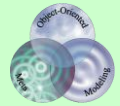
SWEN 225
Software Design

Design Principles

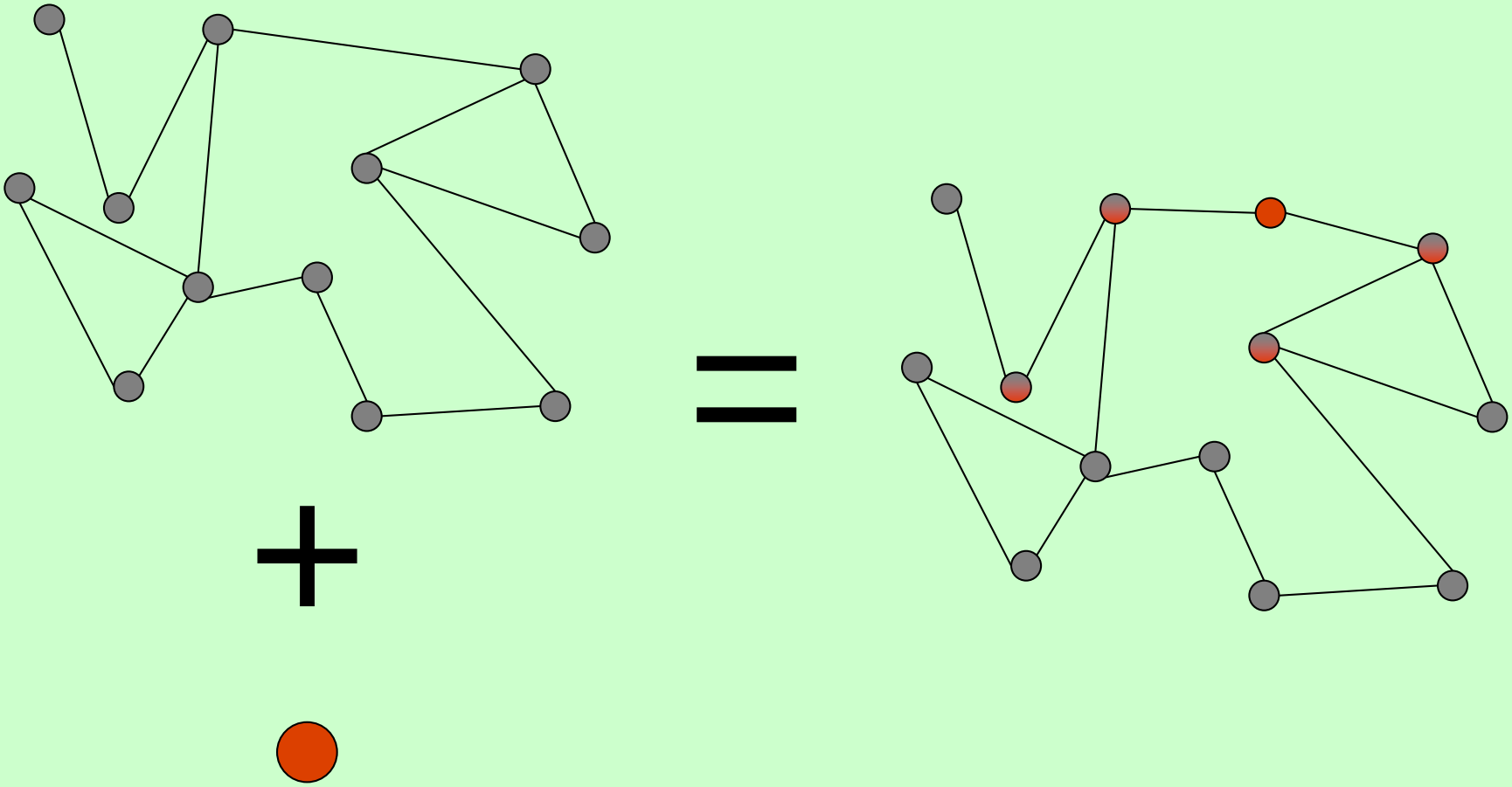
Thomas Kühne
Victoria University of Wellington
Thomas.Kuehne@ecs.vuw.ac.nz, Ext. 5443, Room Cotton 233

Extensibility

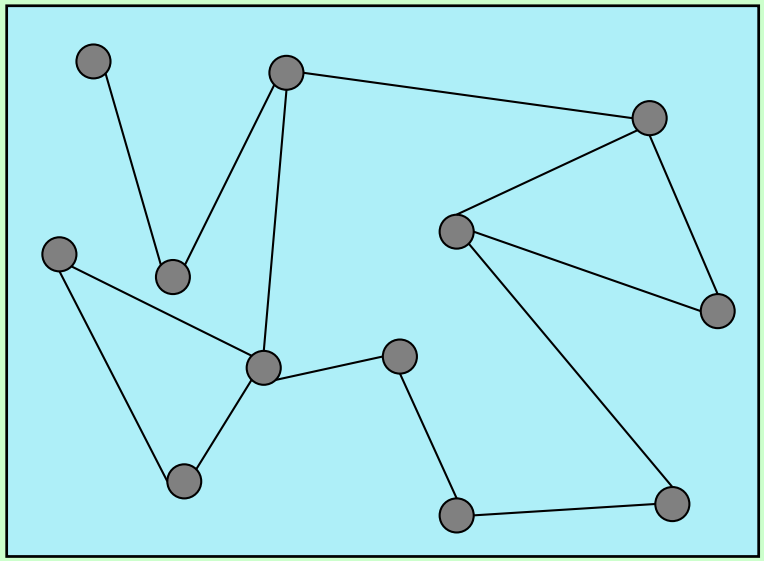
- Extensibility characterizes the ease of adapting software to changes of specification
  - …and we know the specification is going to change



