Coupling
resistance to change
understanding and fixing it

ACCU 2010
17th April 2010

Tony Barrett-Powell
tony.barrett-powell@oracle.com

Introduction

- Exploring software coupling
  - Some history
  - Categories
  - A look at cohesion
- Measuring coupling
- Looking at principles and techniques that help us with coupled software

Motivation

- I've see a lot of highly coupled code
  - Almost entirely on long lived projects
- It's been difficult to change
- Even harder to understand
- Once things reach a certain point it seems to get worse not better
  - Sometimes as soon as it is written
  - Soon a big ball of mud

What is software coupling?

- In essence it is change dependency
  - An element A tightly coupled to element B will need to change if B is changed in some way
- Coupling has no impact in the absence of change
  - However change is in the nature of software development
    - Whether requirements or technology changes
  - The more coupled a set of elements the more widespread or difficult a single change becomes

Definitions

- Thoughts about software coupling have been around for a long time
- Definitions from Structured Design movement, most of the terms still relevant in OO
- Let's look at some of the terms
  - Starting with those from Structured Design
  - To those specific to OO
  - Finishing with higher level coupling terms related to packages/components

No direct coupling

- Not really a measure of coupling, merely a stake in the ground from which to judge others
- Without some coupling there is no software - or bugs :)
- Without coupling between elements, such as classes, we have monster elements with all coupling encapsulated

Data coupling

- Considered “the primary design goal” by Myers [Myers78]
- Coupled by nothing more than homogeneous arguments/parameters
- Directly coupled and visible in the code

Stamp coupling

- Modules coupled data structure parameter
- Not all of the data structure is used by those depending on it
- Directly coupled, but harder to determine if all the structure is required
- Considered normal by [Page-Jones88]

Control coupling

- A module controls the execution of another by passing a control parameter
- Understanding of internals of another module to some extent
- Still considered normal by [Page-Jones88]

External coupling

- Modules reference the same global homogeneous data item
- Potential for unpredictable side effects
  - Not related in time or space
- No longer able to reason about function in isolation
- Testing and potential reuse becoming harder

