SWEN 223
Software Engineering Analysis

Design Principles

Thomas Kühne
Victoria University of Wellington
Thomas.Kuehne@ecs.vuw.ac.nz, Ext. 5443, Room Cotton 233

What is Good Software?

Goals & Solutions

- External Criteria
  (what clients expect)
  - correctness, robustness, efficiency
- Internal Criteria
  (regarding solution)
  - extensibility, modularity …
- Design Techniques
  (how to meet internal criteria)
  - principles, rules, best practices

Only external factors really matter.

But the key to achieving them are the internal ones

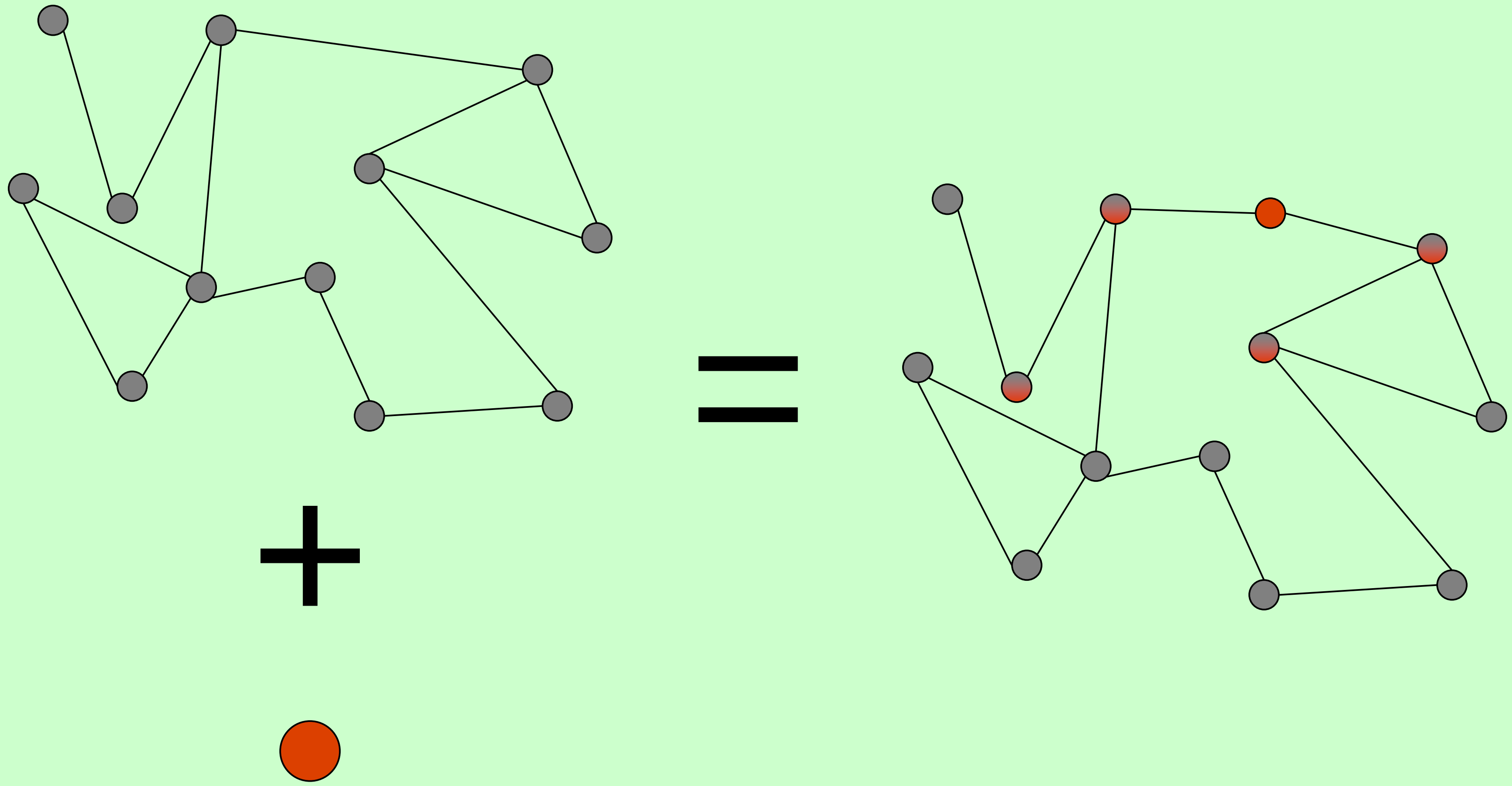
Bad Design

- A piece of software has a bad design if it fulfils its requirements and yet exhibits any or all of the following three traits
  - » Rigidity It is hard to change because every change affects too many other parts of the system
  - » Fragility When you make a change, unexpected parts of the system break
  - » Immobility It is hard to reuse in another application because it cannot be disentangled from the current application