Bad Extensibility

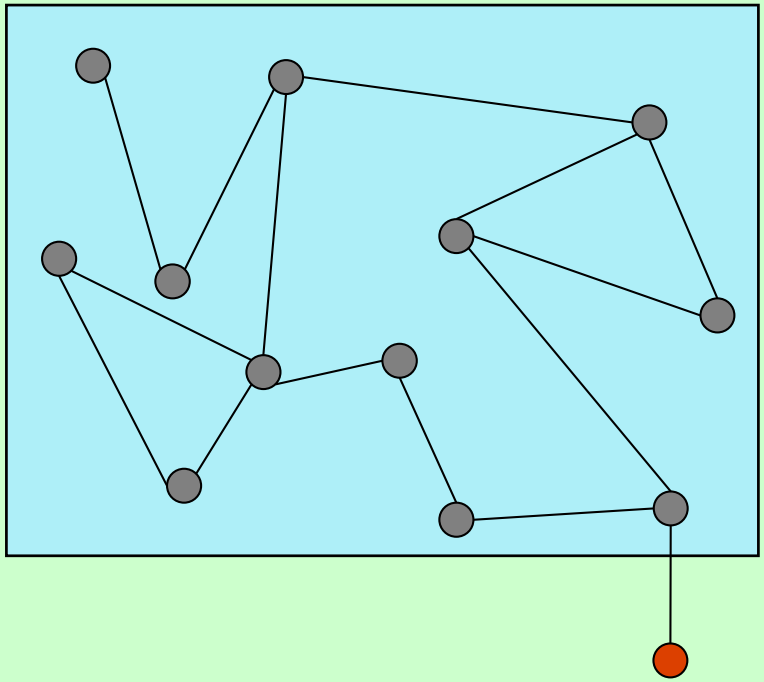


Good Extensibility



+

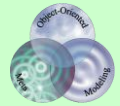
=



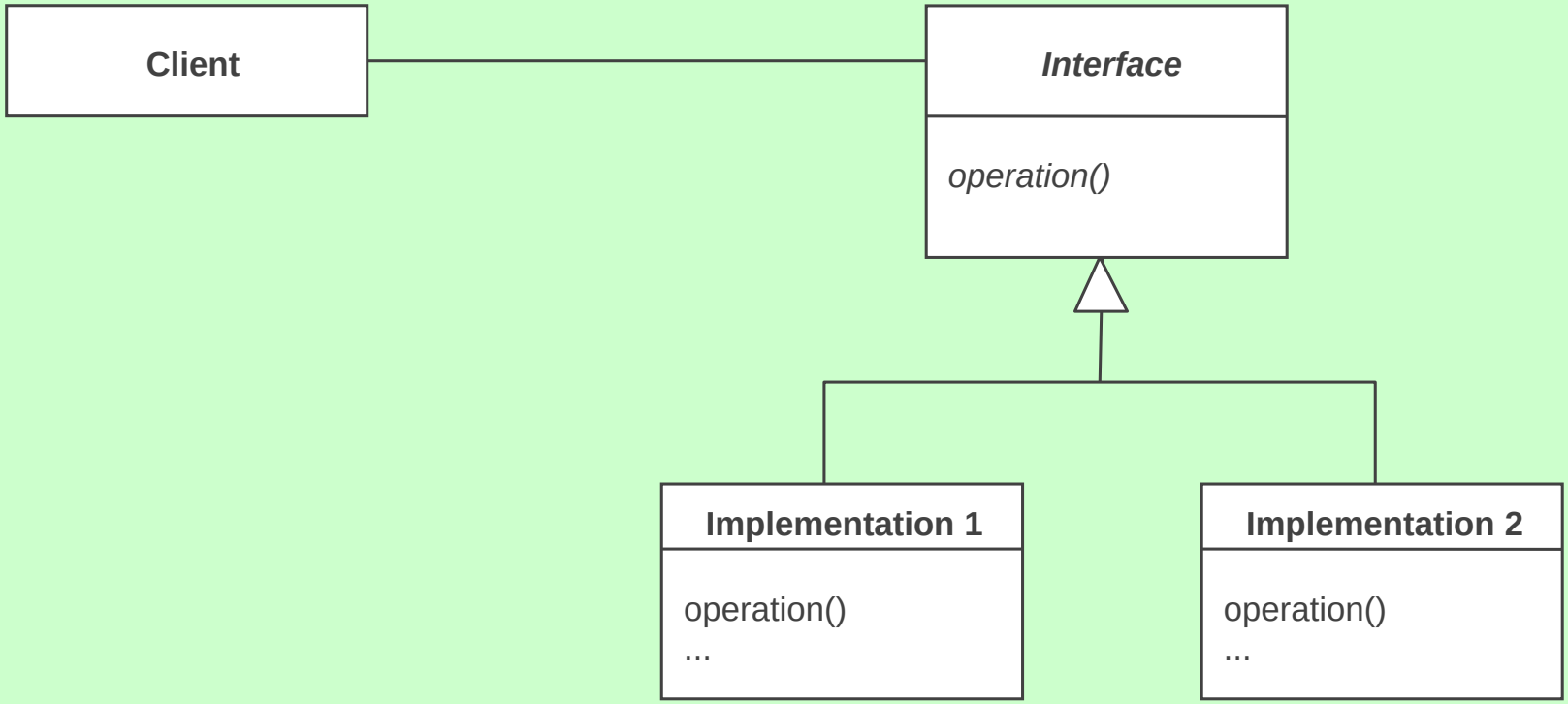
Modularity

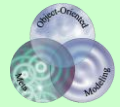
Modular systems respond better to change and are supported by the following design rule

- Don’t expose clients to implementations
  - » use interfaces
  - » use information hiding (→ encapsulation)



Interfaces

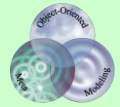

Client
Interface
operation()
Implementation 1
operation()
...
Implementation 2
operation()
...



Few Interfaces

Every module should communicate with as few others as possible.

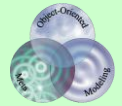
If a system is composed of *n* modules, the number of connections should remain much closer to the minimum, *n–1* **(C)**, than to the maximum, *n* (*n* – *1*) */2* **(B)**.