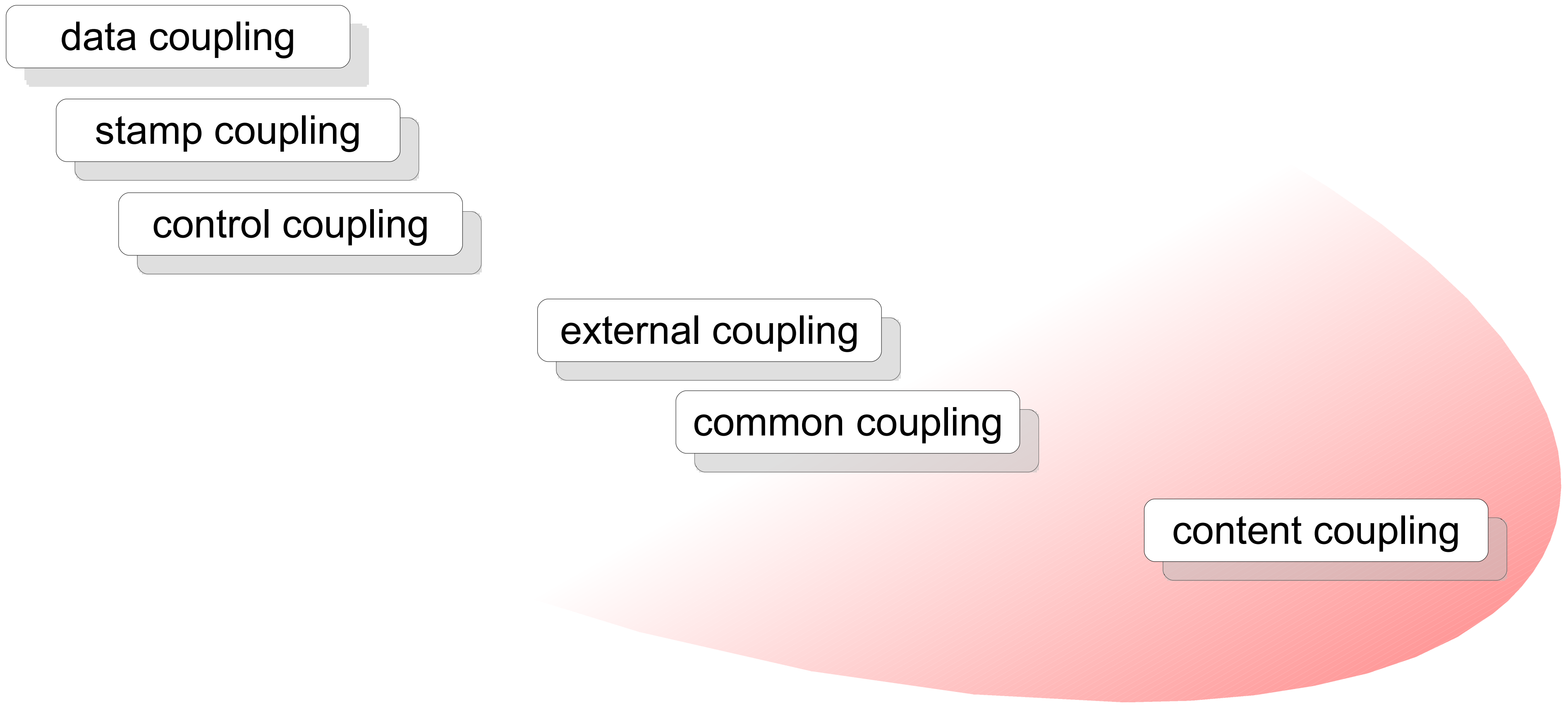
Common coupling

- From the Fortran common keyword
- Modules reference heterogeneous global data
- As with external coupling, possible unpredictable side effects
- Difficult to test in isolation
- Certainly almost impossible to reuse

Content coupling

- Module references or uses internal implementation of another
  - Whether data or functions
- Most tightly coupled
- Considered pathological by [Page-Jones88]
- Not a problem in OO languages?

Low/High coupling[1]



1. from [Page-Jones88]

Inheritance coupling

- The coupling between a child and parent
- Nominally child references parent
- Higher coupling (and a bad thing) if parent references children
- Interface inheritance vs Implementation inheritance

Temporal coupling

- Operations bundled together because they happen in a sequence
- Usually functions (or static methods)
- Usually hiding concepts

Cohesion

- From Structured design:
  “Is a measure of the strength of functional relatedness of elements within a module”
- Also know in Structured design as “Module Strength”

Cohesion (normal)

- Functional
  - elements which all contribute to the execution of one and only one problem-related task
- Sequential
  - elements which are involved in activities such that the output data from one activity serves as input to the next
- Communicational
  - elements who all use the same input or output data

Cohesion (less good)

- Procedural
  - elements are involved in different and possibly unrelated activities where control flows from one to the next
- Temporal
  - elements are involved in activities that are related in time
- Logical
  - elements contribute to activities of the same general category

Cohesion (not)

- Coincidental
  - elements have no meaningful relationship at all

More coupling

- Some coupling can only be discovered
  - Looking in source control history can show modules often change together
    - Even if not directly related in a otherwise detectable way
  - Could be a shearing layer not considered in the design
  - Or repeated code
- Use of multiple languages can lead to indirect coupling
  - Same logic resides in source of each language

