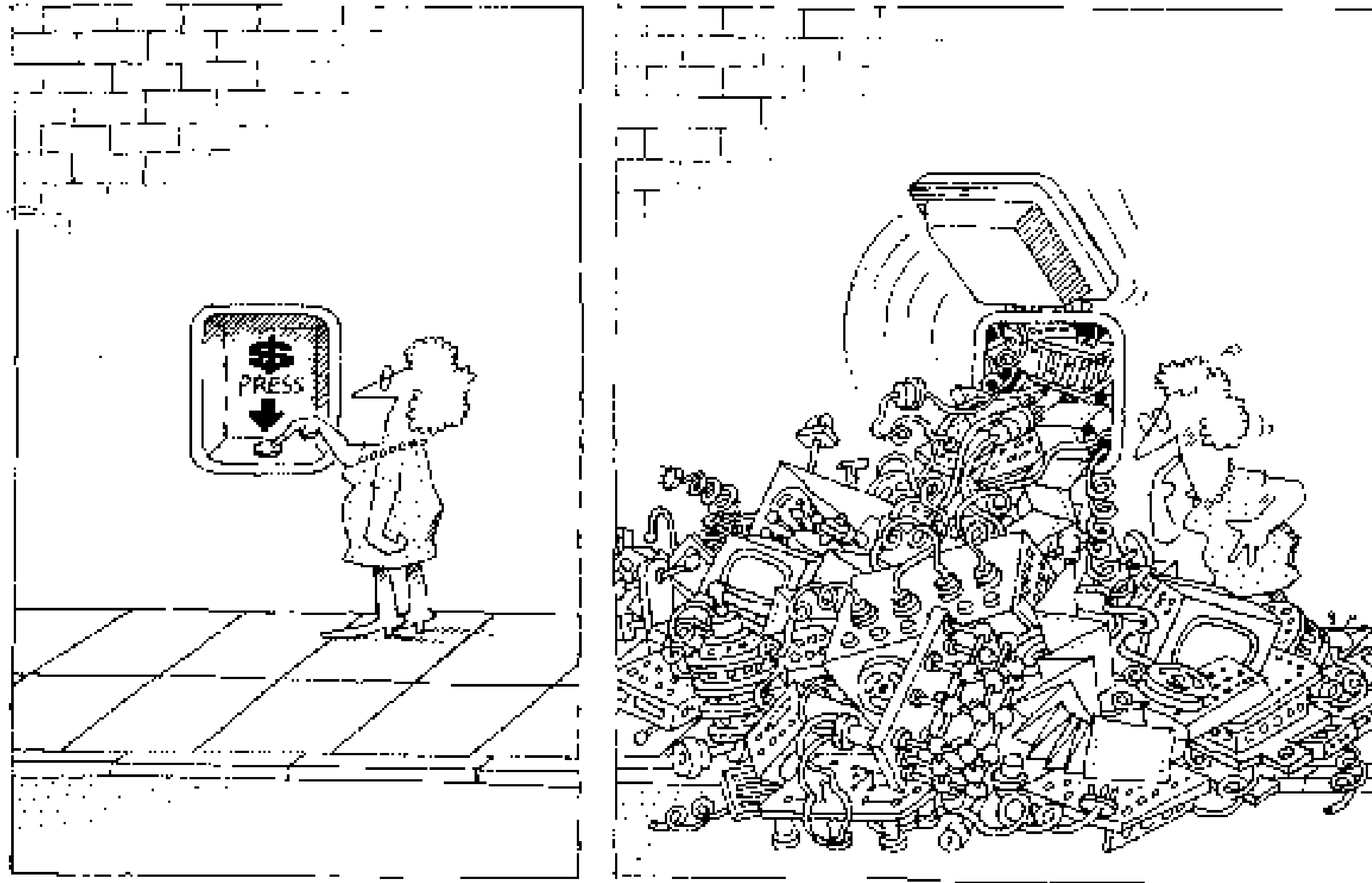
Agenda

- software quality criteria
- modularity
  - how can modularisation be applied?
  - 5 criteria, 5 rules, 5 principles
- modularity and performance
- object-orientation enabling modularity

What is Good Software?

- external criteria: correctness, robustness, extendibility, reusability, efficiency, ease of use ...
  - perceivable to:
    - end user, e.g., a travel agent using a flight reservation system
    - persons who purchase the software, e.g., an airline executive acquiring or commissioning flight reservation systems
- internal criteria: modularity, readability, ...
  - perceptible to IT professionals that have access to the software text
- in the end, only external factors matter, but the key to achieving them is in the internal ones

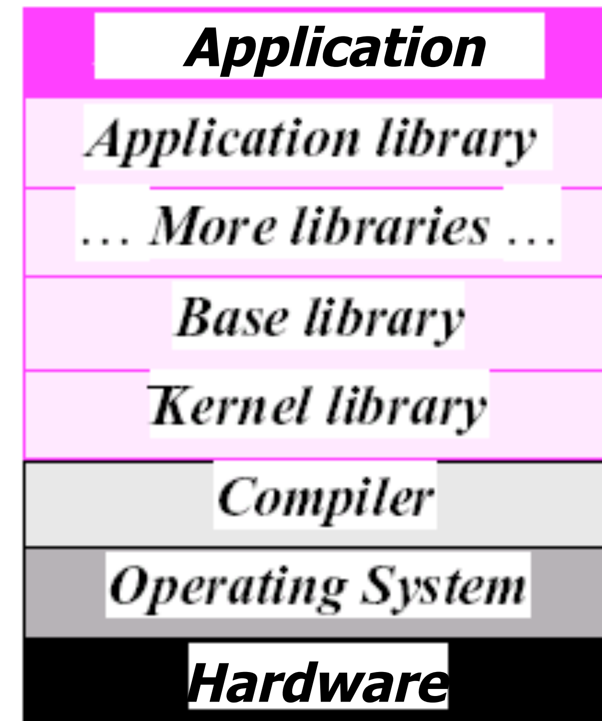
External vs. Internal Criteria



The task of the software development team is to engineer the illusion of simplicity.

External Criteria / Correctness

- correctness is the ability of software products to perform their tasks, as defined by their specification
- due to complexity, methods for ensuring correctness will usually be conditional or layered, each layer relying on lower ones



External Criteria / Robustness

- robustness is the ability of software to react appropriately to abnormal conditions

- correctness addresses the behaviour of a system in cases covered by its specification; robustness characterises what happens outside of that specification

External Criteria / Efficiency

- efficiency is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied, bandwidth used in communication devices

- a *software case of split personality*
  – Mr. Microsecond: developers often show an exaggerated concern for micro-optimisation
  – Dr. Abstract: “make it right before you make it fast” ... “next year’s computer model is going to be 50% faster anyway”

External Criteria / Efficiency

- efficiency is only one of the factors of quality; we should not let it rule our engineering lives

- but it is a factor, and must be taken into consideration, whether in the construction of a software system or in the design of a programming language

If you dismiss performance, performance will dismiss you!

External Criteria / Efficiency

- this course focuses on concepts of object-oriented software design, not on implementation issues → hence, we don't deal explicitly with performance techniques
- but the concern for efficiency will be there throughout: an important message is that modularity does help in achieving good performance

External Criteria / Extensibility

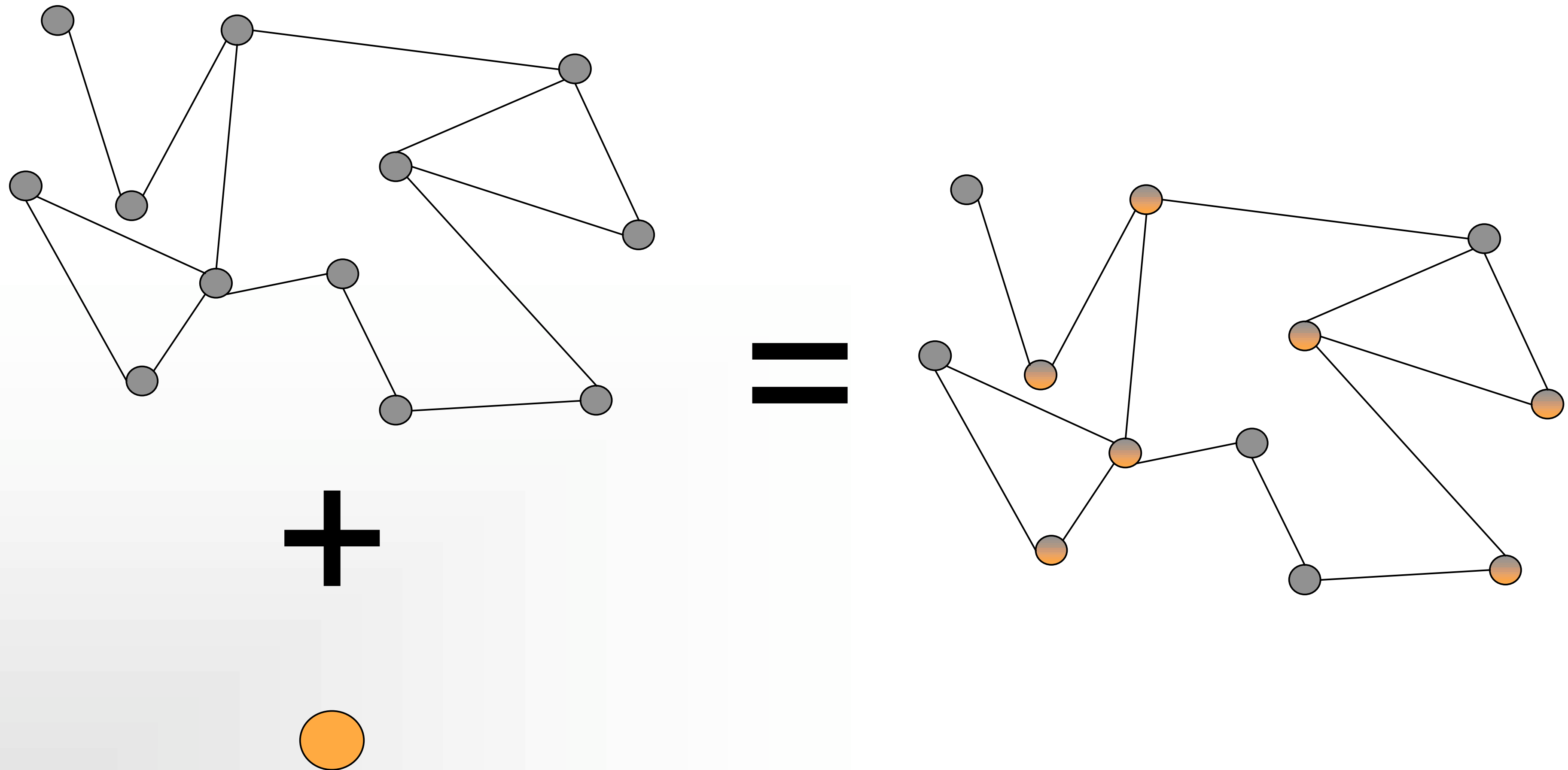
- extensibility characterises the ease of adapting software to changes of specification
- a problem of scale: as software grows bigger, it becomes harder to adapt
  - *a large software system resembles a giant house of cards in which pulling out any one element might cause the whole edifice to collapse*
- change is pervasive in software development
  - even in scientific computation, where we may expect the laws of physics to stay in place, our way of understanding and modelling physical systems will change.