Few Interfaces

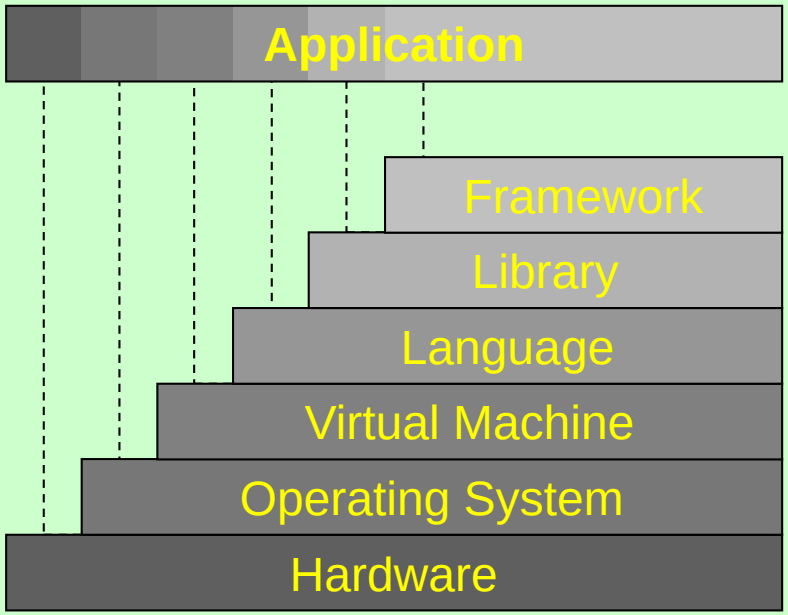
How to reduce the interface count

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

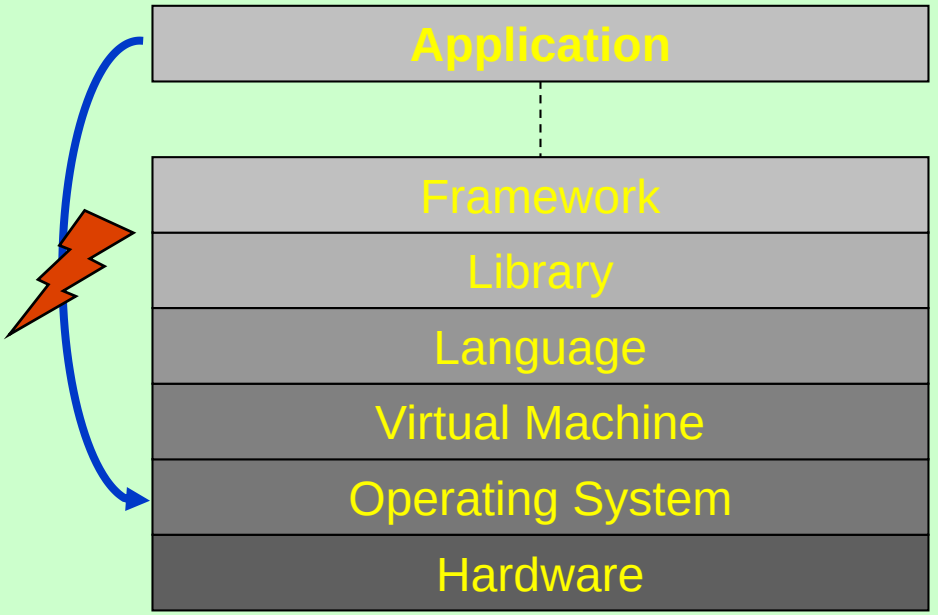


Few Interfaces

Loose Layering



Strict Layering

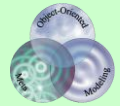


- Loose version is more efficient but also more fragile

Few Interfaces

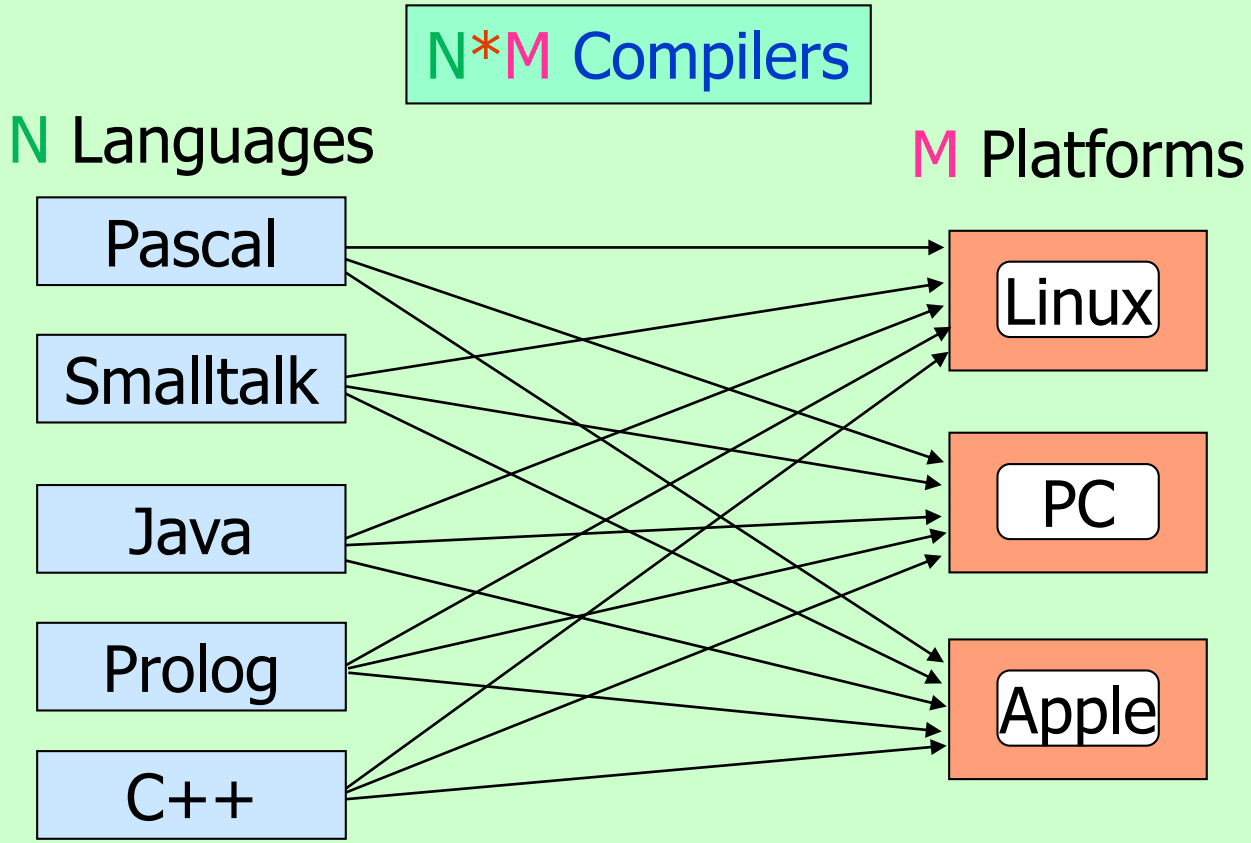
How to reduce the interface count

- 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)