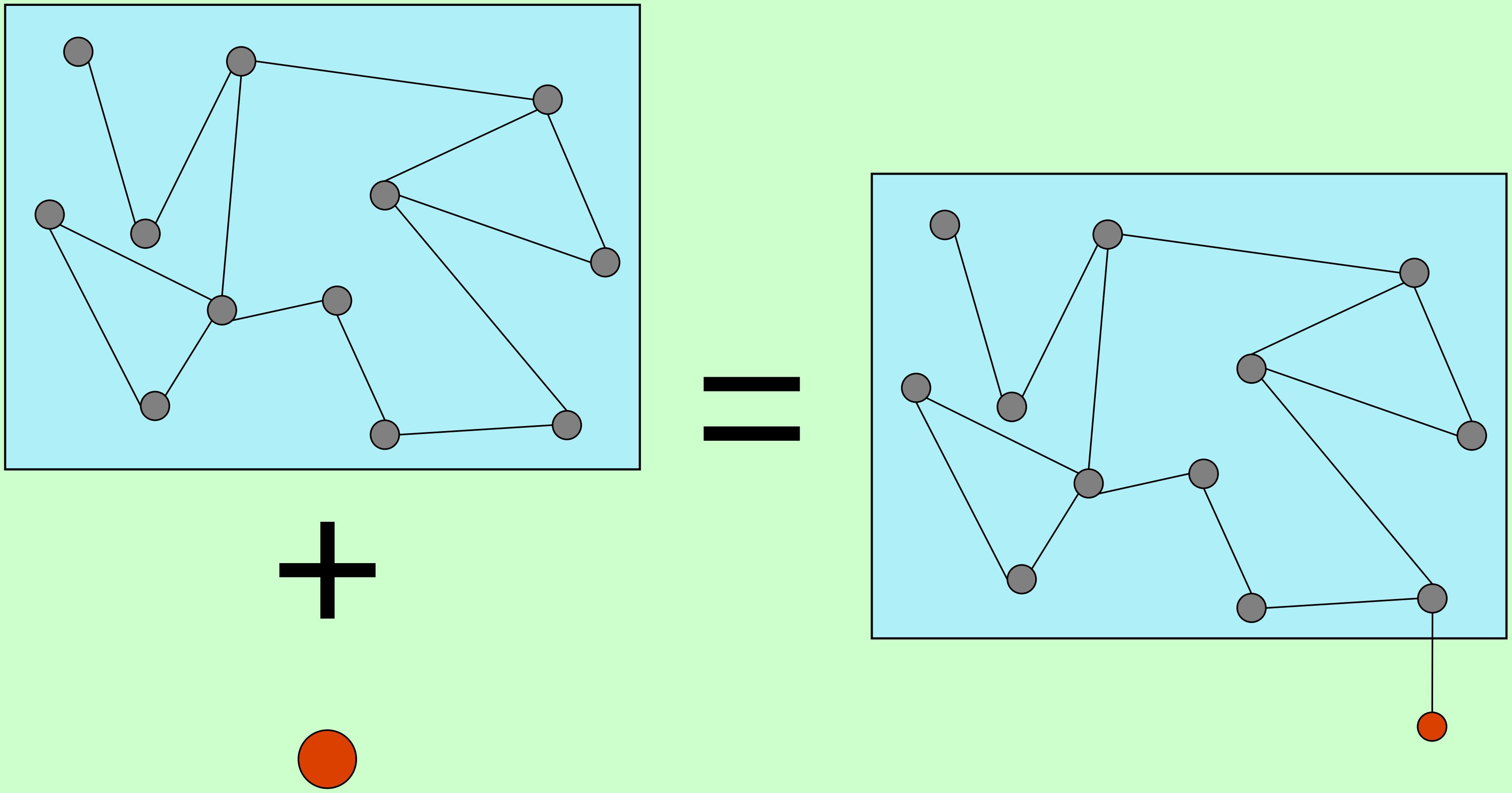
Extensibility

- Extensibility characterizes the ease of adapting software to changes of specification
  - » …and we know the specification is going to change
- "Big" is often "Bad“
  - » as software grows bigger,
    it becomes harder to adapt
  - » …unless, appropriate counter measures are taken

Bad Extensibility



Good Extensibility



Modularity

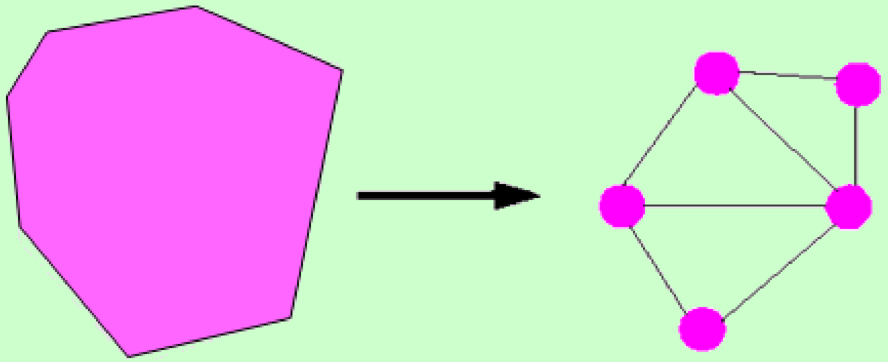
- Any mechanism that works towards producing self-contained building blocks, communicating through restricted and explicit channels only,
  improves modularity
- Modularity aids both
  - extensibility
  - reusability

Five Fundamental Requirements

- Decomposability & Composability
  - how to separate & combine
- Understandability
  - if you want to change, you need to know what
- Continuity
  - avoiding the "change avalanche"
- Protection
  - avoiding "error creep"

Decomposability

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 sub-problems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them.



Divide
&
Conquer

Decomposability

- Division of labor!
  - » decomposition aids parallel development

→ dependencies must be

  - » explicit, as they form the interfaces for cooperative work
  - » few, as every dependency increases communication overhead & adds to system rigidity / fragility

Decomposability

Positive Example

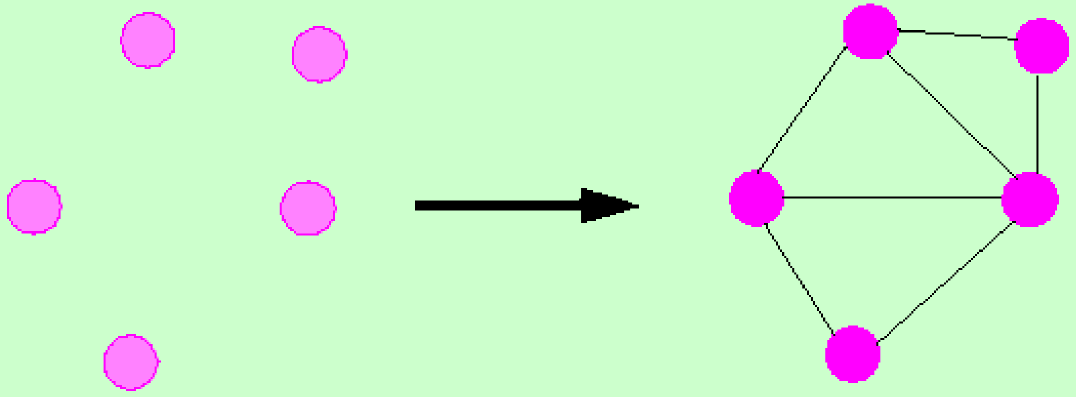
- » structured analysis & design
- » OO classes

Negative Example

- » global, common initialization module: endangers autonomy of participating modules

Composability

A design method satisfies Modular Composability if it favors 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.



Composability

- Reusability!
  - components are required to be sufficiently independent from the immediate goal that led to their existence

→ designated tasks must be

- not too narrow,
  so that reuse makes sense
- well-defined,
  so that the components are usable

Composability

Positive Example

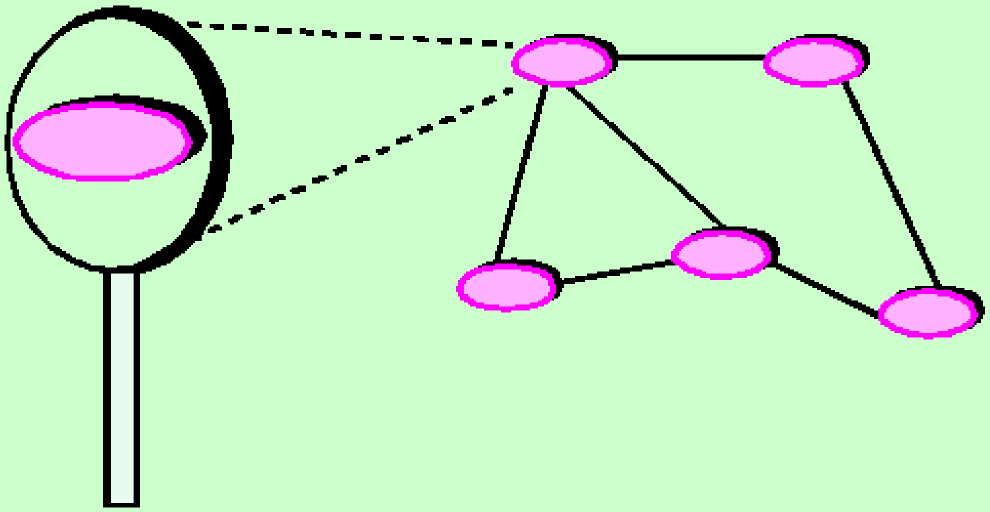
» Unix shell pipes: processing of ASCII streams enables numerous combinations

Negative Example

» language extensions through preprocessors: any single solution works but cannot be combined with others

Understandability

A method favors 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.