External Criteria / Extensibility

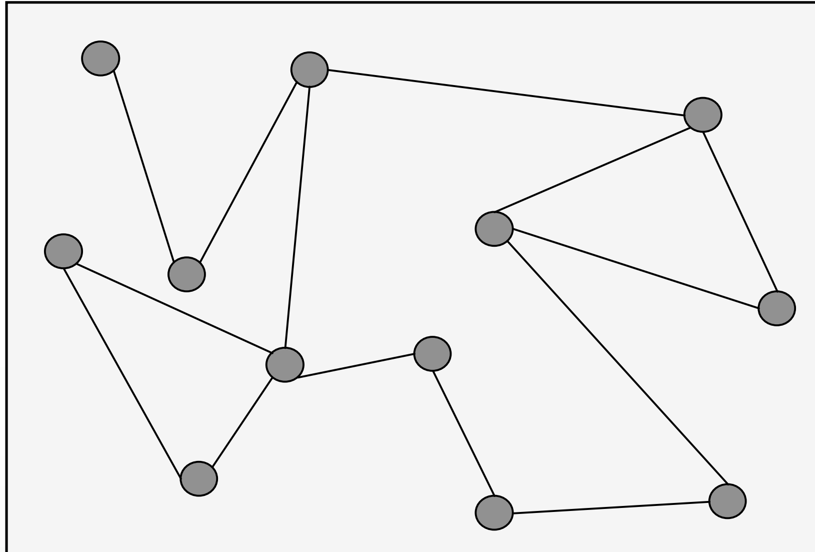
two principles: design simplicity & decentralised architectures

> ***You know you have achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away.***
>
> **Antoine de Saint-Exupéry**

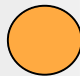
Bad Extensibility



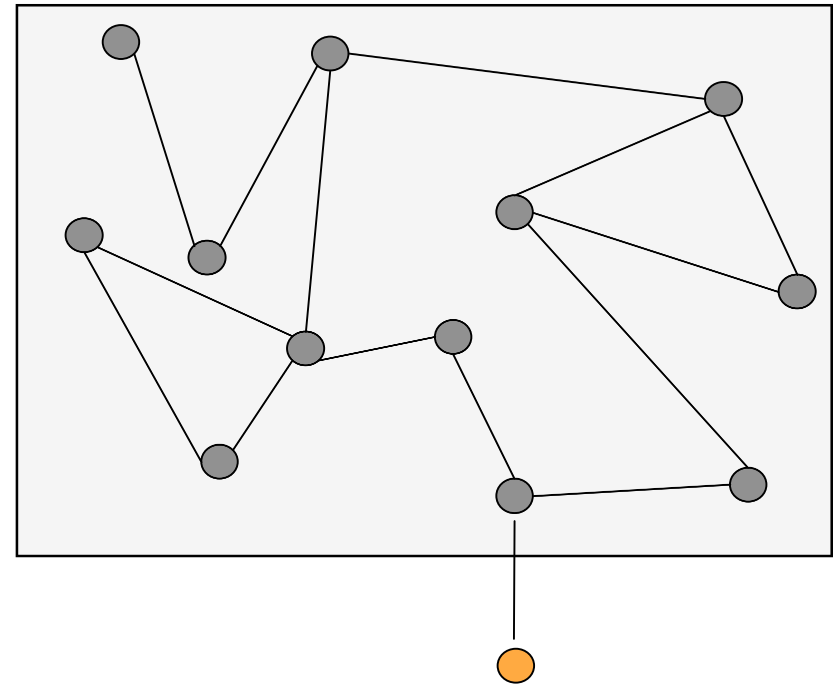
Good Extensibility



+



=



External Criteria / Reusability

- reusability is the ability of software elements to serve for the construction of many different applications

- software systems often follow similar patterns:
  - it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before
  - by capturing such a pattern, a reusable software element will be applicable to many different developments

Ext. Crit. / Extensibility & Reusability

- more important in our context: extensibility & reusability
  - remove inadequacies,
  - adapt to changing needs,
  - extend functionality quickly
  - integrate with other systems
  - use components in other projects

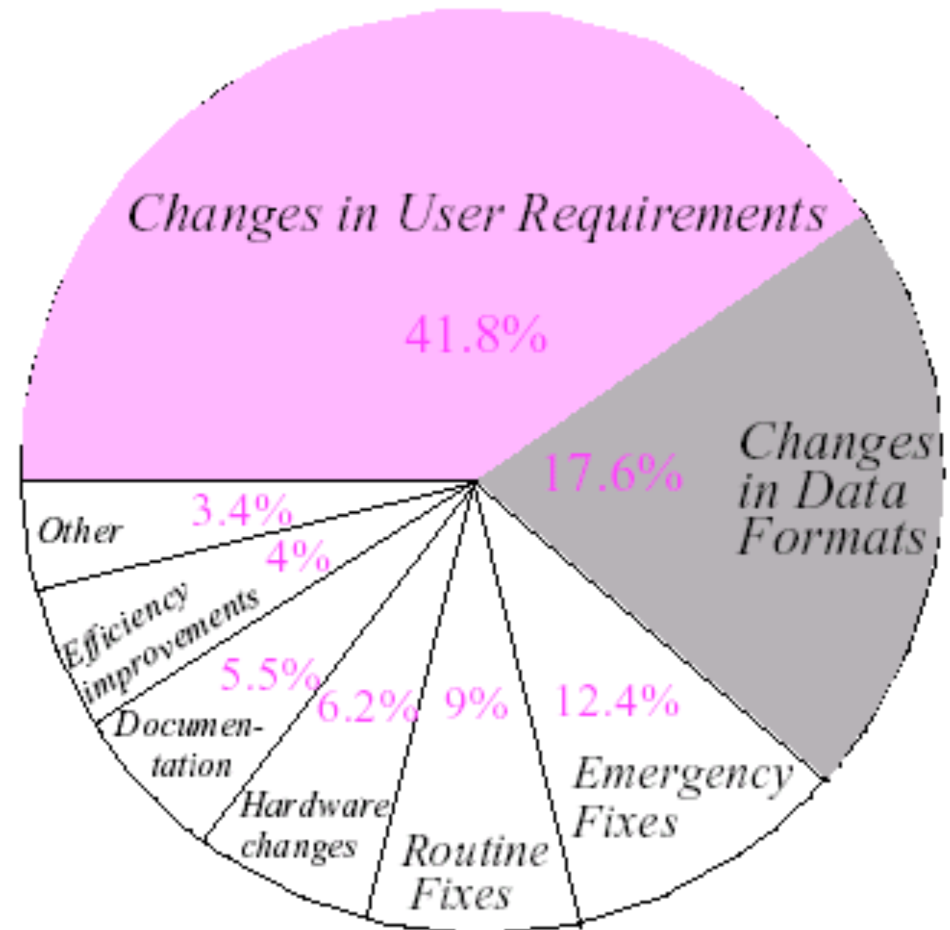
Software spends 60%-80% of its lifetime in maintenance!

Maintenance

2/5 of the cost devoted to user-requested extensions and modifications

"noble" and inevitable part of maintenance

How much of the effort could the industry spare if software was built from the start with more concern for extensibility?

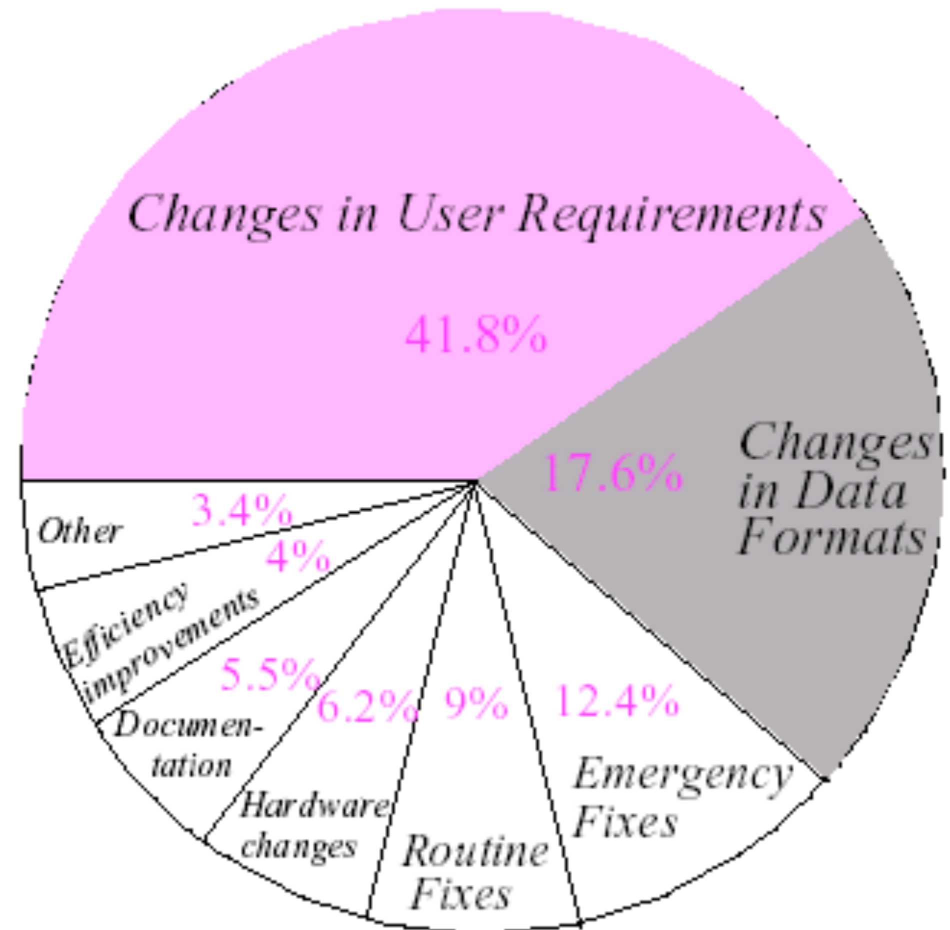


Lientz, Swanson `80 survey of 487 projects

Maintenance

Changes in data formats: The second item in order of percentage cost.

The issue: knowledge on data formats spread out over many parts of the system, causing large program changes if some of the physical structure changes — as it inevitably will.



Lientz, Swanson `80 survey of 487 projects

Ext. Crit. / Extensibility & Reusability

- similar ideas are useful for improving both qualities:
  - e.g., any idea that helps produce more decentralised architectures, in which the components are self-contained and only communicate through restricted and clearly defined channels

- the term modularity will cover extensibility and reusability

Ext. Crit. / Extensibility & Reusability

Mission: Modular software architecture

When quality is pursued, productivity will follow!

“there is never time to do it right, but there is always time to do it over”

Ext. Crit. / Extensibility & Reusability

Mission: Modular software architecture

often de-facto software architecture

Ext. Crit. / Extensibility & Reusability

Mission: Modular software architecture

if you're not able to solve the problem, it might help to put it under a clean cover...

Ext. Crit. / Extensibility & Reusability

Mission: Modular software architecture

Sometimes, this is the only alternative...

Ext. Crit. / Extensibility & Reusability

Mission: Modular software architecture

Best: **incremental growth** …

"MIR" was developed with the goal to support maintenance and growth

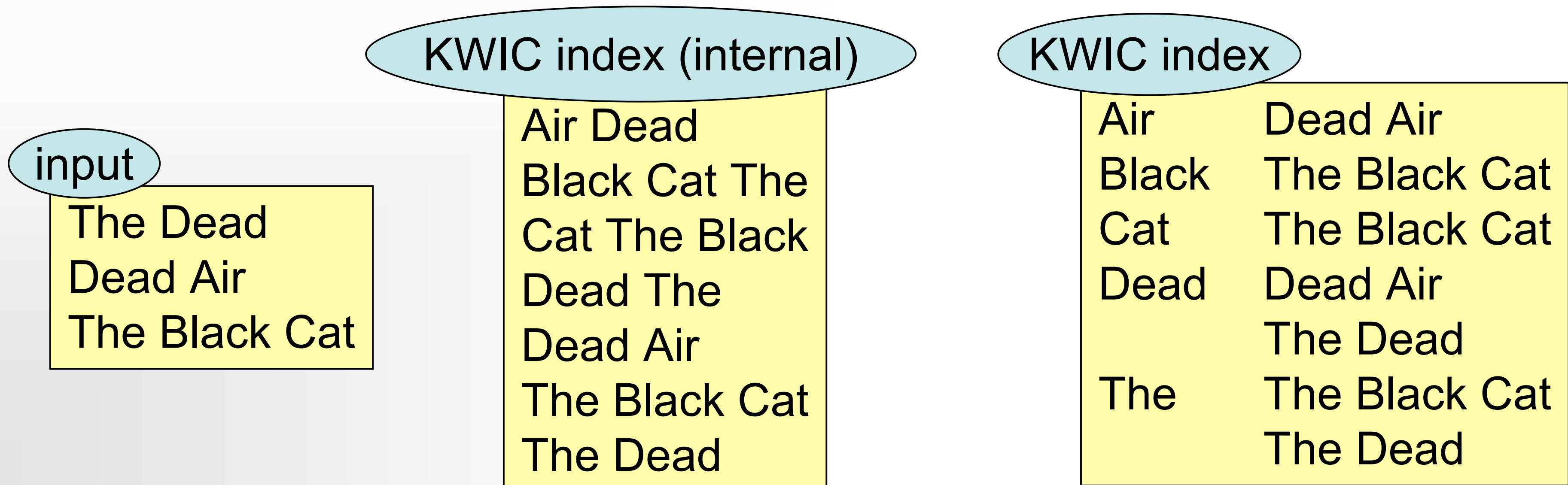
How is Modularity Achieved?

- by decomposing a system into modules
  - what *is* a module?
  - what principles are applied during decomposition?