Finding coupling

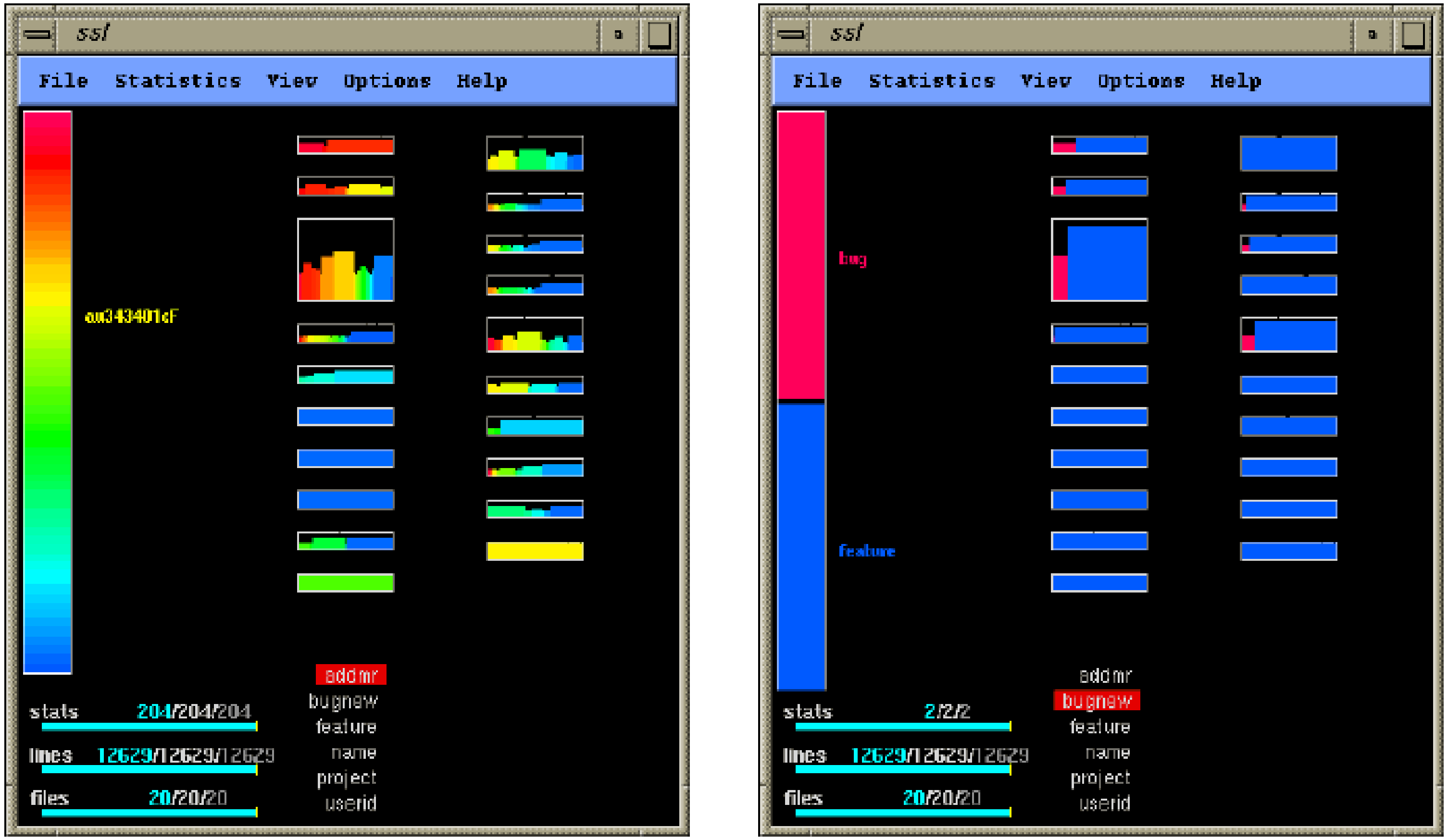


Figure 4: *Summary representation.* Each file is represented as a colored rectangle with a small plot inside, here showing the age on the left pane (blue represents old code and red represents new code) and bug-fixing code on the right pane.

Why is coupling bad?

- It isn't but it can be
  - Without some coupling we have no software
  - But too much and we have a big ball of mud
- It can be a symptom of
  - Poor design
  - Good design that has degraded
  - Changes in dimensions not supported by the design

Why is coupling bad?