Understandability

Positive Example

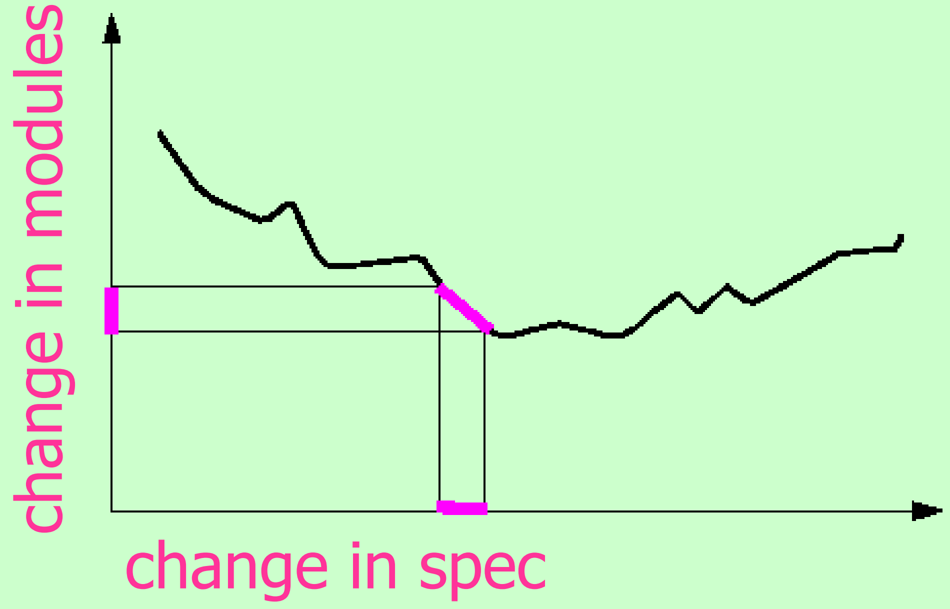
» lazy initialization

Negative Example

» modules, only working correctly if activated in a certain prescribed order; for example, B can only work properly if you execute it after A and before C.

Continuity

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.



Continuity

Positive Example

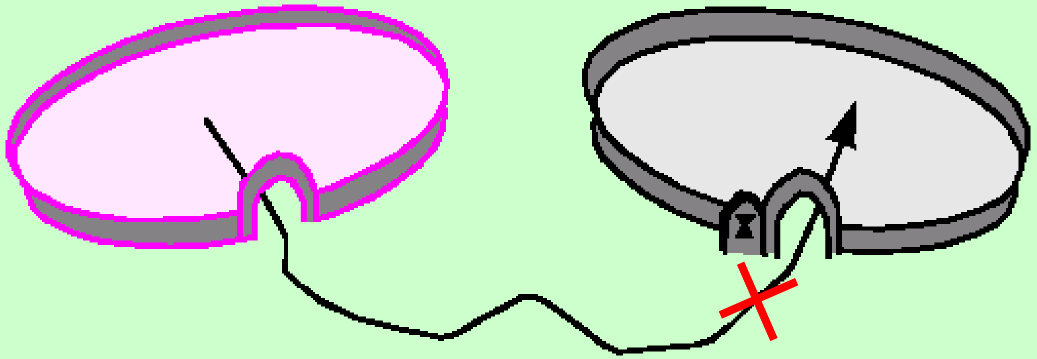
- symbolic constants: If a value changes, the only thing to update is the constant definition instead of numerous occurrences of the value

Negative Example

- case analysis over object types

Protection

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 neighboring modules.



not about avoiding or correcting errors but their propagation

Protection

Positive Example

» preconditions: checking the validity of input data before it is used

Negative Example

» undisciplined exceptions: unless caught at appropriate places, they can create and propagate errors without bounds

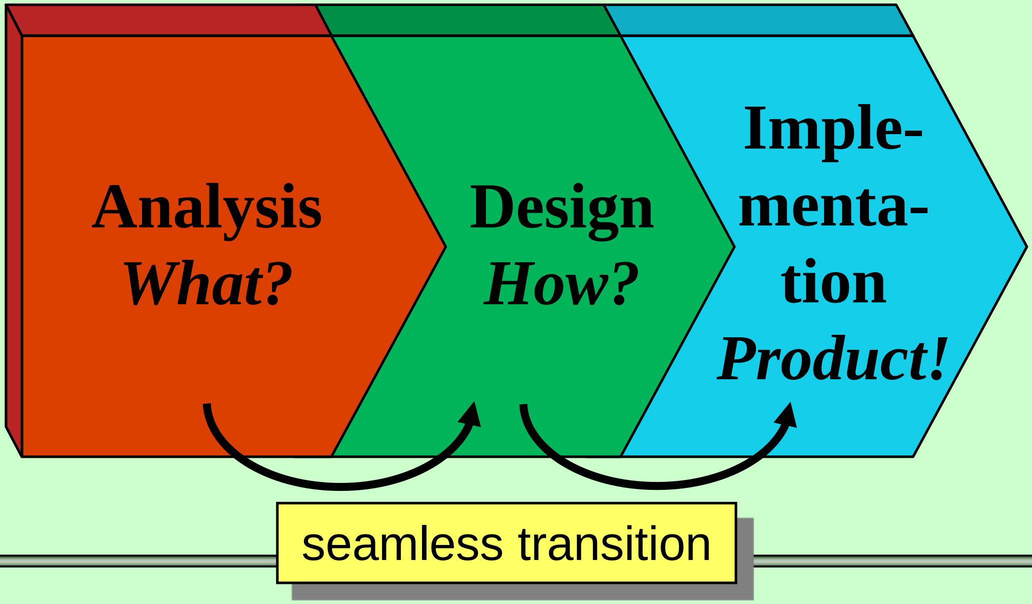
Obtaining Modularity

From the five modularity requirements, we may derive **rules** which we must observe to ensure modularity

- Direct Mapping
- Interfaces
  - few, small, and explicit
- Information Hiding

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 modelling the problem domain.



Direct Mapping

Related to the criteria

- Decomposability
  - » the analysis of the problem domain structure may provide a good starting point for the modular decomposition of the software
- 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

Few Interfaces

Every module should communicate with as few others as possible.