- some characteristics of modularised systems
  - separate, distinct modules for each task
  - well-defined inputs and outputs
  - tested independently
  - maintenance in modular fashion

Example: KWIC Index

- KWIC: KeyWord In Context
  - used to find, e. g., article or book titles quickly even if only one or two keywords from the title are known
- building a KWIC index from a collection of lines
  - each line contains a title to be indexed
  - form circular shifts of all lines and order them alphabetically



KWIC Indexer: Decomposition 1

input
read lines,
store in memory

circular shift
create all shifts,
store in memory

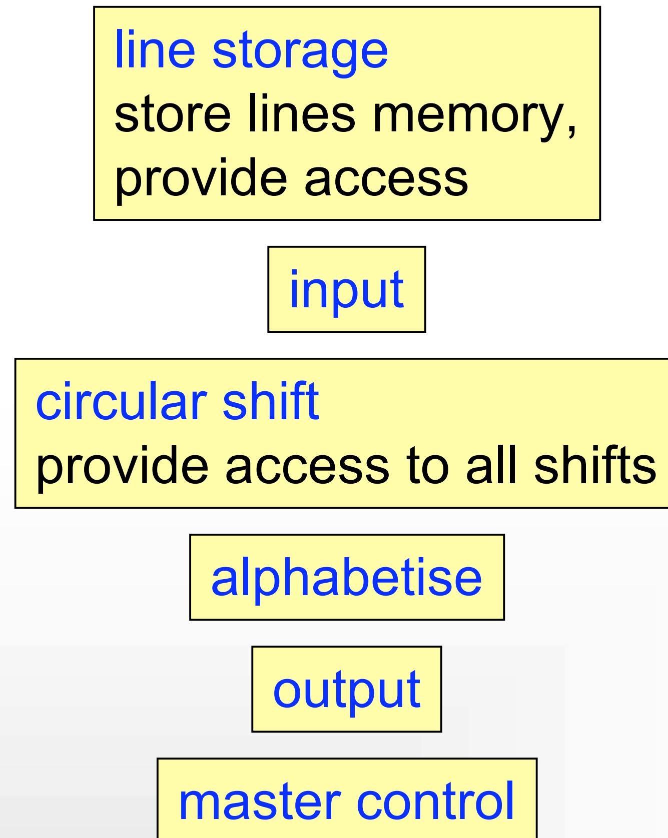
alphabetise
sort the shifts alphabetically,
according to order of lines

output
print the formatted index

master control
control the sequencing of the four processing modules

- modules
  - well-defined interfaces
  - small enough to be understood
  - modules use different shared data representations
- functional abstraction
  - decompose a system into its functional blocks
  - let them operate on shared data
- reaction to evolution
  - change in data representation affects all modules
  - indexing shifts right away affects shifter, alphabetiser and output

KWIC Indexer: Decomposition 2



- modules encapsulate data
  - no more sharing
  - communication via interfaces
- data abstraction
  - provide interface for accessing data
  - hide implementation details of data structures and algorithms
- reaction to evolution
  - change in data representation only affects line storage
  - indexing shifts right away only affects shifter

More Evolution

- typically, it is functionality that evolves, not data
  - functionality changes often affect module interfaces
  - existing functionality may not be sufficient to fulfil new needs
- new KWIC indexer features that might be added:
  - omit shifts starting with "noise" words
  - include only shifts starting with certain words
- data abstraction suffers
  - shifter needs to be modified in either case
  - when both changes apply, the shifter gets more complicated

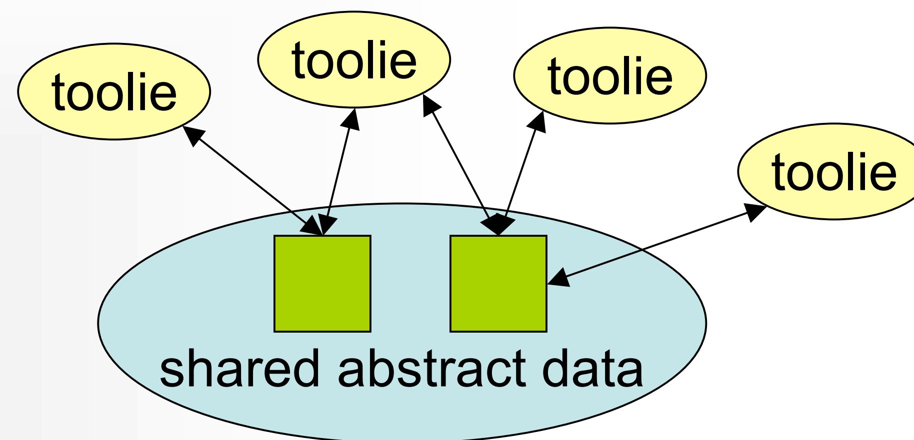
Requirements for Evolution

- requirements
  - incremental enhancements
  - independent development of modifications
  - even if changes cannot be achieved using data abstraction

- examples
  - spreadsheets: use values in data cells to interact with pre-existing equations
  - active data in object-oriented systems: Smalltalk *dependent objects*, JavaBeans *listeners*

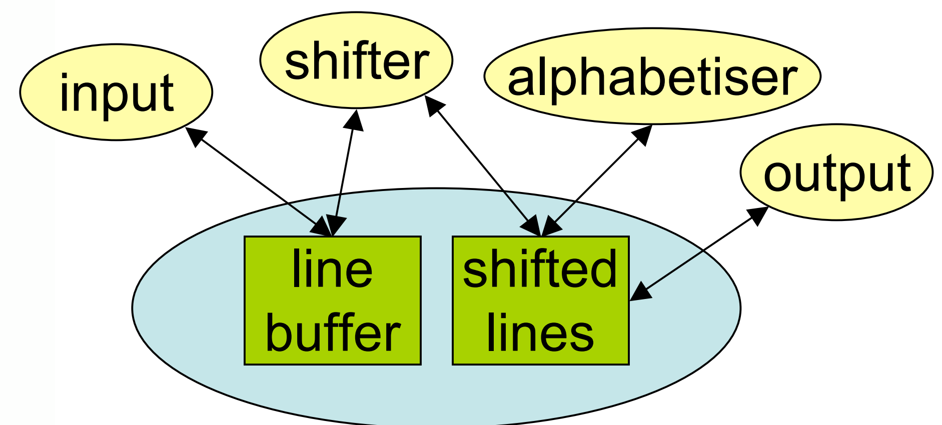
Tool Abstraction

- abstraction technique complementing data abstraction
- the system has a common structure encouraging incremental change
  - pool of shared abstract data
  - collection of "toolies" operating on data
  - when data is updated by a toolie, others may have to be notified



KWIC Indexer and Tool Abstraction

- input, shifted and alphabetised lines are shared abstract data
- toolies
  - input creates a new shared buffer and populates it with read lines
  - shifter creates another shared buffer; acts when the input toolie's insert operation terminates → each line is shifted after insertion
  - alphabetiser is triggered by the completion of shifter activities and sorts (e. g., incrementally)
  - output
- evolution-friendly modularisation
  - introduce dedicated omit and include toolies
  - associate with insert operation of shifter toolie



Modularity (According to Meyer)

- a single definition of modularity would be insufficient; as with software quality, we must look at modularity from more than one point of view

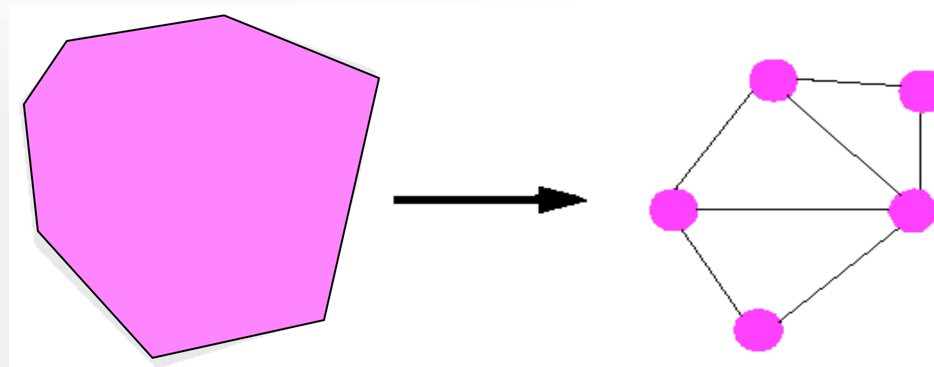
- requirements on a modular design method defined by means of complementary properties:
  - five criteria, five rules, and five principles
  - criteria are independently aimed at characterising modular design methods
  - rules follow from criteria and principles from criteria and rules

Modularity / Five Criteria

- a modular design method should satisfy five fundamental requirements:
  - Decomposability
  - Composability
  - Understandability
  - Continuity
  - Protection

Modular Decomposability Criterion

A software construction method satisfies modular decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them.



Divide to conquer ...

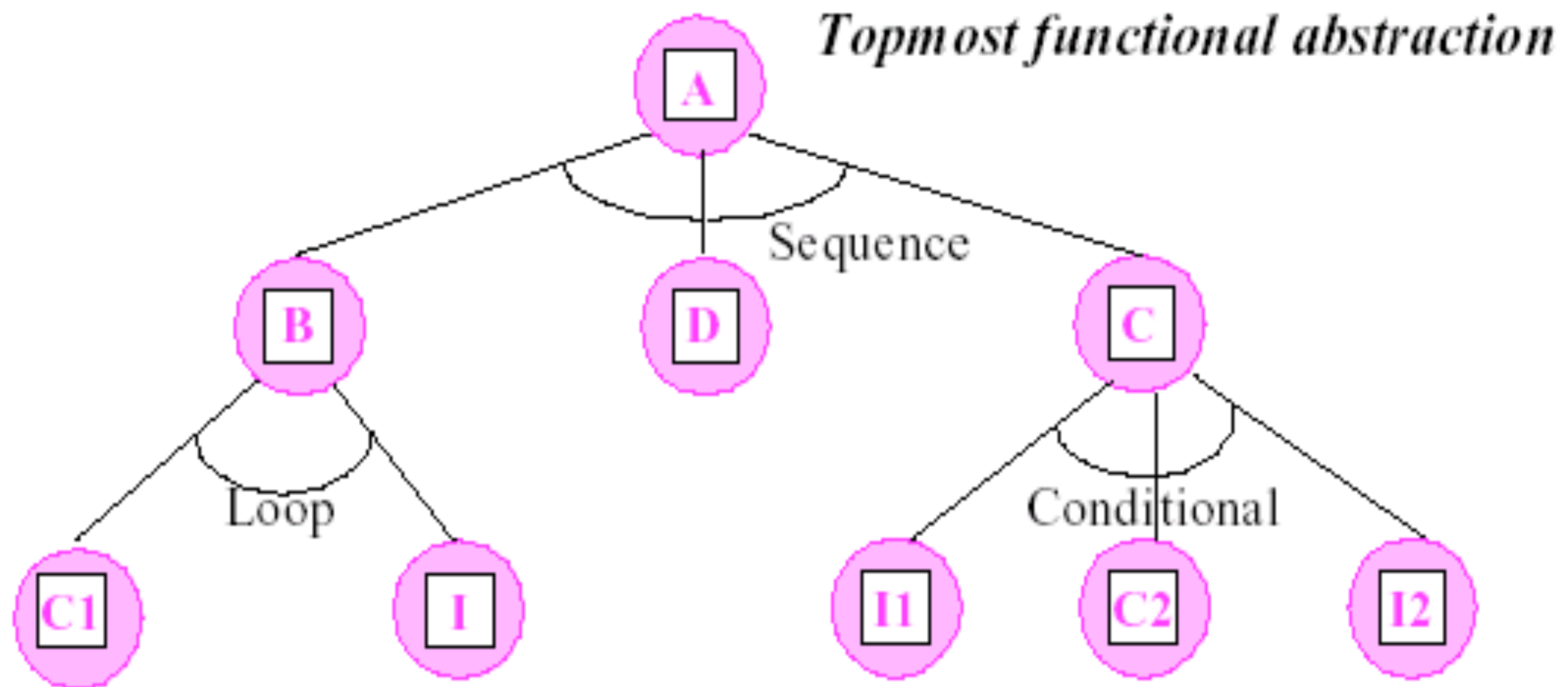
Modular Decomposability Criterion

- a corollary of modular decomposability is division of labor (separation of concerns): once you decompose a system into subsystems you should be able to distribute work on these subsystems among different groups

- difficult goal since it limits dependencies that might exist:
  - keep dependencies between subsytems to the bare minimum
  - dependencies must be known

Modular Decomposability Criterion

- most obvious *example* of a method meant to satisfy the decomposability criteria is *top-down design*

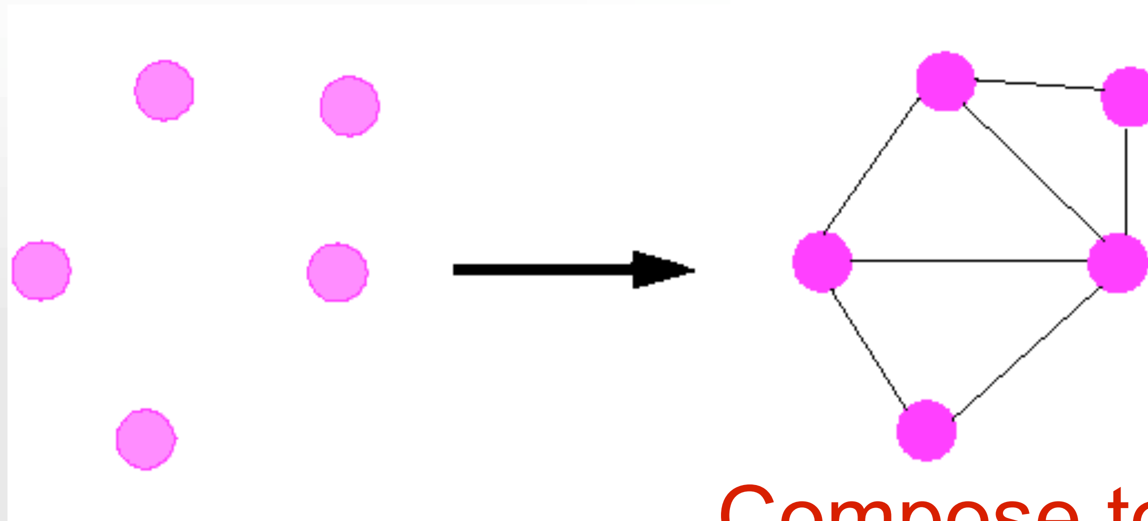


Modular Decomposability Criterion

- counter-example: any method that encourages developers to include, in each software system, a global initialisation module
  - good “temporal cohesion”: all actions of the initialisation module are executed at the same stage of the system’s execution
  - endanger the autonomy of modules: the initialisation module authorisation accesses many separate data structures, belonging to the various modules of the system and requiring specific initialisation actions

Modular Composability Criterion

A design method satisfies Modular Composability if it favours the production of software elements which may be freely combined with each other to produce new systems, possibly in environments quite different from the one in which they were initially developed.