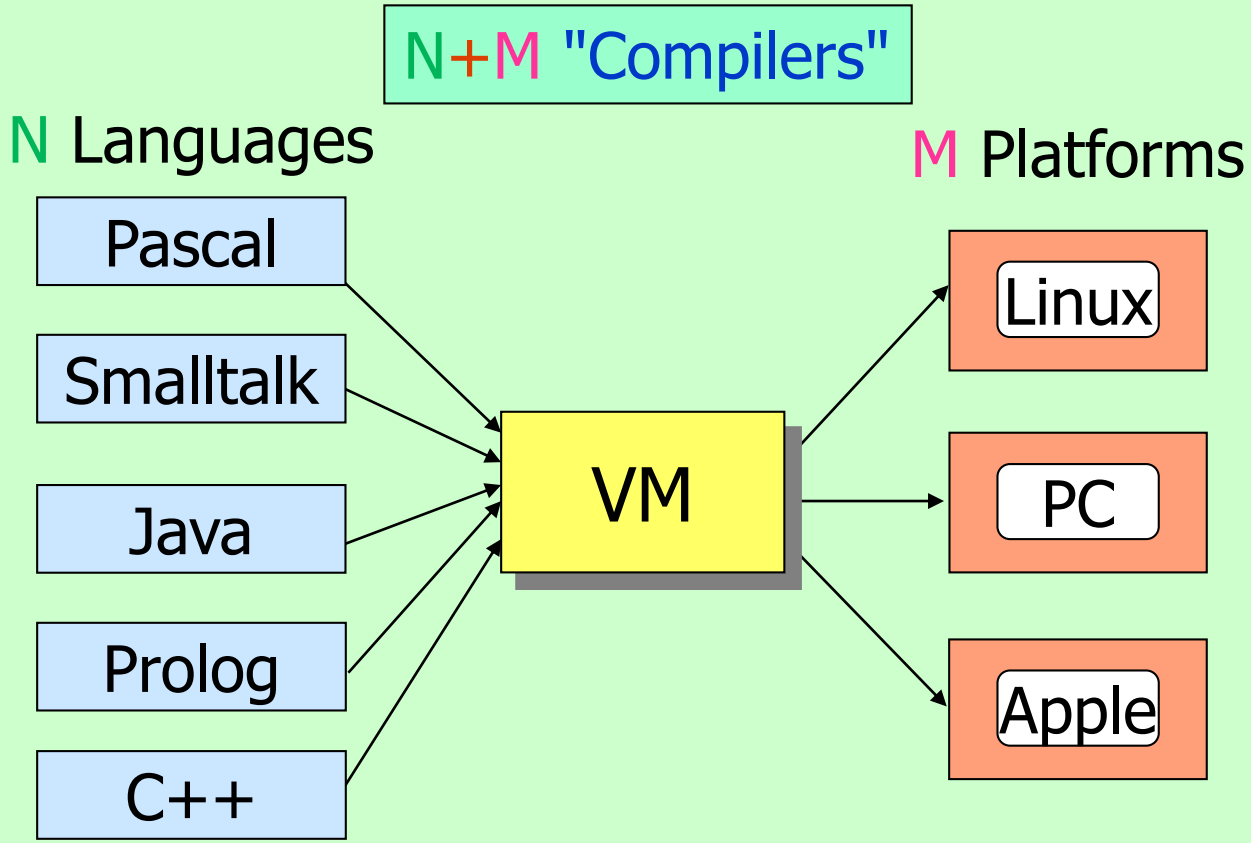
Few Interfaces

One more level of indirection



Few Interfaces

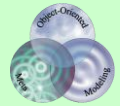
One more level of indirection