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

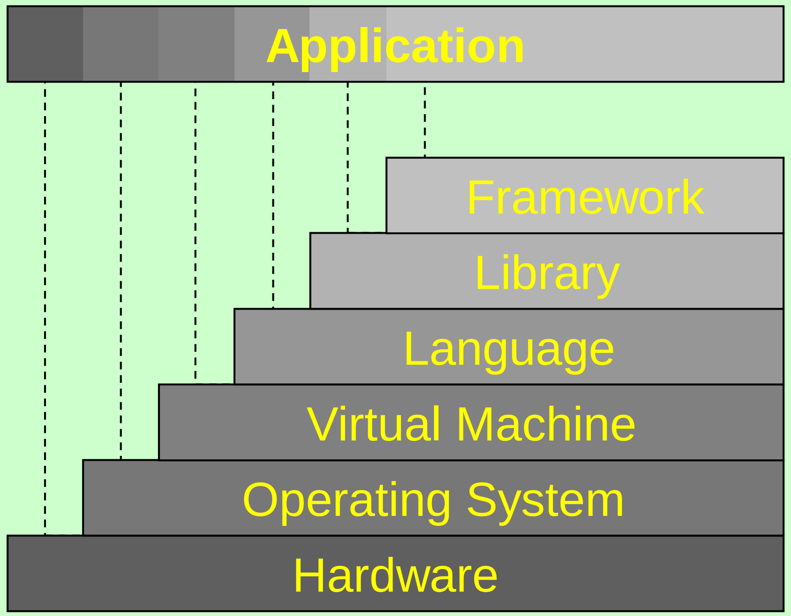
Few Interfaces

How to reduce the interface count

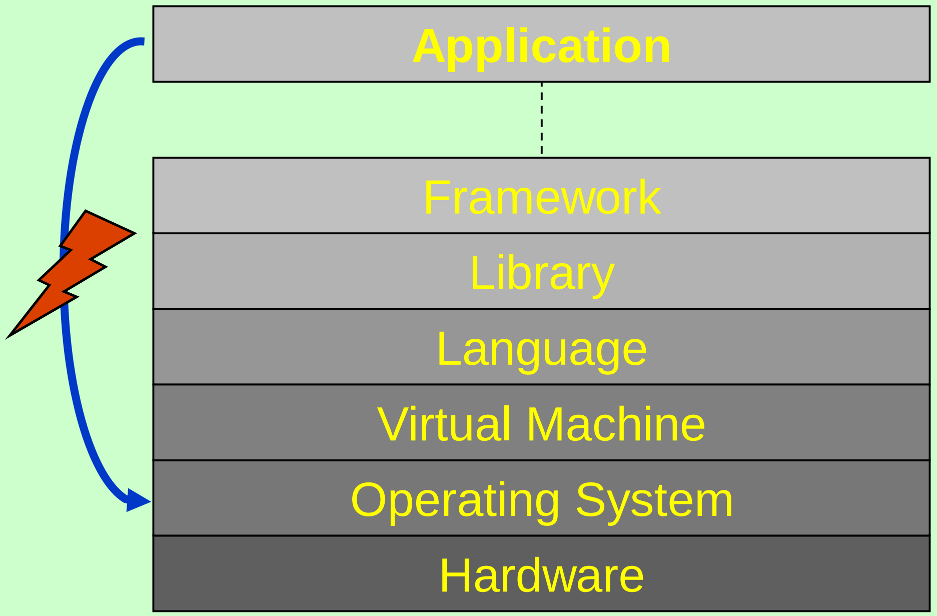
- Layering
  - » layers only interact with adjacent layers, stops change avalanches, separates concerns

Few Interfaces

Loose Layering



Strict Layering



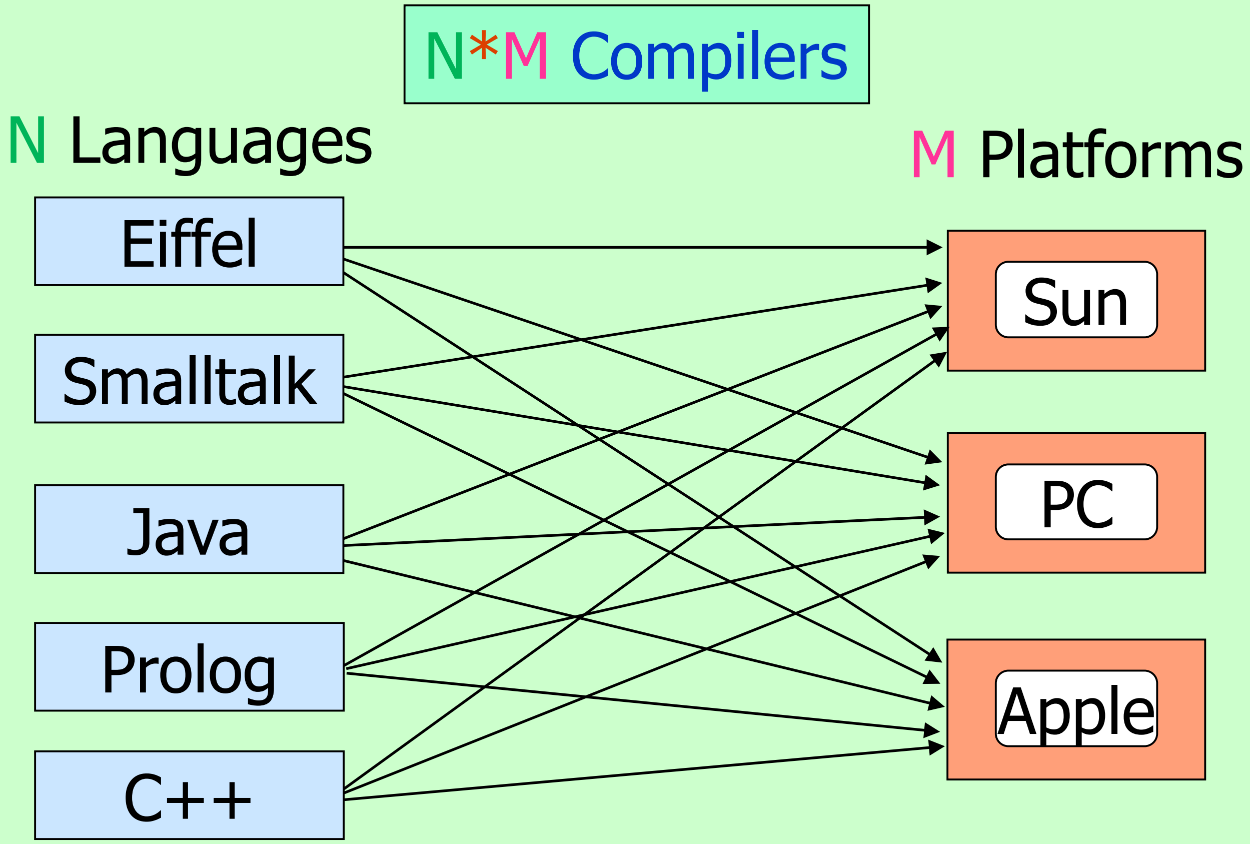
- Loose version is more efficient but also more fragile