Compose to rule ...

Modular Composability Criterion

- the composability criterion yields software elements that will be sufficiently autonomous — sufficiently independent from the immediate goal that led to their existence

- directly connected with the goal of reusability: the aim is to find ways to design software elements performing well-defined tasks and usable in widely different contexts

Modular Composability Criterion

example: Unix shell conventions

- Unix commands operate on sequential character streams as input, and produce an output with the same standard structure
- this makes them composable through the | (pipe) operator: A | B represents a program which will take A's input, have A process it, send the output to B as input, and have it processed by B

Modular Composability Criterion

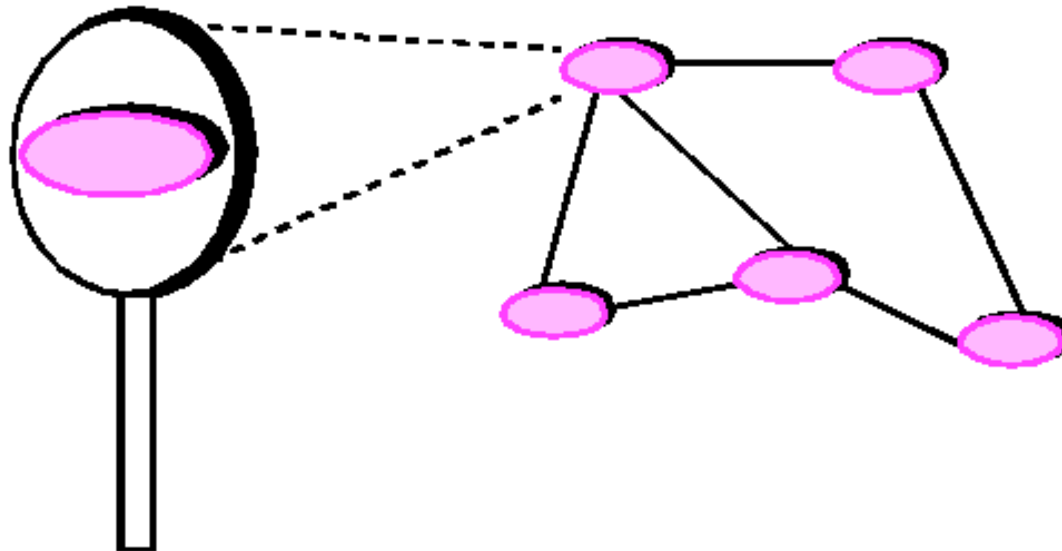
counter-example: preprocessors

- preprocessors are a popular way to extend programming languages
  - accept an extended syntax as input and map it into the standard form of the language
  - typical preprocessors for Fortran and C support graphical primitives, extended control structures or database operations

- usually such extensions are not compatible; then you cannot combine two of the preprocessors

Modular Understandability Criterion

A method favours Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.



Modular Understandability Criterion

counter-example: sequential dependencies

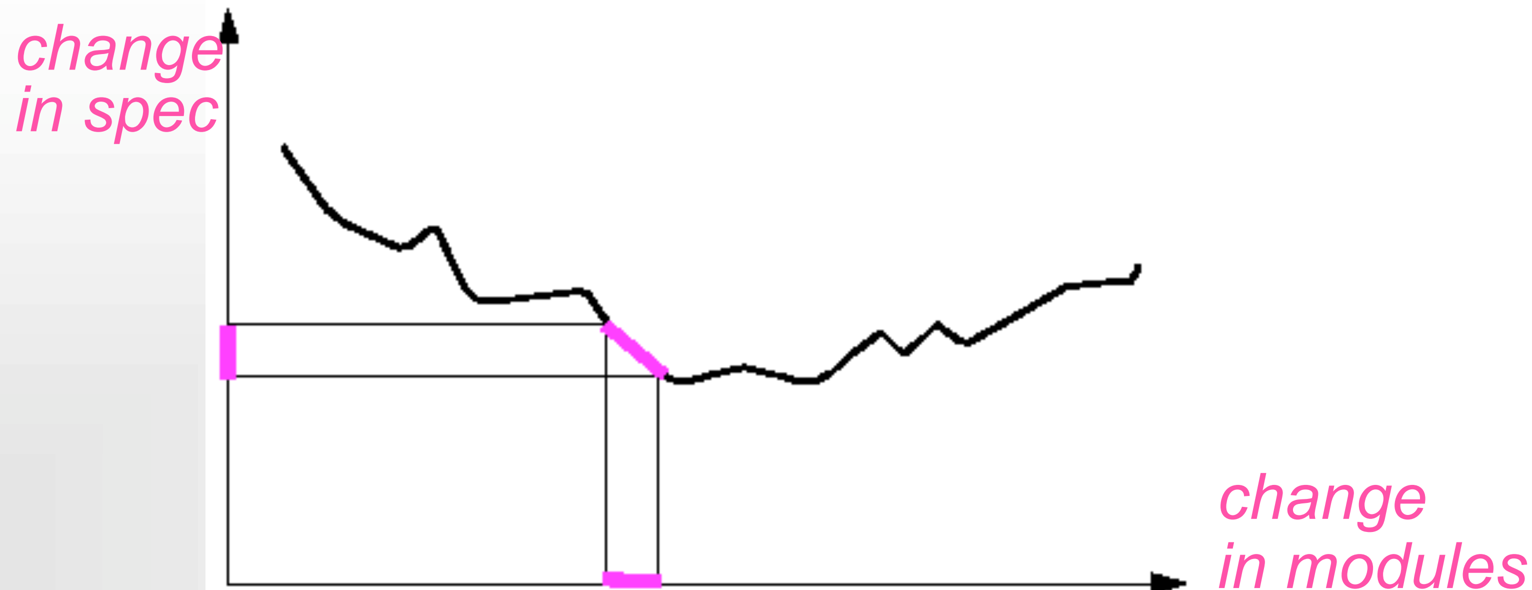
- some modules have been designed such that they will only function correctly if activated in a certain prescribed order
  - for example, B can only work properly if you execute it after A and before C, perhaps because they are meant for use in "piped" form as in the Unix notation: A | B | C

- then it is probably hard to understand B without understanding A and C, too

Modular Continuity Criterion

A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.

software_construction_method: specification -> system



Modular Continuity Criterion

example: symbolic constants

- style rule: instead of using numerical or textual constants directly, rely on symbolic names, and the actual values only appear in a constant definition
- if the value changes, the only thing to update is the constant definition
- this rule is a wise precaution for continuity since constants, in spite of their name, are remarkably prone to change

Modular Continuity Criterion

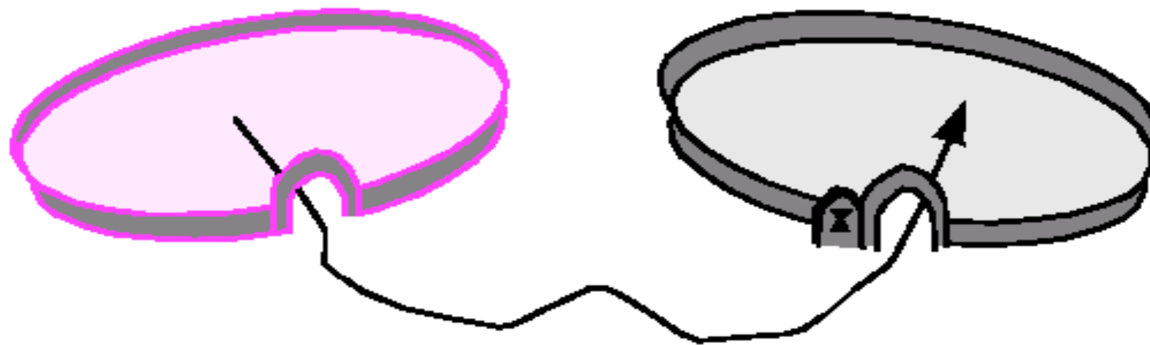
counter-examples: *using physical representations* and *static arrays*

- a method in which program designs are patterned after the physical implementation of data will yield designs that are very sensitive to slight changes in the environment

- languages such as Fortran or standard Pascal, which do not allow the declaration of arrays whose bounds will only be known at run-time ("dynamic arrays"), make program evolution much harder

Modular Protection Criterion

A method satisfies Modular Protection if 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 will only propagate to a few neighbouring modules.



does not address the avoidance or correction of errors, but the aspect directly relevant to modularity: their *propagation*

Modular Protection Criterion

example: validating input at the source

- a method requiring that you make every module that inputs data also responsible for checking their validity is good for modular protection (assertions)

counter-example: undisciplined exceptions

- exceptions make it possible to decouple the algorithms for normal cases from the processing of erroneous cases
  - but they must be used carefully to avoid hindering modular protection

Modularity / Five Rules

- from the criteria, five rules follow which we must observe to ensure modularity:
  - Direct Mapping
  - Few Interfaces
  - Small interfaces
  - Explicit Interfaces
  - Information Hiding

Modularity / Direct Mapping

The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain.

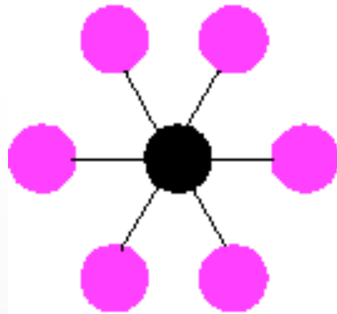
Program "what" and not "how"...

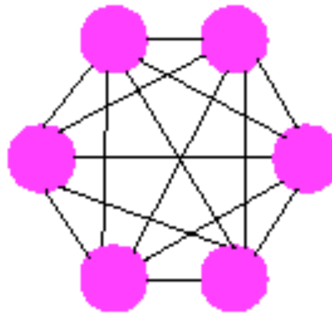
Modularity / Direct Mapping

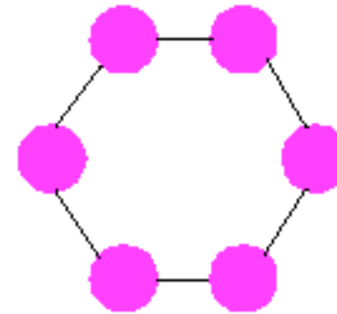
- follows from two of the criteria:
  - continuity: keeping a trace of the problem's modular structure in the solution's structure makes it easier to assess and limit the impact of changes
  - decomposability: if some work has already been done to analyse the modular structure of the problem domain, it may provide a good starting point for the modular decomposition of the software

Modularity / Few Interfaces

Every module should communicate with as few others as possible.