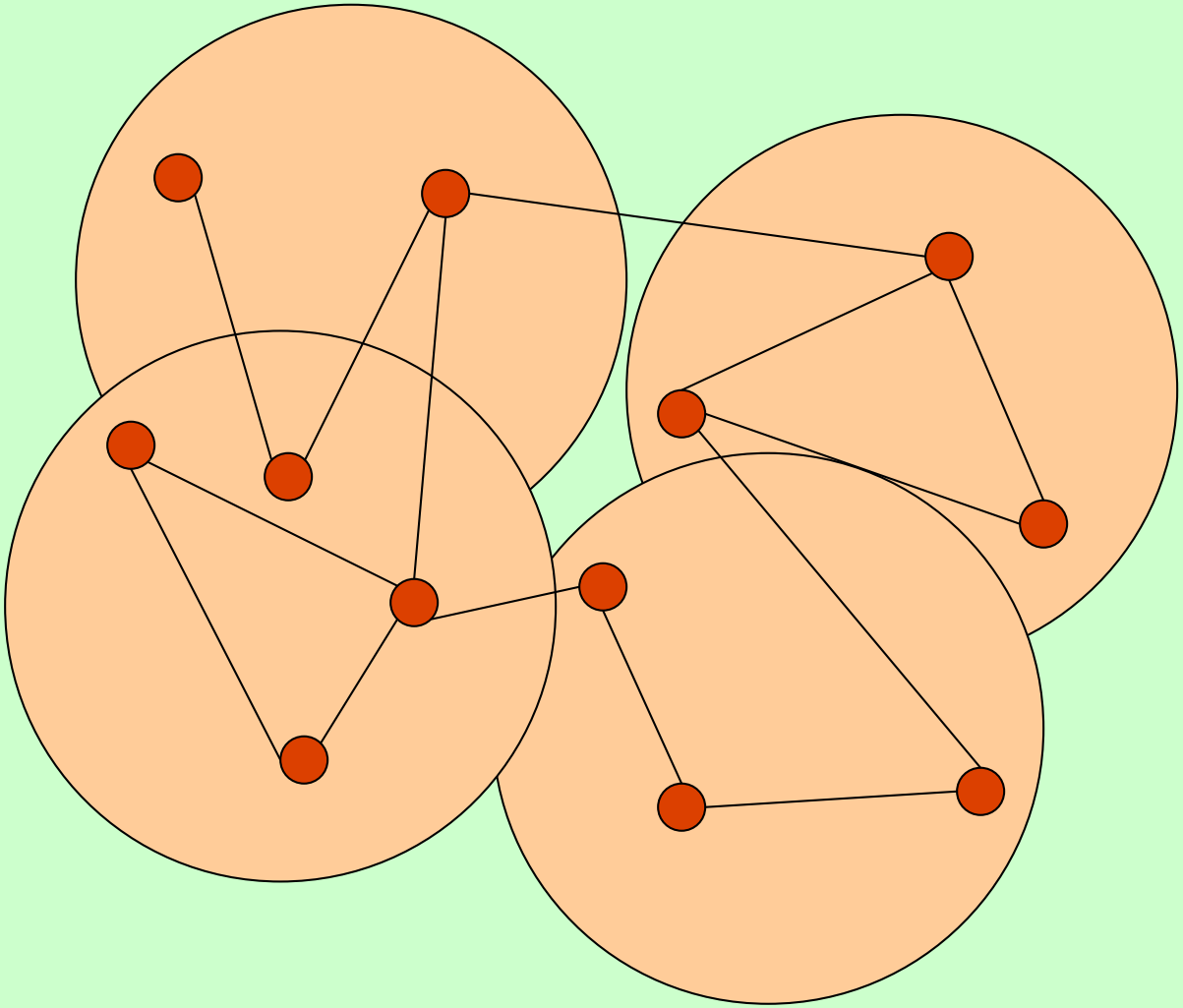
Caveat

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

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



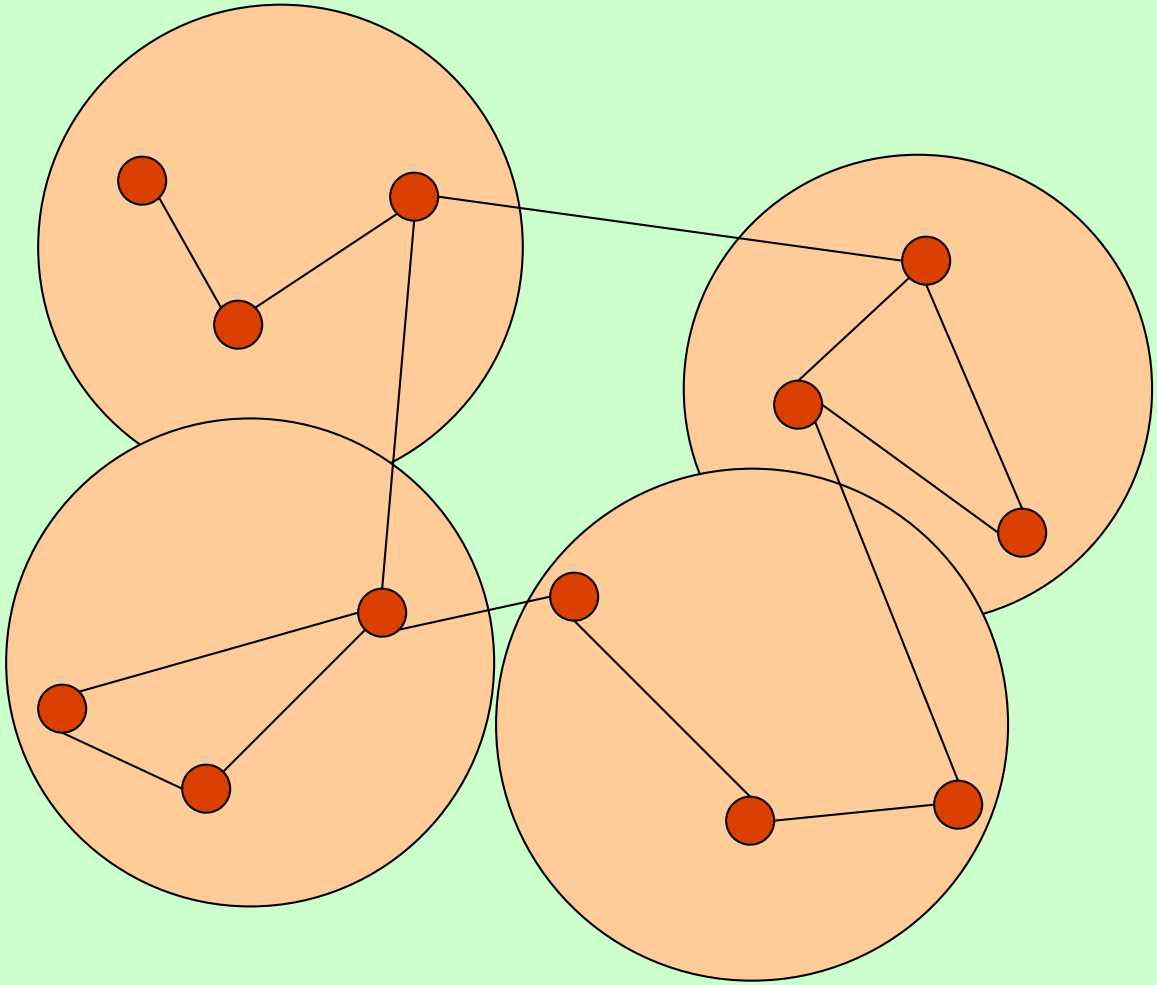