Few Interfaces

How to reduce the interface count

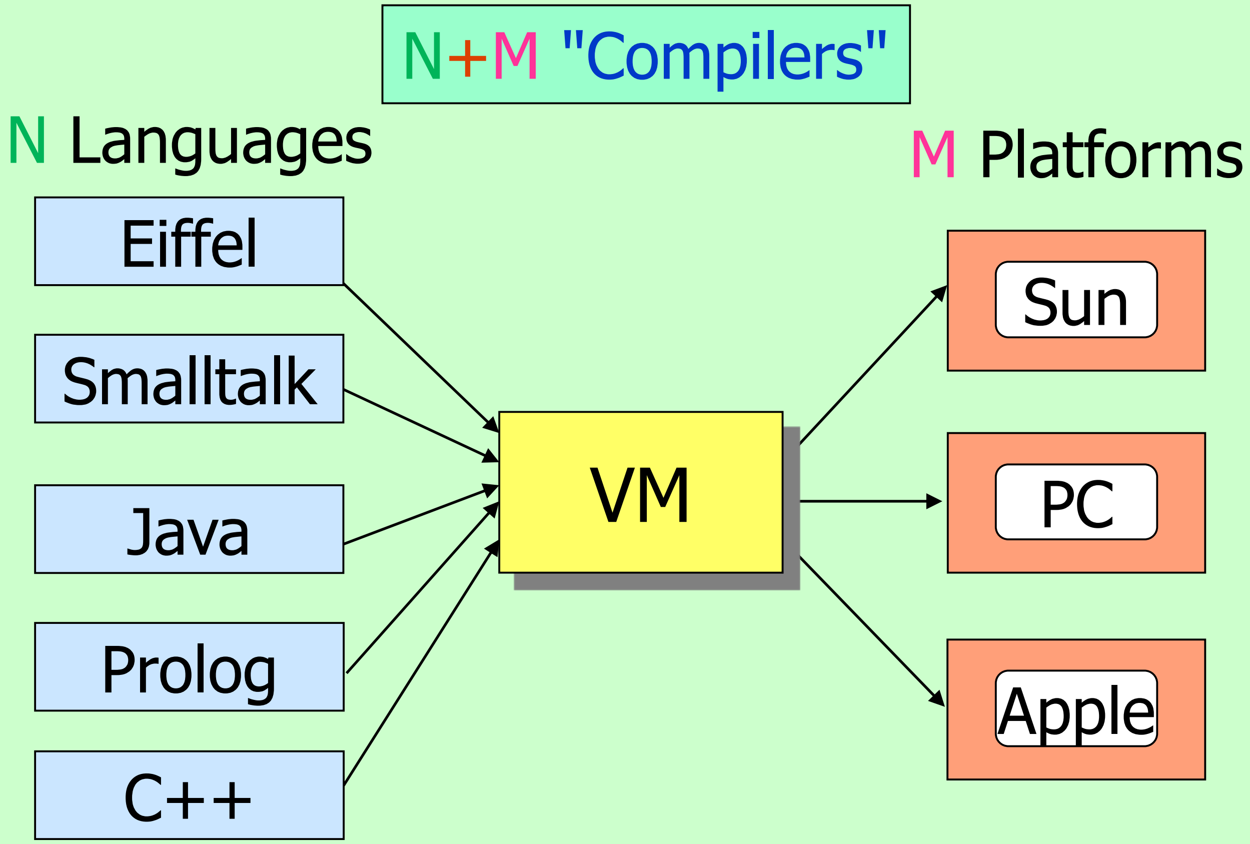
- Layering
  - » layers only interact with adjacent layers,
    stops change avalanches,
    separates concerns
- One more level of indirection ...
  - » reduce connection explosion (n+m not n*m)
  - » introduce variation point
    (decoupling & selection)

One more level of indirection



Few Interfaces

One more level of indirection



Caveat

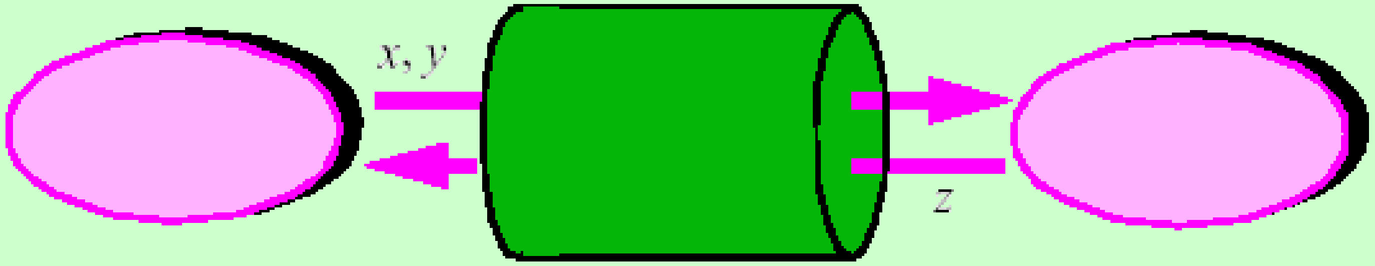
There is no problem in computer science that cannot be solved by adding yet another level of indirection.

BUT

There is no performance problem that cannot be solved by removing a level of indirection.

Small Interfaces

Two communicating modules, should exchange as little information as possible.



Small Interfaces

Counter-example: Fortran's "common block"

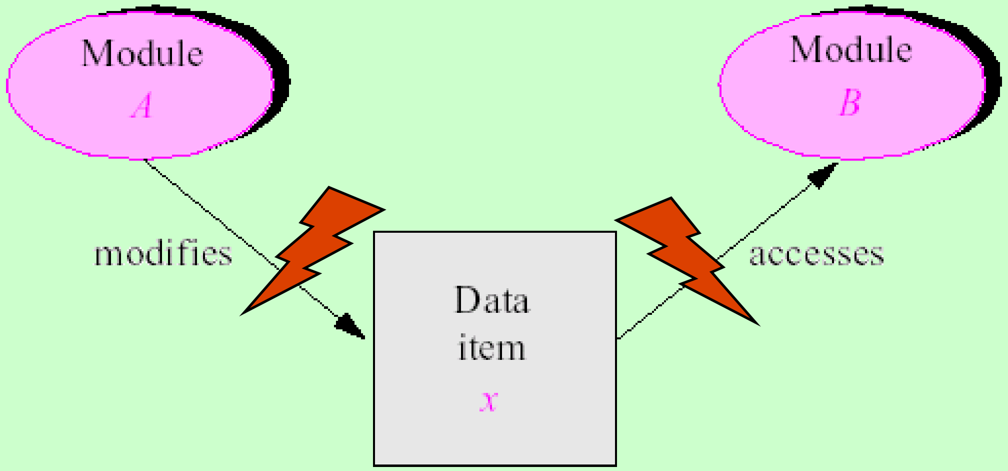
```
COMMON /common_name / variable_1, ... variable_n
```

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

Languages with nested structures may suffer from similar trouble

Explicit Interfaces

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

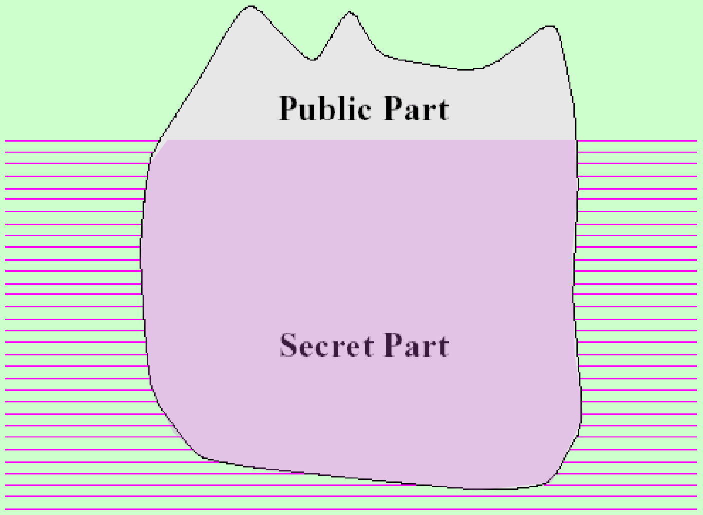


A Totalitarian Regime Upon Modules

Not only should any conversation be limi-ted to few participants and consist of just a few words; such conversations must be held in public!