(A)


(B)


(C)

if a system is composed of *n* modules, the number of connections should remain much closer to the minimum, $n-1$ **(A)**, than to the maximum, $n(n-1)/2$ **(B)**

Modularity / Few Interfaces

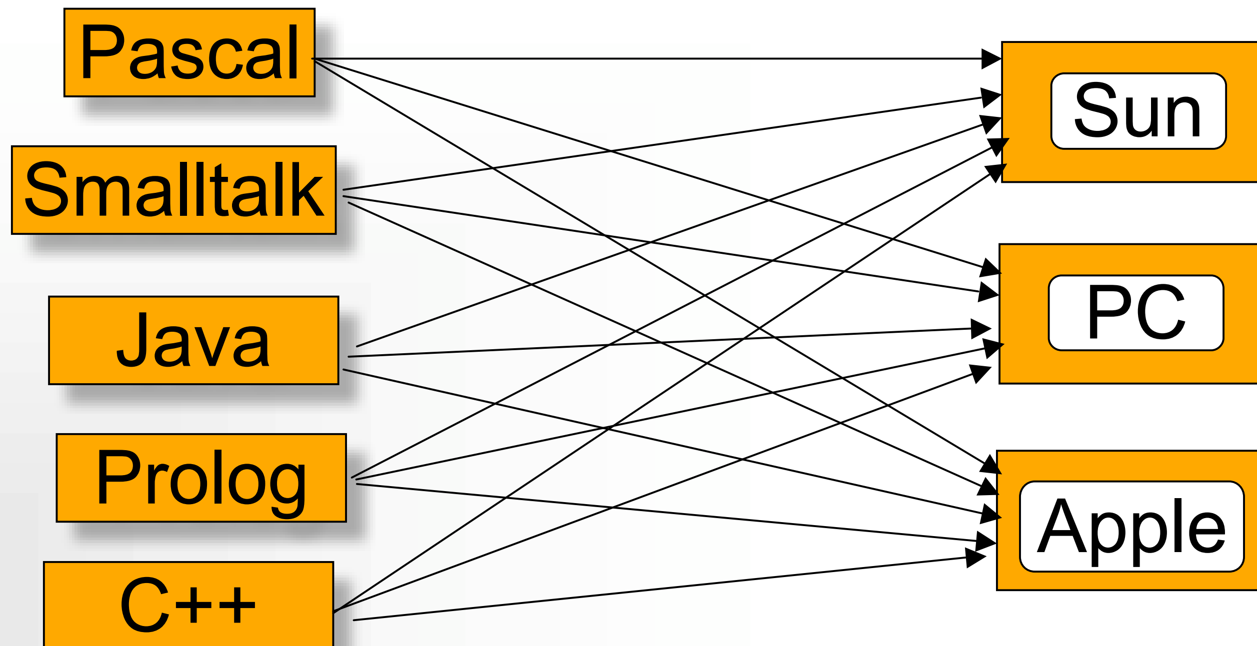
Every module should communicate with as few others as possible.

Design techniques:

- layering
  - layers only interact with adjacent layers
  - stop change avalanches
  - separate concerns
- adding one level of indirection
  - reduce connection explosion (n+m, not n*m)
  - introduce variation point (decoupling and selection)

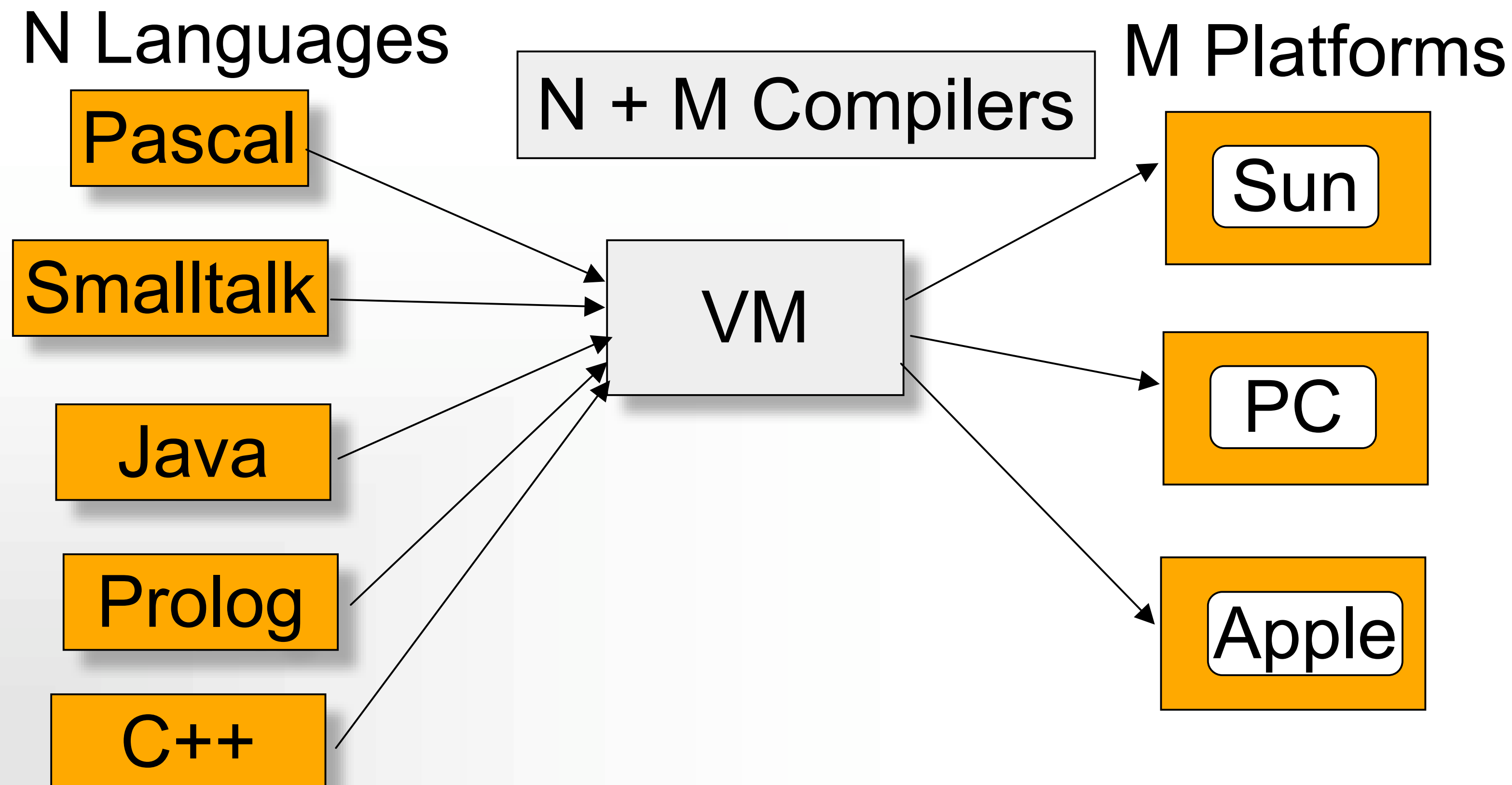
Modularity / Few Interfaces

One more level of indirection...



Modularity / Few Interfaces

One more level of indirection...



Modularity / Small Interfaces

If two modules communicate, they should exchange as little information as possible.

Modularity / Small Interfaces

- counter-example: Fortran's "garbage common block"

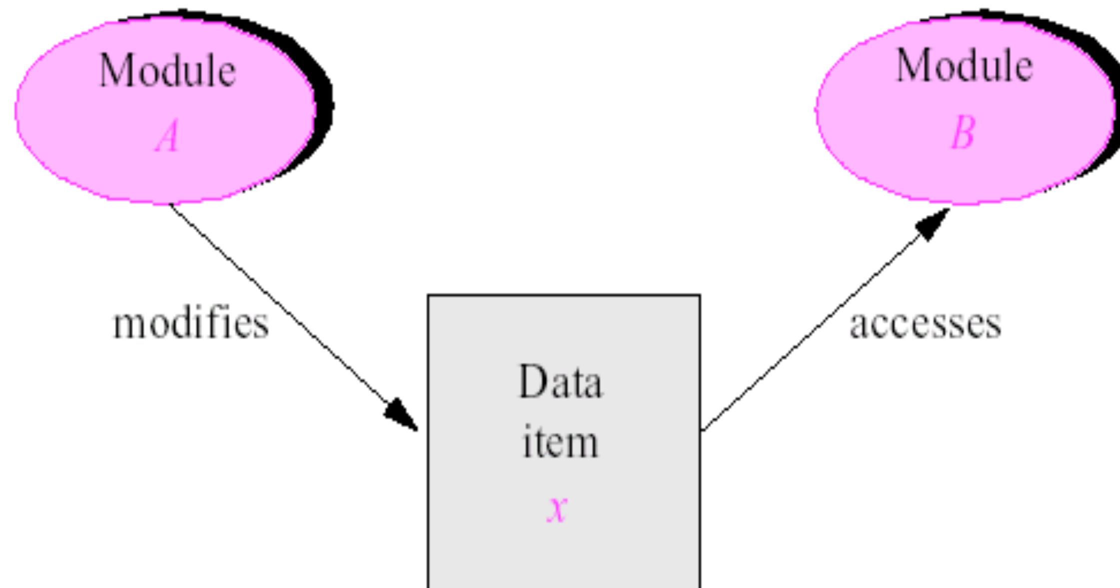
```
COMMON /common_name/ variable_1, ..., variable_n
```

  - the variables listed are accessible to any other module which includes a `COMMON` directive with the same `common_name`
  - every module may also misuse the common data; modular continuity and protection are particularly nastily ignored

Developers using languages with nested structures can suffer from similar troubles.

Modularity / Explicit Interfaces

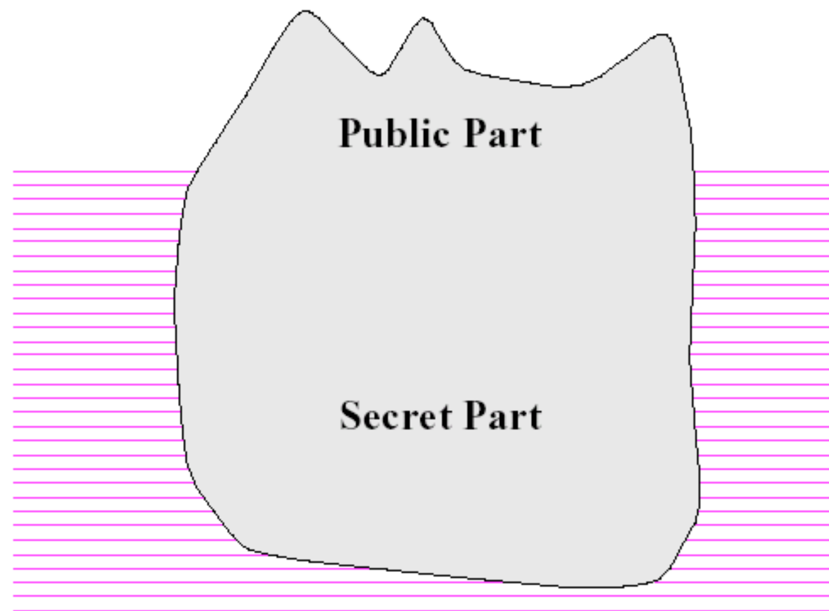
Whenever two modules A and B communicate, this must be obvious from the text of A or B or both.



totalitarian regime upon modules: not only should any conversation be limited to few participants and consist of just a few words; such conversations must be held in public!

Modularity / Information Hiding

The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.



design technique: abstraction

- ignore details in favour of the essential
- used in generalisation hierarchies

Modularity / Information Hiding

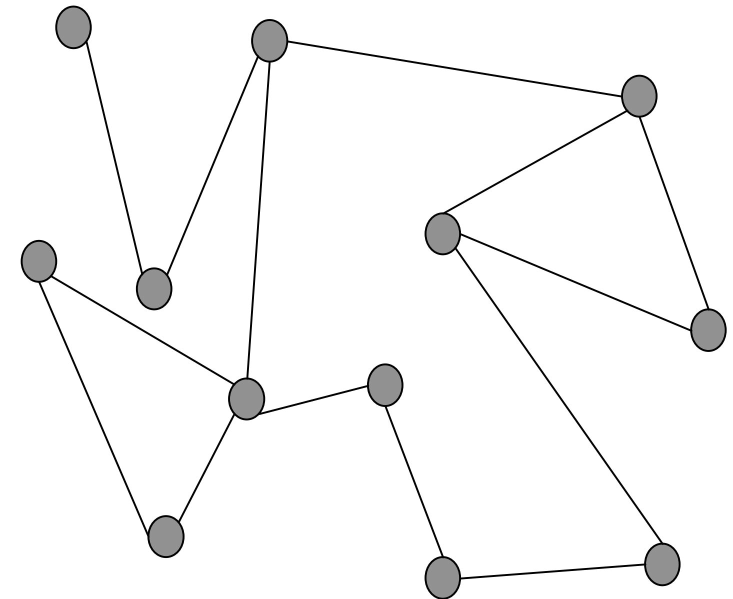
- the information hiding rule is crucial and helps in satisfying numerous modularity criteria

- continuity: changes to module details are not made public (prefer small public parts)

- emphasis on separating interface from implementation supports decomposability, composability, and understandability
  - the public parts of modules should specify their functionality

Low Coupling

- coupling measures "interconnectedness"
  - class C1 is coupled to class C2 if C1 requires C2 directly or indirectly
  - a class that depends on 2 other classes has a lower coupling than a class that depends on 8 other classes
  - coupling measures the strength of dependency between classes and packages



Low Coupling

- a class with high coupling is undesirable because:
  - changes in related classes force local changes
  - this class is harder to understand in isolation
  - this class is harder to reuse because its use requires the inclusion of all classes it is dependent upon

- no coupling at all is not desirable either (decomposability)

High Cohesion

- cohesion measures the strength of the relationship amongst elements of a class
  - classes with high cohesion can often be described by a simple sentence
  - all operations and data within a class should naturally “belong” to the concept the class models
  - when methods within a class use a "similar" set of instance variables, the class is considered highly cohesive