High Coupling



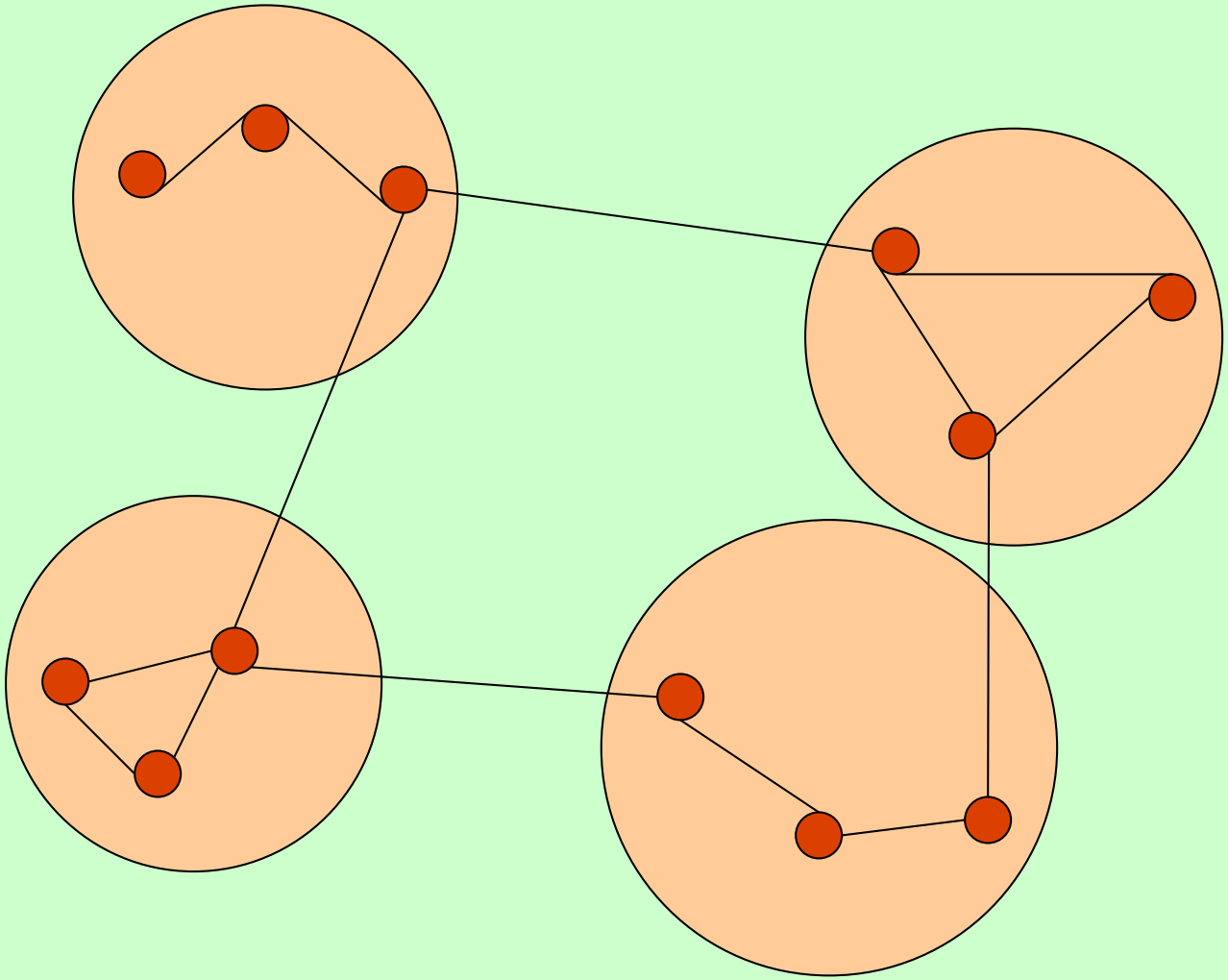
distance of components corresponds inversely to coupling

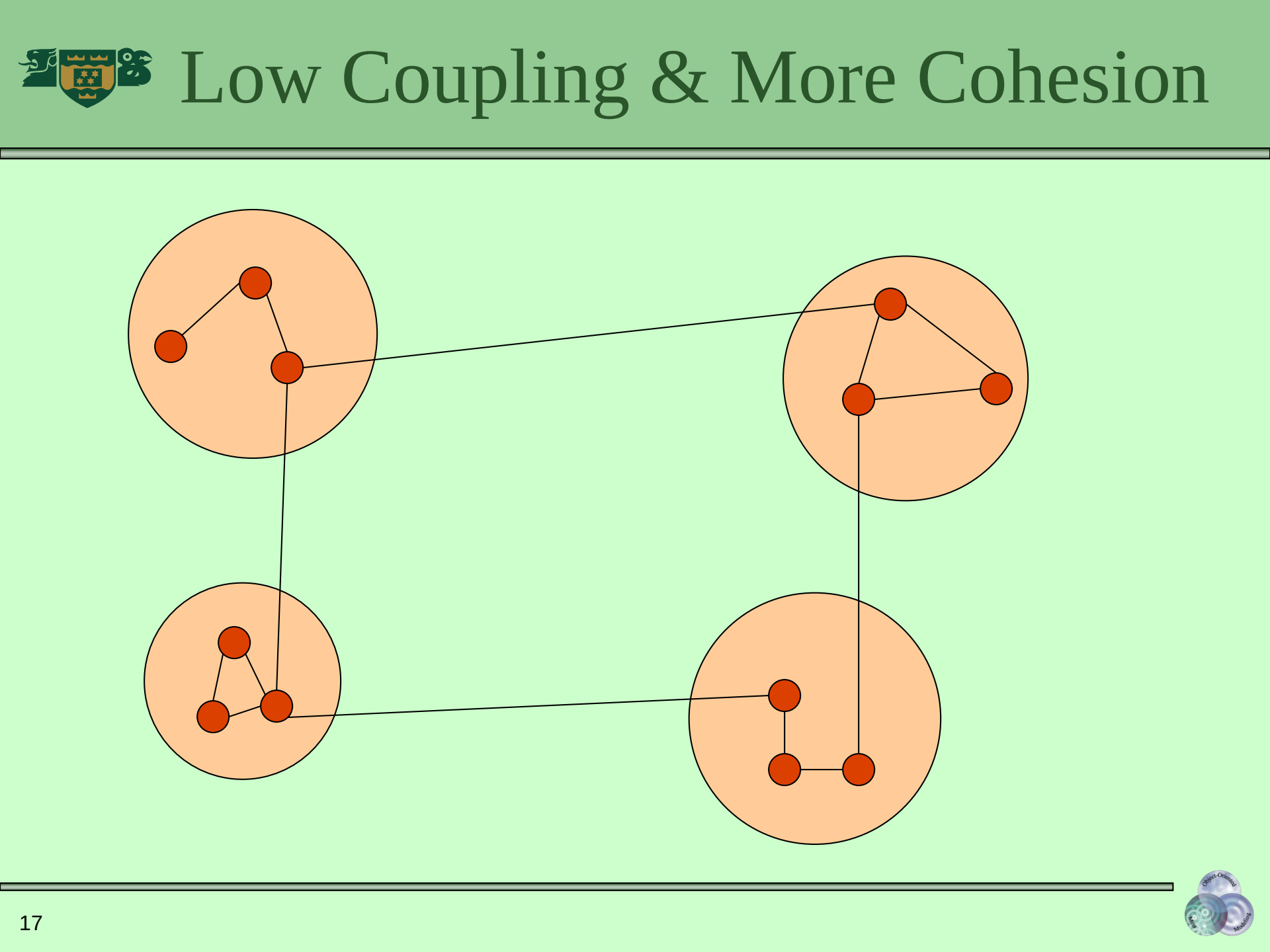
cohesion corresponds inversely to extent