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.



addresses
Continuity
and is supported by object-oriented encapsulation

More Modularity Heuristics

- "What" not "How"
  - » request behavior,
    instead of spelling out the solution
- Low Coupling & High Cohesion
  - » use interfaces to decouple entities
  - » create entities with a well-defined meaning

"What" not "How"

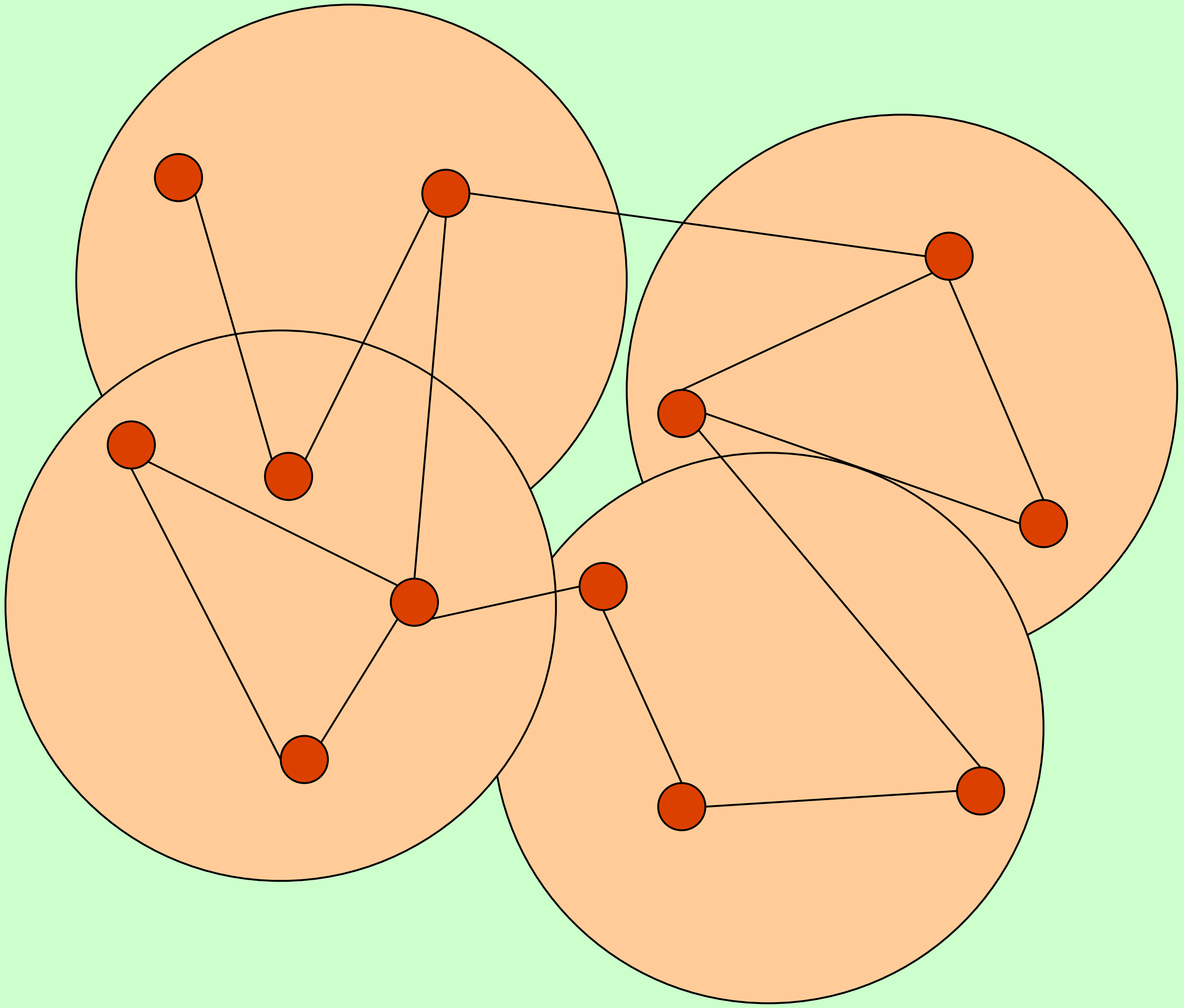
Violating the Dilbert Principle:

```
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.
```