- *high cohesion* and *low coupling* are preferred

Types of Cohesion

- coincidental
  - no meaningful relationship amongst elements of a class
  - see the "Blob" anti-pattern
    - huge classes containing virtually all kinds of functionality that control everything and leave nothing to other classes (except data representation)
- logical cohesion
  - elements of a class perform one kind of a logical function, e.g., interfacing with the point of sale hardware
- temporal cohesion
  - all elements of a class are executed "together"

High Cohesion

- why classes with low cohesion are undesirable:
  - hard to comprehend
  - hard to reuse
  - hard to maintain - easily affected by change

- low cohesion indicates...
  - such classes are very coarse-grained abstractions
  - classes have been assigned responsibilities that should have been delegated to other objects

Cohesion Levels

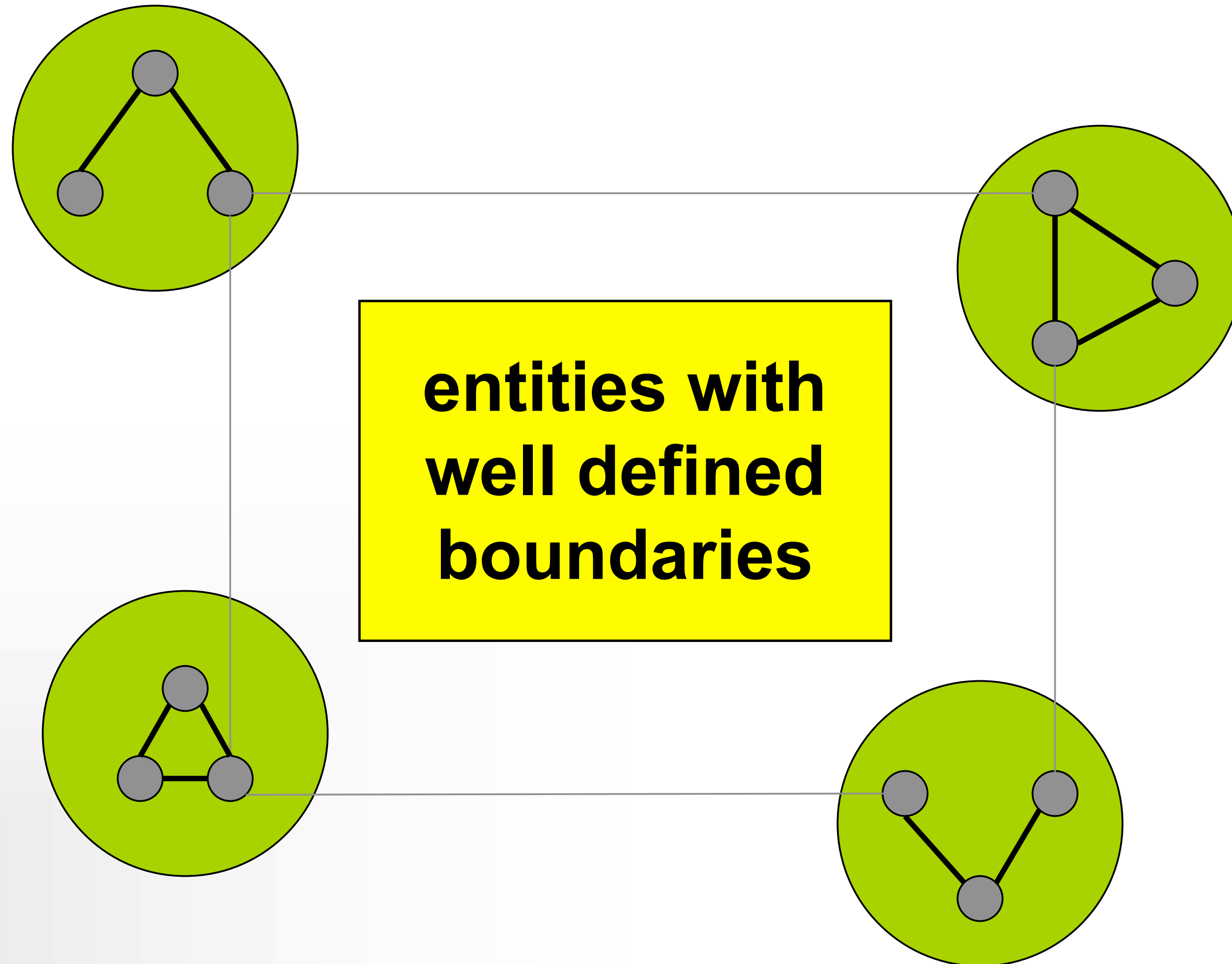
- very low cohesion: a class is responsible for many things in different functional areas
  - a class called RDB-RPC-Interface is responsible for interacting with relational databases *and* for handling remote procedure calls
  - split into two families of classes

- low cohesion: a class has sole responsibility for a complex task in one functional area
  - a class called RDBInterface responsible for interacting with relational databases. Its methods are related, but there are lots of them and they do too much
  - split into several lightweight classes sharing the work

Cohesion Levels

- moderate cohesion: a class has moderate responsibilities in a few different areas that are logically related to the class concept, but not to each other
  - a class Company responsible for (a) knowing its employees and (b) knowing its financial information
- high cohesion: a class has lightweight responsibilities in one area and collaborates with other classes to fulfill tasks
  - a class called DBInterface partially responsible for interacting with databases
  - it interacts with a number of other classes related to DB access

Low Coupling & High Cohesion



Modularity / Five Principles

- from the rules, and indirectly from the criteria, five principles of software construction follow:
  - Linguistic Modular Units principle
  - Self-Documentation principle
  - Uniform Access principle
  - Open-Closed principle
  - Single Choice principle

Modularity / Linguistic Modular Units

Modules must correspond to syntactic units in the language used.

The language may be a programming language, a design language, a specification language etc.

In the case of programming languages, modules should be separately compilable.

Modularity / Linguistic Modular Units

This principle excludes combining a method that suggests a certain module concept and a language that does not offer the modular construct, forcing software developers to perform manual translation or restructuring.

Such an approach defeats several of the modularity criteria and rules: continuity, direct mapping, decomposability, composability, protection.

Program "what" and not "how"...

Modularity / Self-Documentation

> The designer of a module should strive to make all information about the module part of the module itself.

Understandability, uniformity

documentation → internal documentation about components of the software, not user documentation about the resulting product

This principle precludes the common situation in which information about the module is kept in separate project documents.

Make programs look like designs!

Modularity / Uniform Access Principle

All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

In many programming languages, the expression denoting the application of f to x depends on what implementation the software developer has chosen for f: is the value stored along with x or must it be computed whenever requested?

Uniform access hides complexity and avoids reimplementation!

Modularity / Uniform Access Principle

- example: bank account
  - is the balance stored or computed from a history of withdrawals and deposits?
- no uniform access (Pascal)
  - stored: `account.balance`
  - computed: `balance(account)`
- uniform access (Java)
  - stored/computed: `account.balance()`
  - ensures continuity

Modularity / Open-Closed Principle

Modules should be both open and closed.

An open module is still available for extension. E.g., it should be possible to expand its set of operations or add fields to its data structures.

A closed module is available for use. It has a well-defined, stable interface. At the implementation level, closure also implies that you may compile it, store it in a library, and make it available for others (its *clients*) to use.

Modularity / Open-Closed Principle

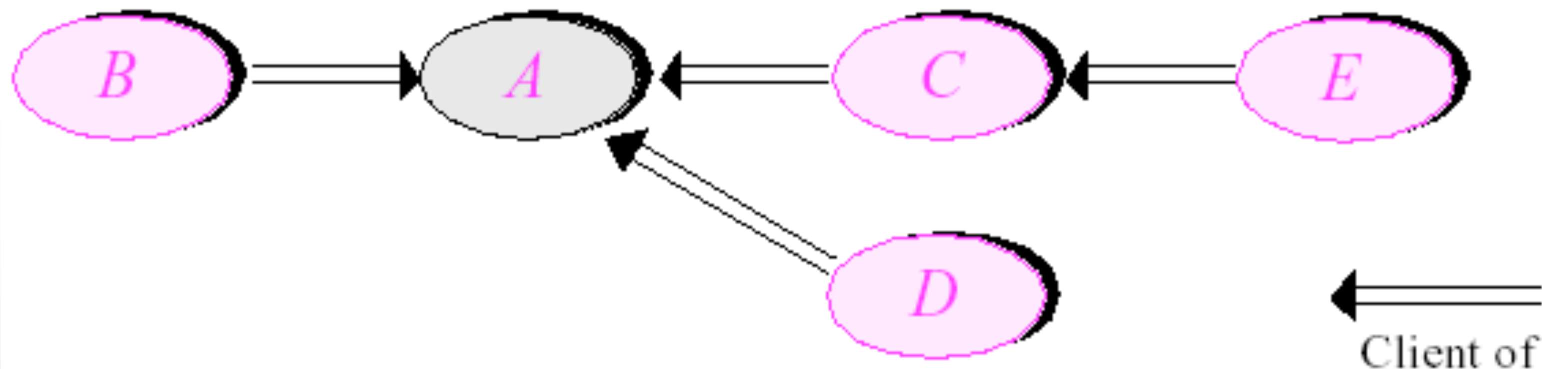
Openness is a natural concern → it is almost impossible to foresee all data and operations that a module will need in its lifetime

Closedness is a concern especially from a project manager's viewpoint: in a system comprising many modules, most will depend on some others; if we never closed a module until we were sure it includes all the needed features, no multi-module software would ever reach completion.

Modularity / Open-Closed Principle

Traditional approaches

module *A* is used by client modules *B*, *C*, *D*, which may themselves have their own clients (*E*, *F*, ...).

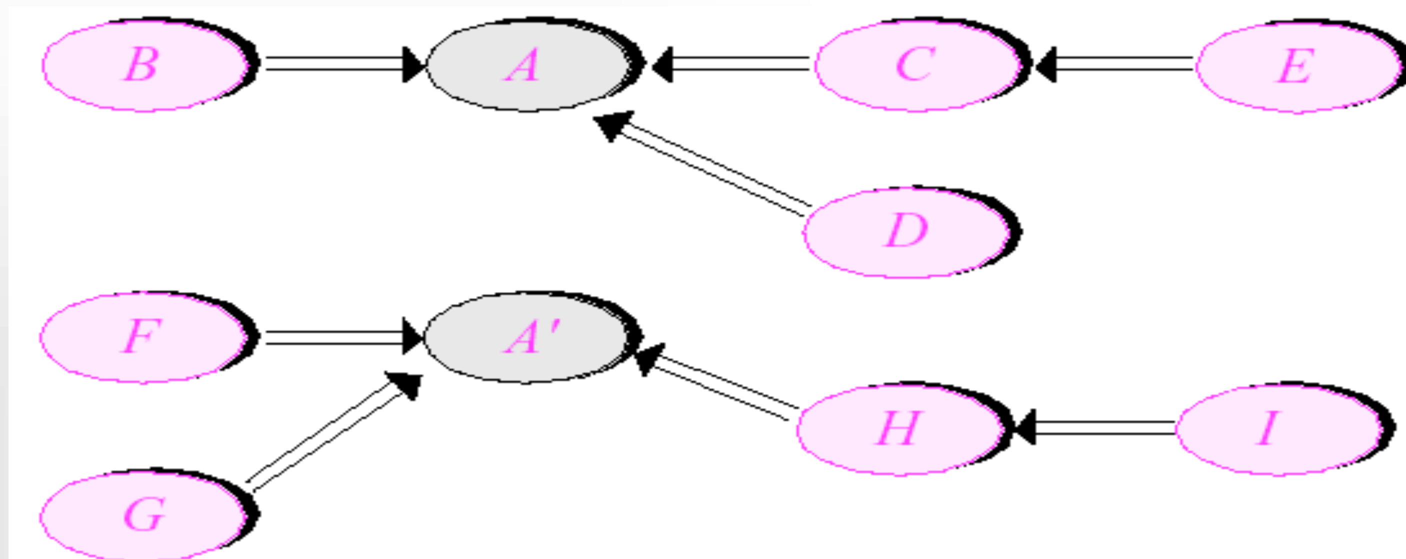


Later on, new clients arrive — *B'* and others — which need an extended or adapted version of *A*, which we may call *A´*