- Too much coupling results in
  - difficulty in understanding how the code achieves a particular function
  - inability to identify where to make a change
  - the likelihood that a change will break some apparently unrelated function

Measuring coupling

- Some simple google-fu can find many ways to measure coupling
  - One of the simplest is method coupling count [feathers]
  - Others far more involved with weightings for different types of coupling
  - Support from static analysis tool for many coupling measures
- Generally measured at module/class level

Measuring coupling

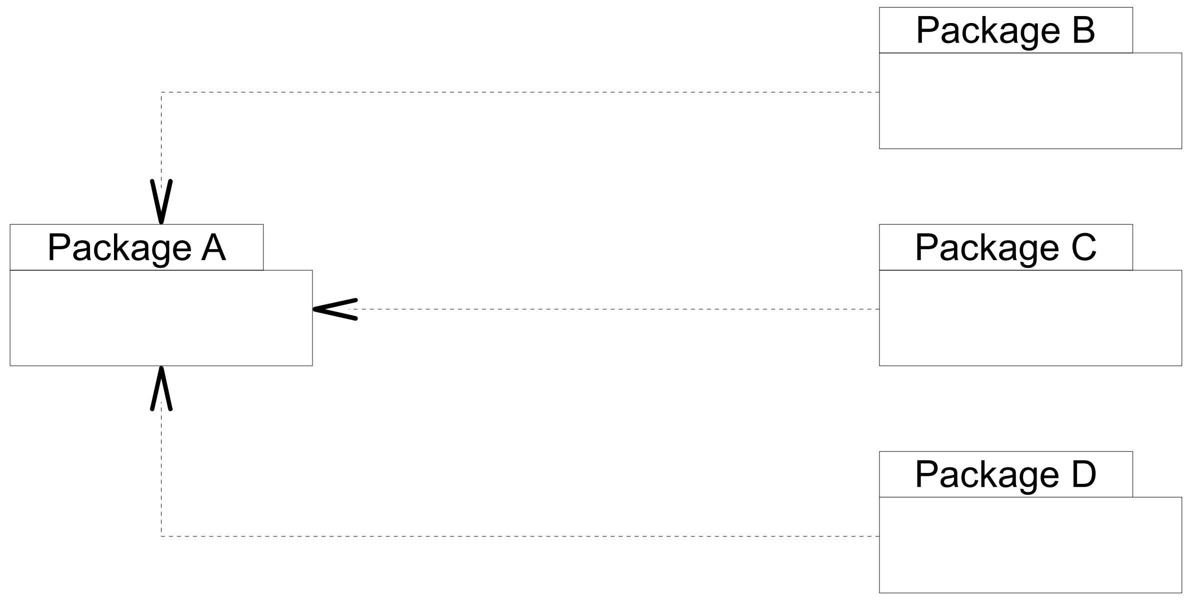
- Method coupling count
  - Number of values passed in and out of a method
  - Used to evaluate refactoring safety
- Class coupling
  - An ubiquitous metric:

$$C = 1 - \frac{1}{(d_i + 2 \times c_i + d_o + 2 \times c_o + g_d + 2 \times g_c + w + r)}$$