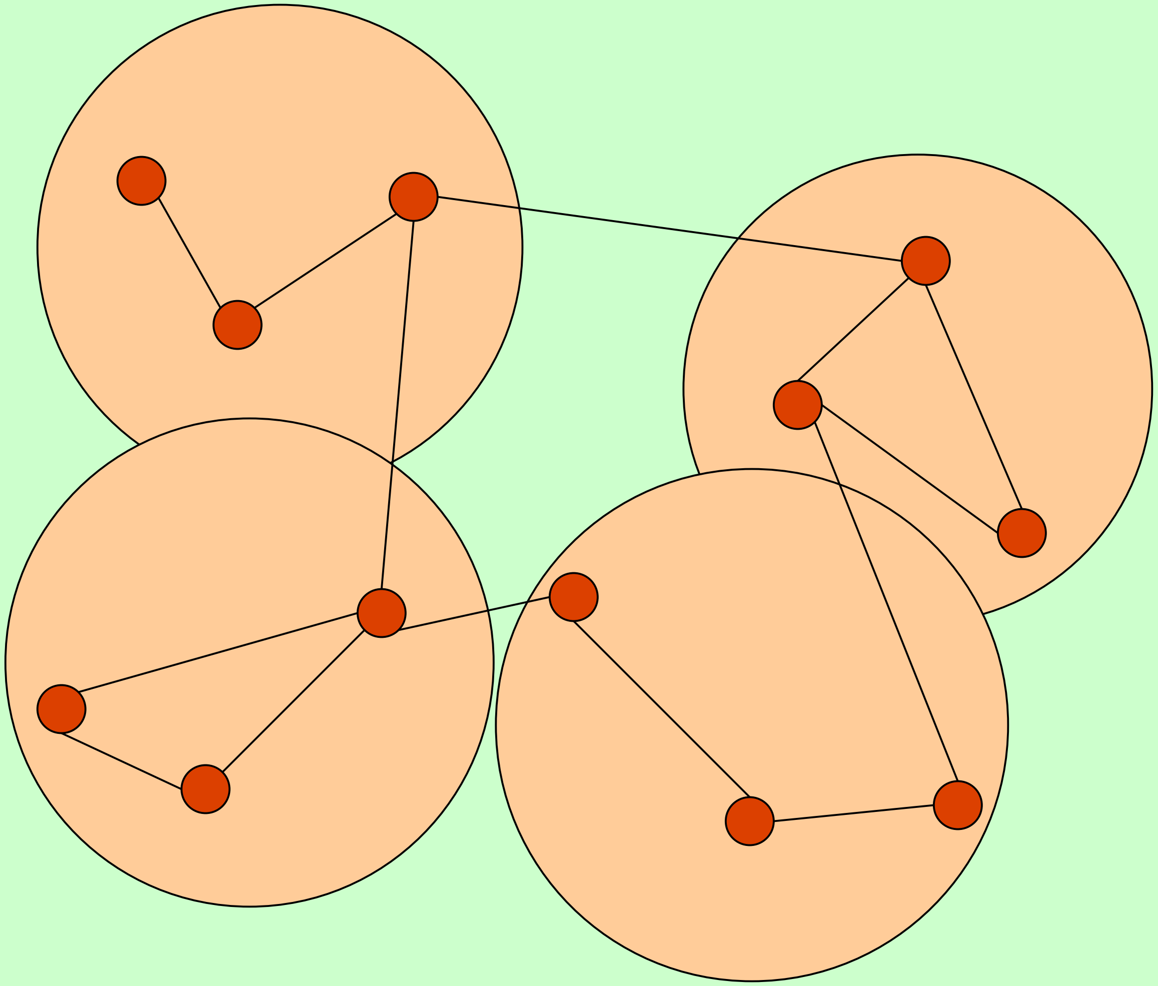
High Coupling



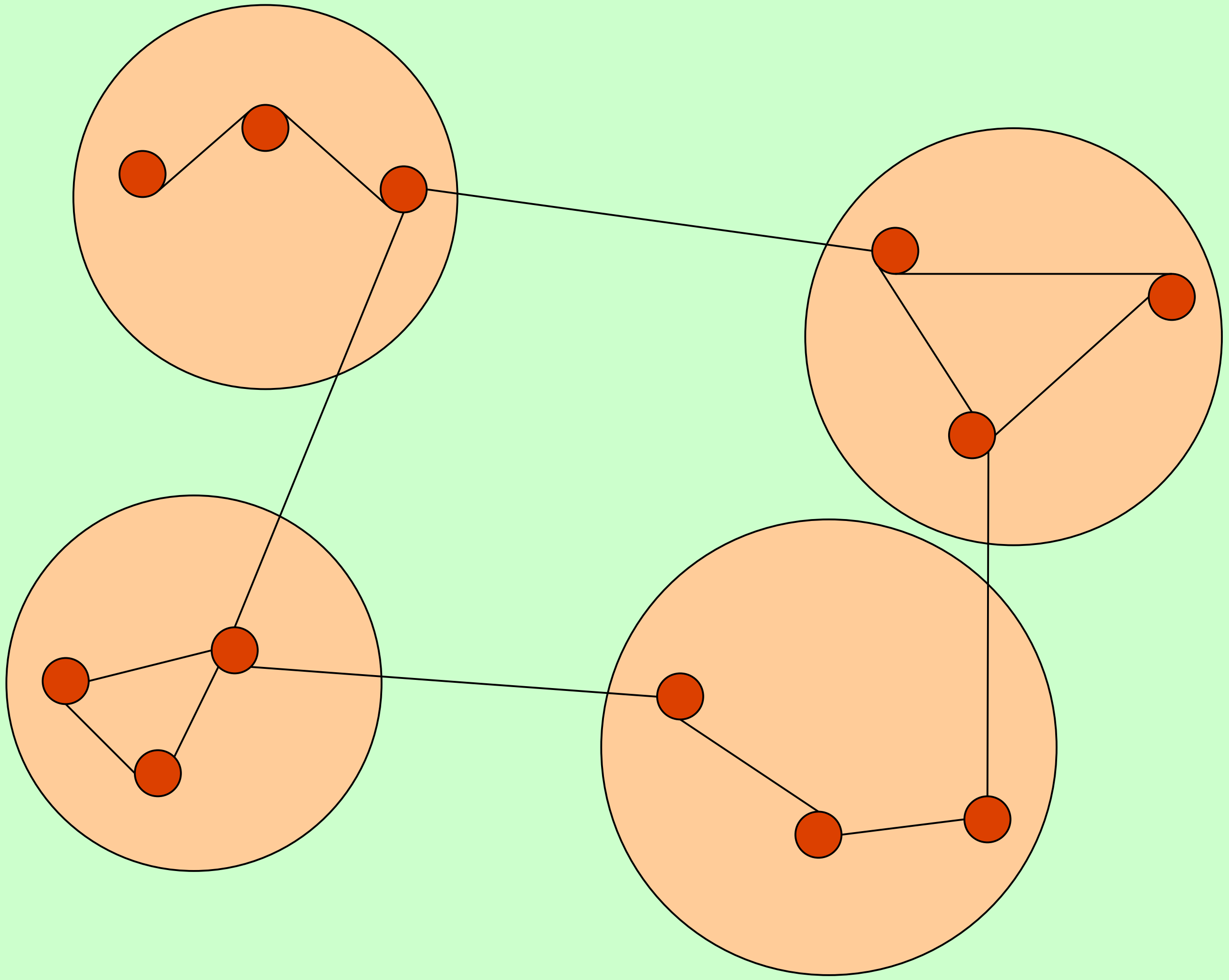
distance of components corresponds inversely to coupling