Modularity / Open-Closed Principle

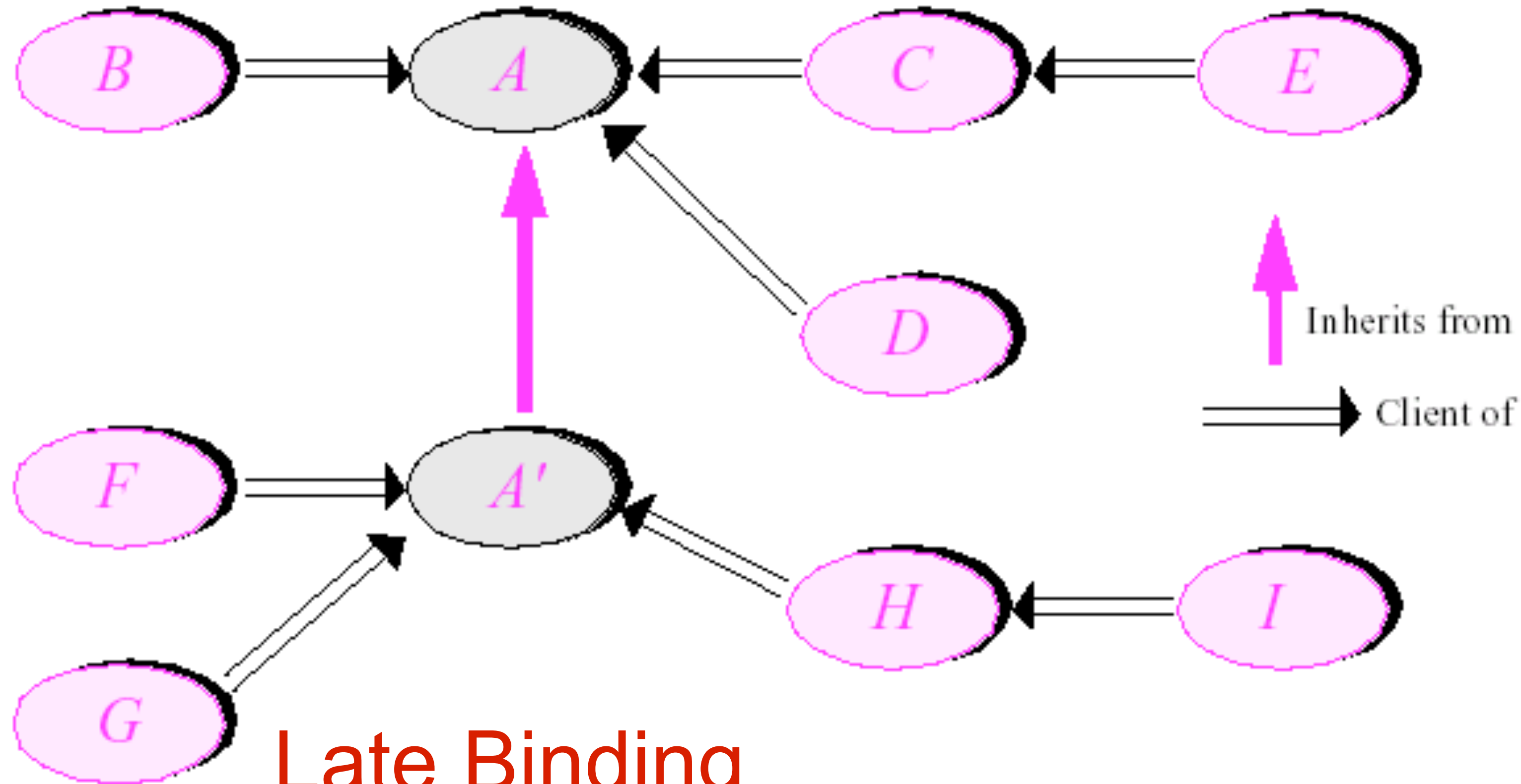
Traditional approaches

- adapt module *A* so that it will offer extended or modified functionality required by new clients
  - obvious potential for disaster: adaptations may violate assumptions of existing clients (leads to cascading change)
- leave *A* as it is, make a copy *A'*, perform necessary adaptations on *A'*; clients access *A'*, and it has no more connection to *A*
  - seemingly better (no modification of existing software), but introduces unnecessary redundancy (upon each new modification request)



Modularity / Open-Closed Principle

Inheritance



Late Binding

- no commitment to details (e.g., declarative style) defer responsibilities until you have the knowledge
- needed to choose the right variant

Modularity / Single Choice Principle

Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

By requiring that knowledge of the list of choices be confined to just one module, we prepare the scene for later changes

distribution of knowledge, or need-to-know policy

a strong form of Information Hiding and a consequence of open/closed principle: the designer of supplier modules such as *A* and *A'* seeks to hide information (regarding the precise list of variants available for a certain notion) from the clients.

Modularity versus Performance

- to make software easier to understand, one often makes changes that make the program run more slowly
- ignore performance issues in favour of design purity or in hope of faster hardware?
  - NO!
    - software has been rejected for being too slow → responsiveness!
    - essential for systems such as heart pacemakers, where late data is bad data!
    - faster machines merely move goalposts
- forget about modular design?
  - NO!
    - there is no inherent conflict between performance and modular design!
    - modularity helps write fast software!

Two Approaches to Performance

- constant attention approach
  - all programmers at all times do whatever they can for improving performance
- hot-spot driven attention approach
  - statistics show that most of the programs waste most of their time on a small fraction of code (~ 10%)
  - build the program in a well factored manner without paying attention to performance until you get at the performance tunig stage (PTS)
  - at PTS you follow a specific process to find the performance killer place and to tune the program

Which one do you prefer?

Modularity versus Performance

- constant attention?
  - performance changes make program harder to work with; this slows down development
  - the resulting software usually isn't quicker
  - changes spread all over the program and made with a narrow perspective of program’s behaviour
  - after all, 90% of the time spent on these changes is wasted time!

Modularity versus Performance

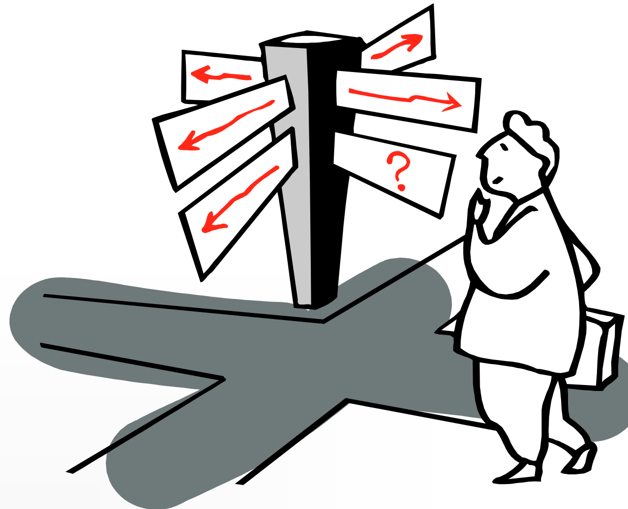
- hot-spot driven attention?
  - begin PTS by running a profiler; find the hot-spot
  - after each change you compile, test, and profile
  - undo change if it wasn’t effective
  - by focusing the attention on the hot-spot, much more effect is achieved for less work

Modularity versus Performance

- well-modularised programs help with hot-spot driven optimisation:
  - development time is shorter, hence you get more time for performance tuning
  - finer-granularity for performance analysis
  - cleaner code – better understanding of options

Modularity AND Performance ?

modularisation slows down software in the short term



It helps writing faster software in the long term because it makes software easy to tune during optimisations

OO Techniques

The Paradigm

- a code organisation principle
  - gather functions around data

The Mechanisms

- encapsulation
  - hide implementation details
- polymorphism
  - generic code
- inheritance
  - reuse of superclass code

If you had to pick two out of the three: What would be your choice?

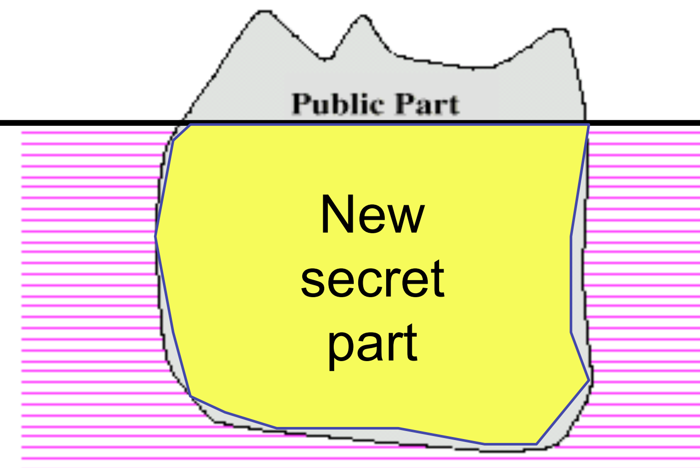
First-Class Data

Year 2000 problem

- implementation decision
  - 2 digit year representation
  - spread throughout application
- maintenance nightmare
  - difficult to find all references
  - difficult to find all Date implementations
- use of a single Date abstraction instead
  - local change with no rippling effects

Encapsulation

```
class Queue
{
  public int size();
  public insert(Element e);
  ---
  private Element elements[];
  private putAt(int pos, Element e);
  ...
}
```



Encapsulation (2)

Getters & Setters

```
double amount;

amount = account.balance()
amount -= 200;
account.setBalance(amount);
```

type of internal money representation exposed to client!

account possibly manipulated by others in between (not only in a concurrent setting)

Messages as Goals

```
account.withdraw(200);
```

Encapsulation (3)

Getters & Setters

```
Date d="23/11/2002";

book.checkedOut(true);
book.loanDate(d);
book.borrowedBy("Joe");
```

type of internal date representation must be known to client!

Messages as Goals

```
book.checkOutFor("Joe");
```

method will create a date itself, ensuring accurracy of information!

Encapsulation (4)

Sending a Sequence of Messages

```
Account a;

a = bank.customer("fred").account();
```

not the business of the client to know that banks have customers

---

return type of "customer()" could change

Law of Demeter

```
a = bank.accountFor("fred");
```

Encapsulation (5)

Violating the Dilbert Principle

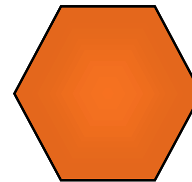
```
total := 0.
aPlant billings do: [:each |
  (each status == #paid and: [each date > startDate])
  ifTrue: [total := total + each amount]
].
```

Let someone else do the work for you

```
total := aPlant totalBillingsPaidSince: startDate.
```

Polymorphism

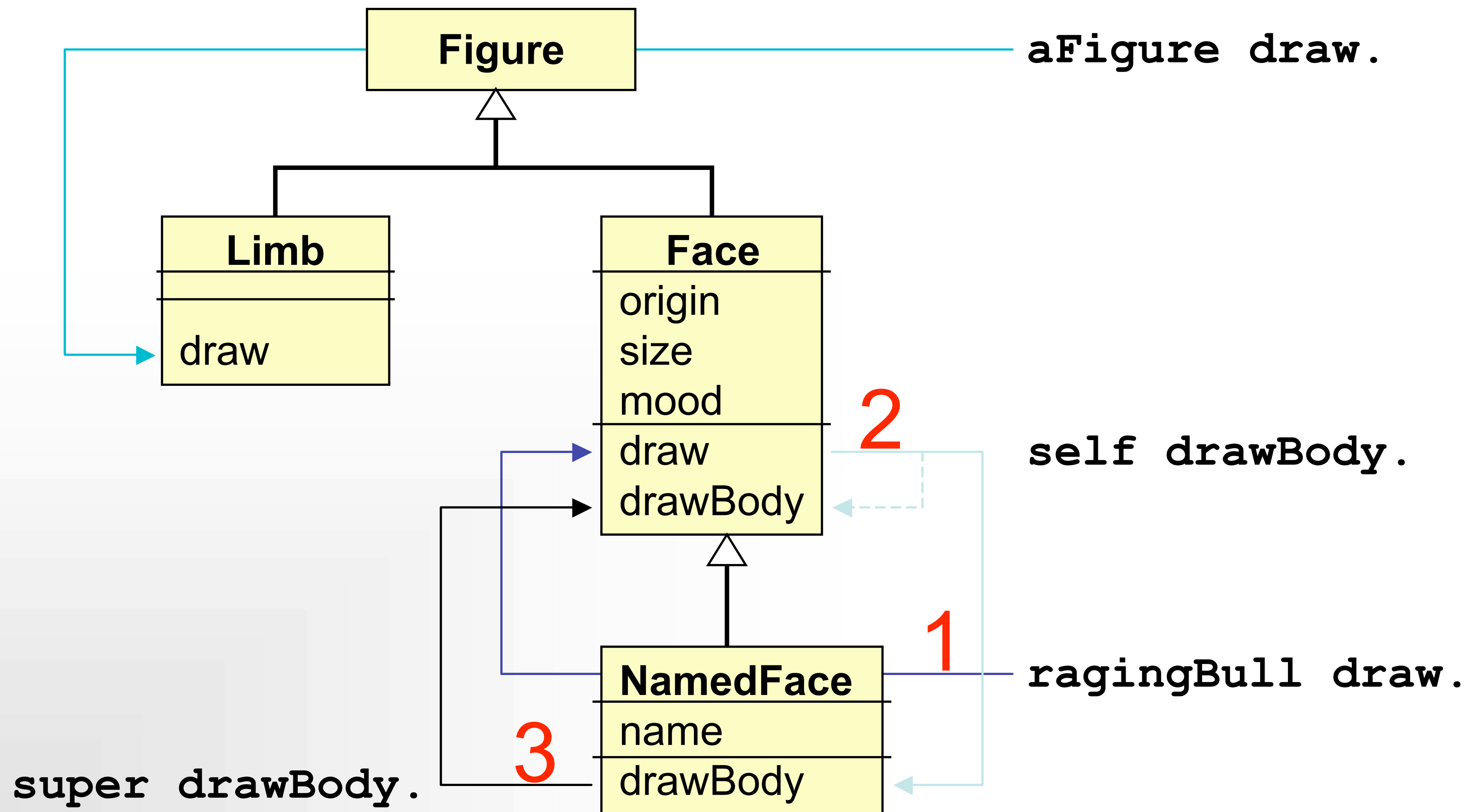
```
figures add: face; add: bull; add: hexagon.
figures do: [ :figure | figure draw ].
```