where: w = fan out, r = fan in

Afferent coupling

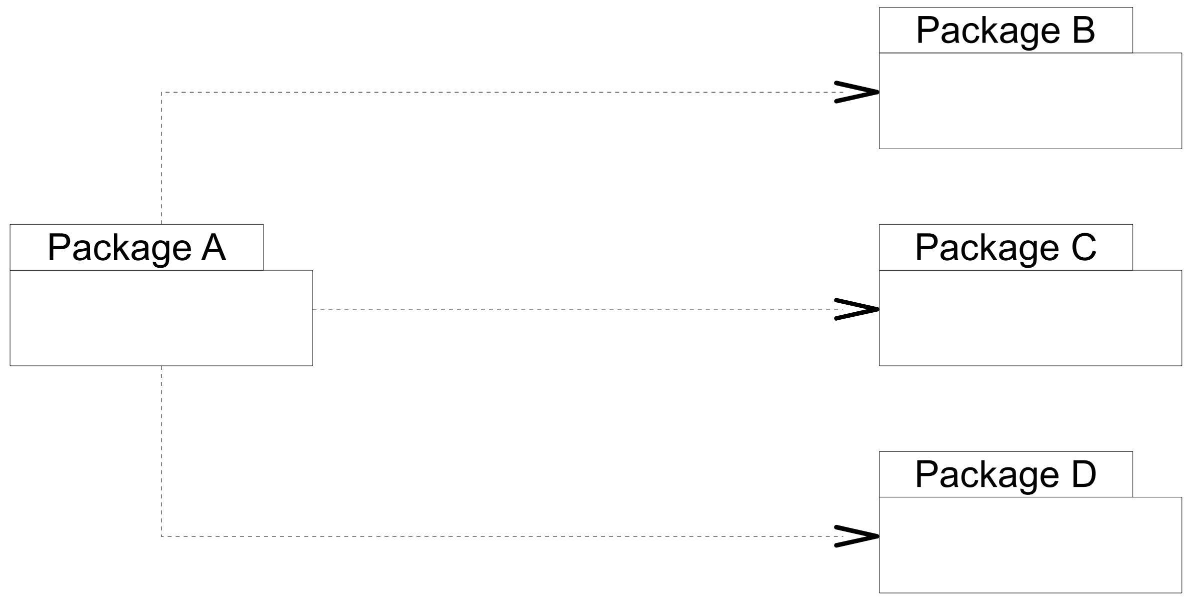
- The number of classes outside a package that depend on classes in a package



- A measure of the impact of changing the package

Efferent coupling

- The number of classes inside a package that reference classes outside a package



- A measure of the impact of changes outside the package

Instability

- How susceptible a package is to change:

  $I = C_e / C_a + C_e$

  where:
  - I: Instability, range [0 (max. stable),1 (min. stable)]
  - C_a: Afferent coupling
  - C_e: Efferent coupling

Abstractness

- How likely a package is to change and therefore impact others:

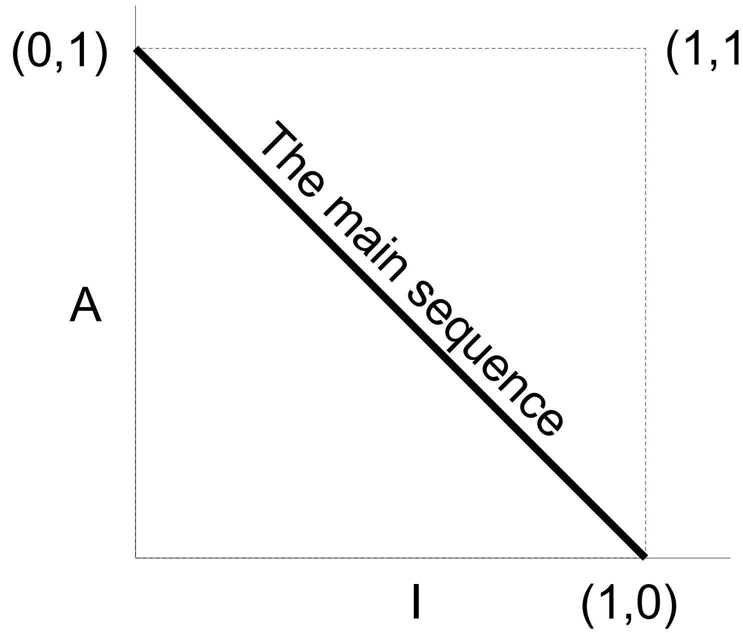
  $A = N_a / N_c$

  where:

  N_c: the number of classes in a package
  N_a: the number of abstract classes in a package
  A: abstractness, range [0 (concrete), 1 (abstract)]

Package balance

- The Abstract-Instability graph:
- Distance from the main sequence indicates how imbalanced a package is
- An imbalanced package will make the design more difficult to change
  - Consider the impact of a stable and concrete package (0,0)



Don't eliminate all coupling

- We should never try to reduce coupling to standard library classes
  - For example `java.util.String`
  - As stable as it is possible to be
- Some fundamental concepts or dependencies that will never change
  - protection against changes here will only complicate the design making it more difficult to comprehend
  - consider targeting the likely areas of change first