cohesion corresponds inversely to extent

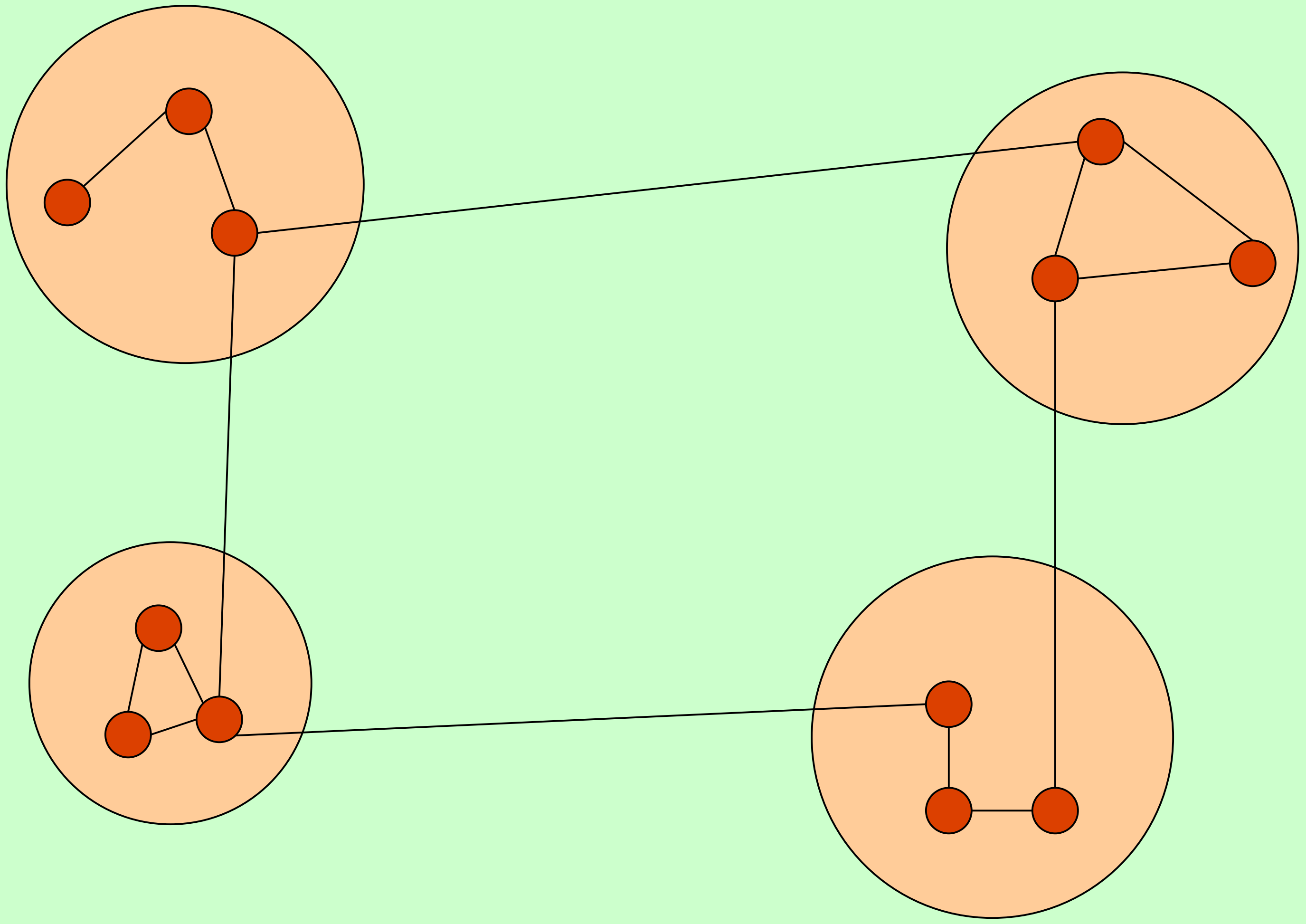
Less Coupling



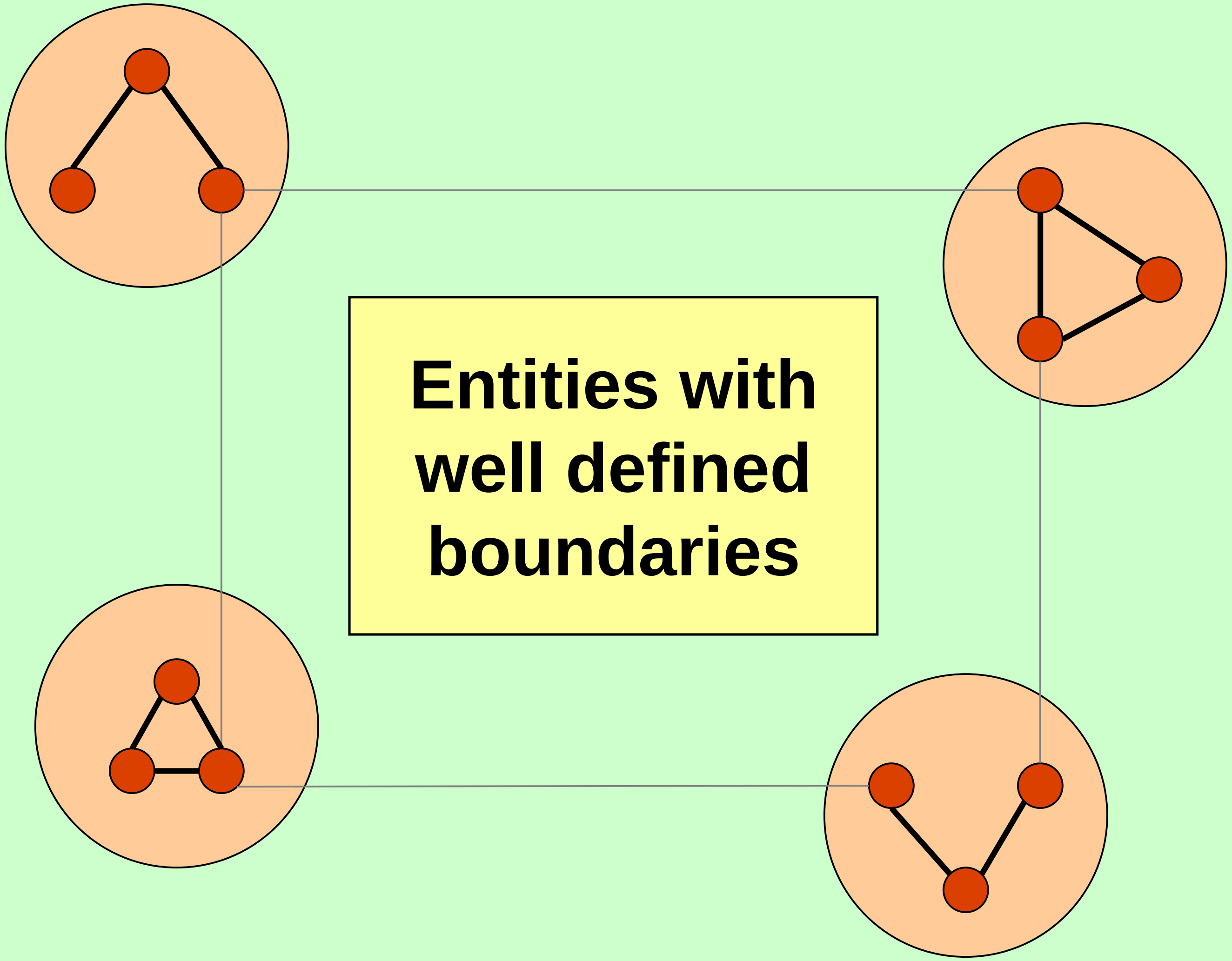
Less Coupling & Low Cohesion



Low Coupling & More Cohesion



Low Coupling & High Cohesion



Entities with well defined boundaries

Cohesion Levels

- Very low cohesion: a class is responsible for many things in different functional areas

  A class called Storage-RPC-Interface, responsible for interacting with relational databases and files and for handling remote procedure calls. Should be split into several classes.

- Low cohesion: A class has sole responsibility for a complex task in one functional area

  A class called StorageInterface, responsible for interacting with relational databases and files. The methods of the class are all related, but there are lots of them and they do too much.
  Should be split into several lightweight classes sharing the work.

Cohesion Levels

- Moderate cohesion: A class has moderate responsi-bilities in a few different areas that are logically related to the class concept, but not to each other

  A class called RDBInterface, responsible for interacting with relational databases. Its features include creating, modifying and querying databases.

- High cohesion: a class has lightweight responsi-bilities in one area and collaborates with other classes to fulfill tasks

  A class called RDBQueryInterface, partially responsible for interacting with databases. It interacts with a number of other classes related to DB access.

Low Cohesion & High Coupling?

- Classes with low cohesion are much more difficult to understand
- Changes to a class with low cohesion may affect more than one (the intended) aspect

- High coupling causes changes to propagate through the system
- Highly coupled classes are difficult to reuse as they dependent upon many other classes

Recognising Good Design

- If all methods within a class use a similar set of instance variables, the class is considered highly cohesive
- Classes with high cohesion can often be described by a simple sentence

High cohesion and low coupling is preferred

- Loosely coupled classes' communication is based on abstract interfaces rather than concrete classes
- Loosely coupled classes can be described and understood with minimal reference to other classes