Less Coupling

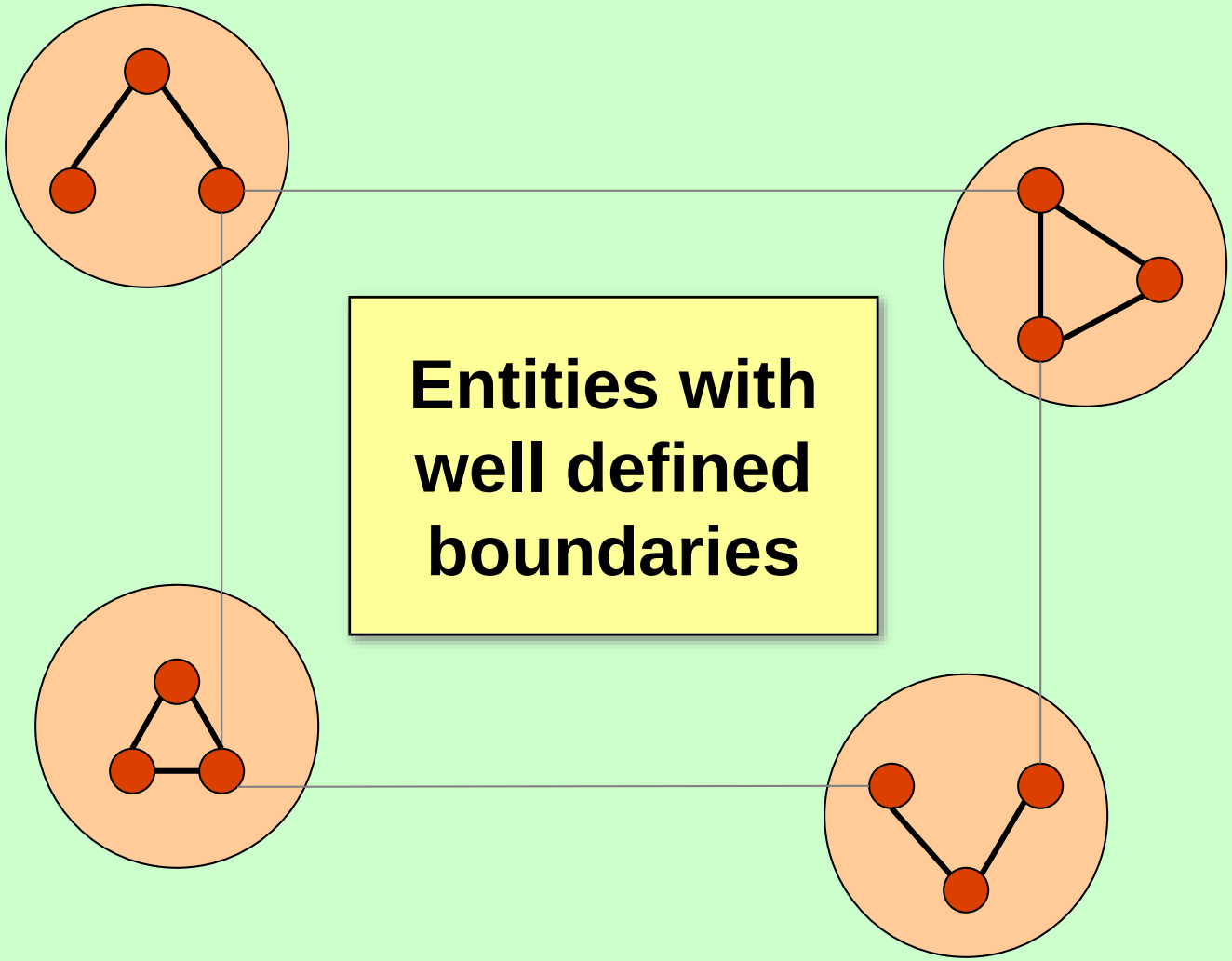


Less Coupling & Low Cohesion




Low Coupling & More Cohesion

Low Coupling & High Cohesion



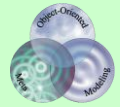
Cohesion Levels

1. 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.

2. 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.



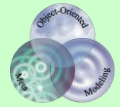
Cohesion Levels

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

❹ High cohesion: a class has lightweight responsibilities in one area and collaborates with other classes to fulfill tasks

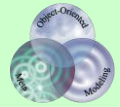
A class called RDBQueryInterface, partially responsible for interacting with databases. It interacts with a number of other classes related to DB access.



Low Cohesion & High Coupling?

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

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

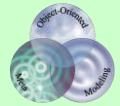


Recognising Good Design

- 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

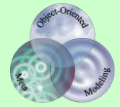
High cohesion and low coupling is preferred

- 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



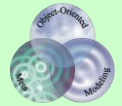
Modularity

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



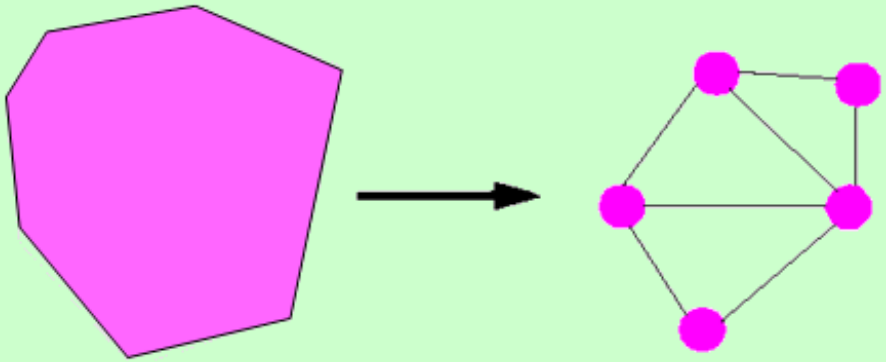
Five Fundamental Requirements

- 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.