Execute

- one of several subclass methods
- a parent method
- parent method from subclass method
- subclass method from parent method

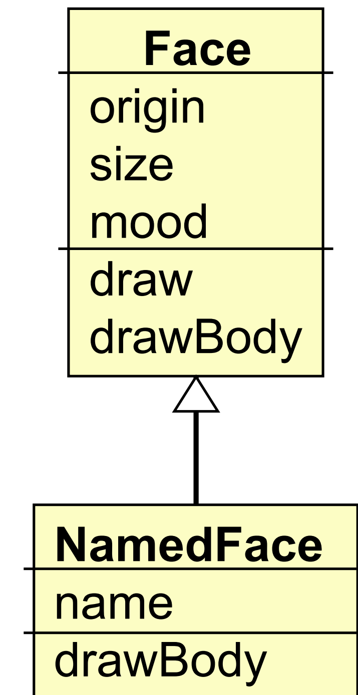
Polymorphic Messages



Inheritance

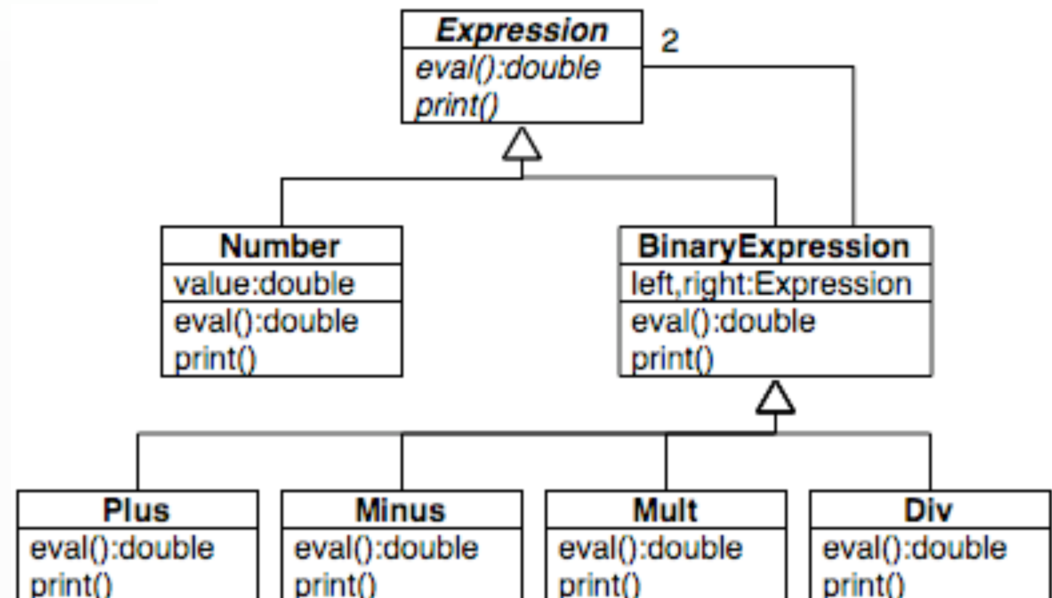
Advantages

- subclass needs to provide changed/additional features only
- code duplication avoided
- single place of change/correction
- possible to specify policies (e.g., draw) which rely on subclass mechanisms (drawBody)
- implements open/closed principle



Modelling Arithmetic Expressions

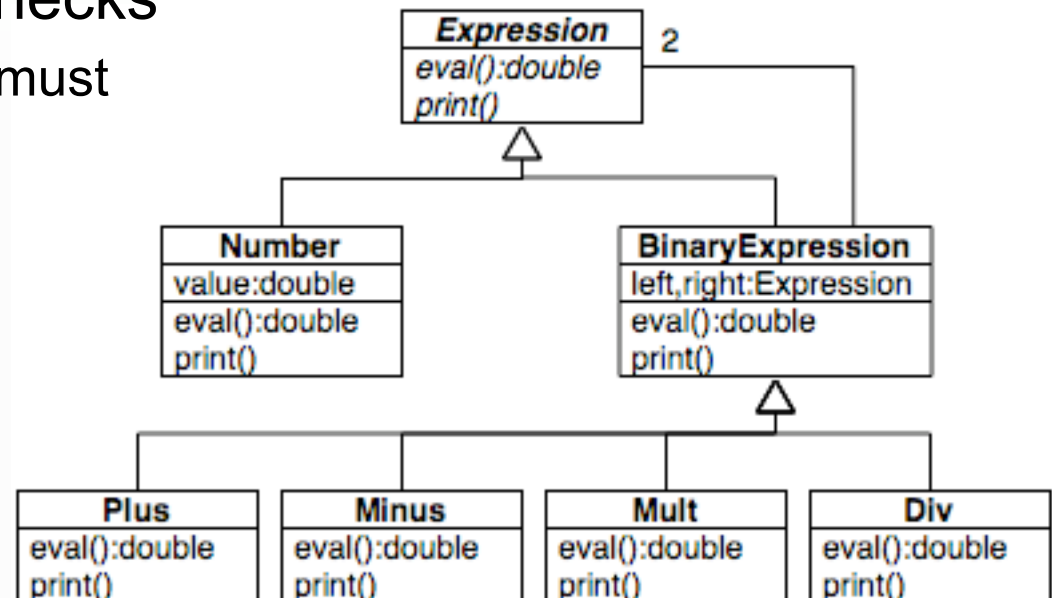
- requirements
  - a representation for arithmetic expressions
  - only numbers and basic operations
  - capable of evaluating expressions
  - capable of pretty-printing
- object-oriented decomposition



- evolution ideas
  - new expressions
  - persistence
  - syntax checks
  - later: additional semantic checks

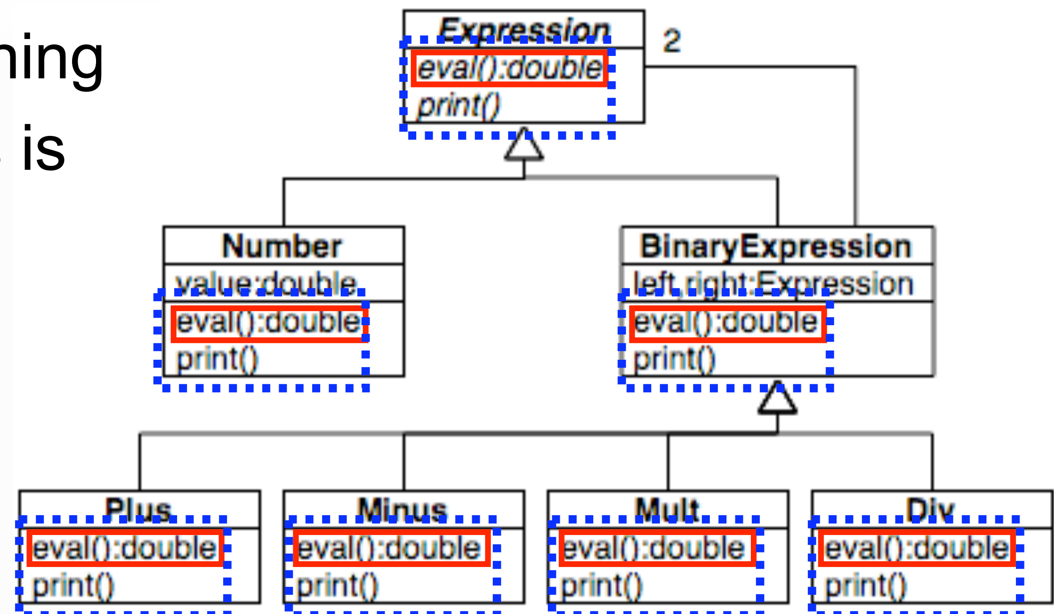
Evolution Problems

- new expressions are easy
  - add new subtype (e. g., `UnaryExpression` and `UnaryMinus`)
- persistence
  - add `save()` and `retrieve()` methods to *all* classes
- syntax checks
  - ditto: add `check()` to *all* classes
- later: additional semantic checks
  - implementation of `check()` must be *modified* in *all* classes



Separation of Concerns

- observation
  - requirements decompose by tool, or feature
  - design and code decompose by object
- two problems frequently met in OO decompositions
  - scattering: a single requirement affects multiple design and code modules
  - tangling: material pertaining to multiple requirements is interleaved within a single module

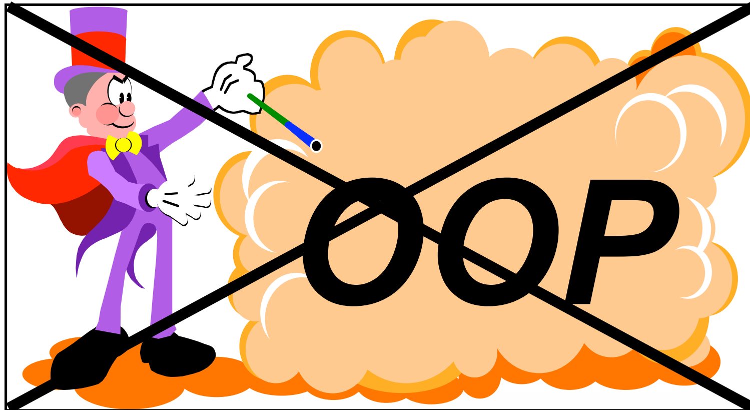


Separation of Concerns

- this is the "tyranny of the dominant decomposition"
  - modularisations support only small sets of decompositions
  - typically, only a single dominant one can be cleanly applied
  - it satisfies important needs, but hinders others from a clean realisation

- tendencies in other abstractions
  - data abstraction: scattered functionality
  - functional abstraction: scattered data representation
  - tool abstraction: increasingly difficult tool interactions

OOP Needs Principles



OOP is not a panacea, it helps you solve the problem, but doesn't solve the problem for you.

Some Known OO Defects

- inheritance
  - weak encapsulation
  - bad dynamics
  - weak support for black-box reuse
- state
  - hard to control
  - protocol based interfaces
- data centered
  - limited functional extensibility
  - asymmetric functions
- references
  - sharing
  - "null pointer" exceptions

Putting Reuse Mechanisms to Work

- building reusable OO software requires deep understanding of
  - OO concepts: objects, interfaces, classes, and inheritance
  - reuse mechanisms related to them
    - inheritance versus object composition
    - inheritance versus templates
- the challenge remains in applying them to build flexible, reusable software

Putting Reuse Mechanisms to Work

- the key to maximising reuse:
  - anticipate new requirements and/or changes to existing requirements
  - design systems so that they can evolve accordingly
- a design that doesn't take change into account is almost sure to need redesign in the future:
  - class redefinition and reimplementation,
  - client modification,
  - retesting
- unanticipated changes are invariably expensive

Some Causes of Redesign

- tight coupling between classes
  - classes are hard to reuse in isolation
  - leads to monolithic systems: can't change or remove a class without understanding and changing other classes
  - the system becomes a dense mass that's hard to learn, port, and maintain
  - loose coupling increases the probability that a class can be reused by itself and that a system can be learned, ported, modified, and extended more easily

Some Causes of Redesign

- low cohesion
  - too many unrelated aspects tangled into one class
  - bloated interface makes the class hard to understand and evolve
  - changing one aspect may affect other aspects
  - (modular decomposition, composition, understandability, continuity, protection)

Some Causes of Redesign

- object creation by specifying a class explicitly
  - commitment to a particular implementation instead of a particular interface can complicate future changes (open/closed, single choice)
- dependence on specific operations
  - commitment to a particular way to satisfying a request hard to change the way a request gets satisfied both at compile-time and at run-time (small interfaces)

Some Causes of Redesign

- dependence on hardware / software platform
  - harder to port to other platforms
  - even difficult to keep it up to date on its native platform (information hiding, single choice)
- dependence on object representation/implementation
  - clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes (small interface, uniform access)

Some Causes of Redesign

- algorithmic dependencies
  - algorithms are often extended, optimized, and replaced during development and reuse
  - objects that depend on an algorithm will have to change when the algorithm changes
  - information hiding, open/close, single choice

Design patterns encode ways of structuring design such that changes like this are facilitated.