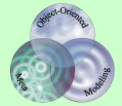
Divide
&
Conquer

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

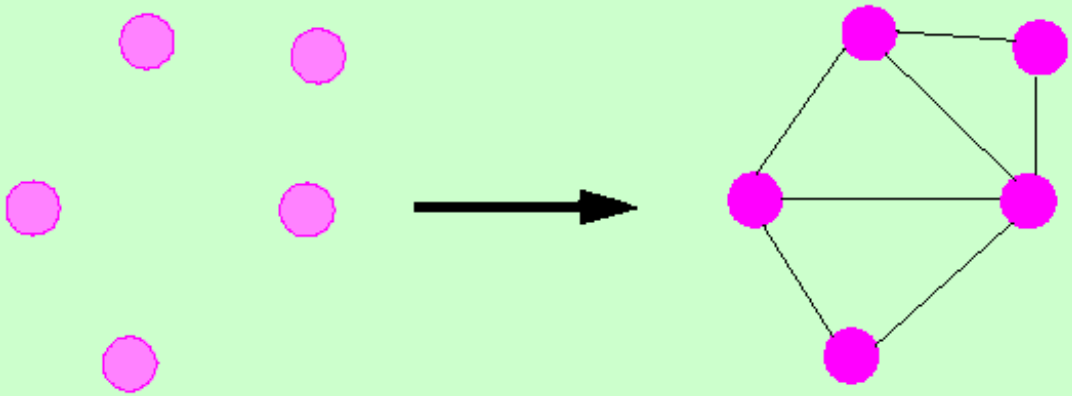
- structured analysis & design
- OO classes

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.

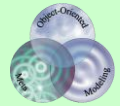


Composability

- 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



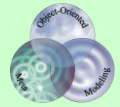
Composability

Positive Example

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

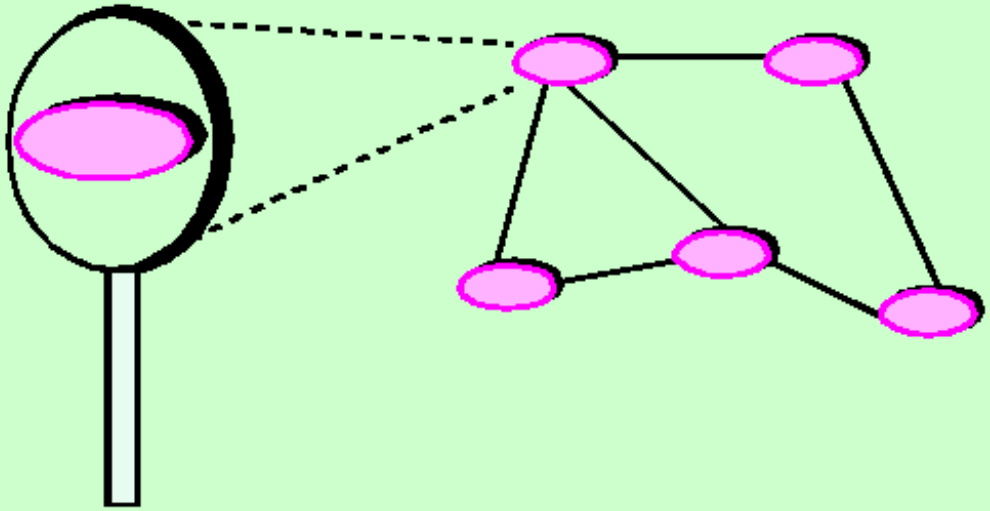
Negative Example

- 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.



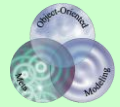
Understandability

Positive Example

- lazy initialization

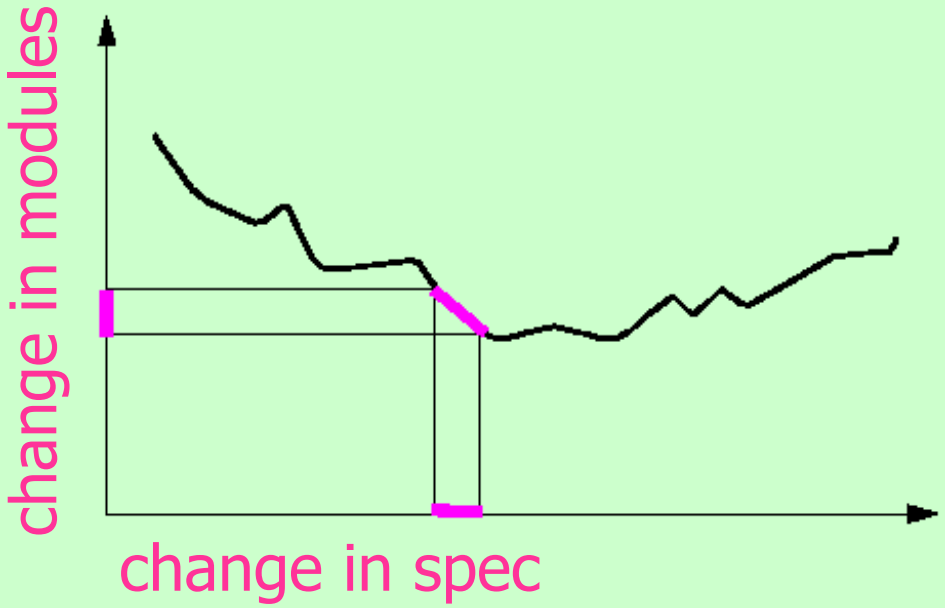
Negative Example

- 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.



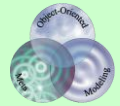
Continuity

Positive Example

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

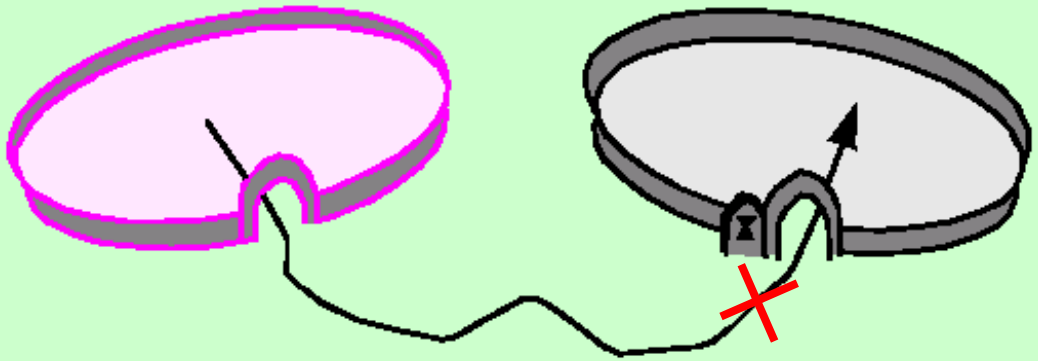
Negative Example

- 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.

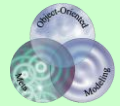


not about avoiding or correcting errors but their propagation

Protection

Negative Example

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



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