Some examples

- The following example are simple coding or architectural problems I've seen across more than one project
- Whilst not unusual (or maybe it's the code I've worked on) they demonstrate good design principles which have been ignored (or forgotten in the heat of the moment)
- These all are concrete problems (though presented generically) that cause a design to be hard to change

Problem: reach through

- In IDE driven development (menu based programming?) and in javascript code
- The use of long chains of calls (or members): getTom().getEmily().getHarry().doSomething()
- Makes change very hard
  - especially in javascript
- Obviously, this style of coding is very fragile in the face of change

Principle: encapsulation

- aka: Information hiding
- Reach through is “breaking and entering” through the interface of the class(es)
- The client shouldn't have knowledge of how (and with what) the class has been implemented
- Excessive getters (and setters)

Principle: law of demeter

- An attempt to minimize coupling between functions
- States that “any method of an object should call only methods belonging to:
  - itself
  - any parameters that were passed in
  - any objects it created
  - any directly held component objects”
- Specifically rules out calling methods of objects obtained from the above

Principle: tell don't ask

- Procedural code gets information then makes decisions. Object-oriented code tells objects to do things. [Sharp97]
- Responsibility about action(s) to take based on the state of an object lies with the object and not the caller
- Therefore, let the object figure out what to do not the client(s)

Problem:
type based switch statements

- There is a hierarchy of types but for some operations the processing is in a method outside the hierarchy
- The processing is usually controlled by a switch statement, based on type of element
- There is (more often than not) a multiplicity of such switch statements
- Additions of new types (or almost any change) to a family of types ripple through these switch statements

Principle: DRY

- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system [Hunt00]
- If there are multiple places to make a single change:
  - a programmer has to remember (or discover) where to make this change
  - bugs will keep coming back in different situations
  - concepts are more difficult to comprehend as more code is needed to be examined

Principle: single choice

- Whenever a software system must support a set of alternatives, one and only one module in the system should know about their exhaustive list [Meyer97]
- Allows addition of variants without ripple effect
- Creational patterns [GoF95] support this principle

Problem: repeated rules

- Usually found where a mix of languages exist in a project, but not always
- Business rules tend to gravitate outwards and inwards during development of a system
  - repeated in the languages at the periphery of the system
  - for example: UI and persistence

Technique: code generation

- Define rules in a code neutral form which can be used to create the required implementations
  - Preferably text based
  - Parsing is generally the most difficult part
  - Doesn't have to be complicated
  - Could be no more than a shell script

Problem: unwanted inheritance

- aka: fat interface
- A class inherits from another, but for some methods does nothing, asserts or throws a not implemented exception
  - Removes functionality from the base class
- An eroded design
- A likely source of bugs for the future

Principle: liskov substitutability

- if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program
- Interface inheritance (is-a) not implementation inheritance
- If broken, suggests an undesirable coupling of concepts in the parent type

Problem: values that persist

- Values used to represent the concepts of the problem space directly use persistence or serialization implementations
- Change to mechanisms difficult
- Writing unit tests for these values non-trivial

Principle: single responsibility (cohesion)

- Forces that cause a module, or class, to change [Martin03]
- Each responsibility of a class is a dimension of change
- From example, the domain concept may change separately from the mechanism used for persistence
  - different reasons for change
  - different responsibilities
- Common-Closure principle for packages

Technique:
TDD

- Test driven development (Test first)
- Drives a uncoupled design
- Ensures the separation of concepts and allows responsibilities to be assigned to these concepts more readily
- Unit testing is a force for good in a design, TDD ensures this force is applied immediately
- Integration and System testing do not have the same impact on design

Principle: open-closed

- “Modules should be both open and closed” [Meyer97]
  - Open to extension
  - Closed for modification
- Abstraction
  - “it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviours” [Martin03]

Principle: open-closed

- Interfaced based design
  - Interfaces (or abstract classes) are stable
    - certainly more than implementations
  - Are both open and closed
  - Dependency on interfaces (or abstract classes) reduces coupling (or reasons for change)

Problem: layered architecture

- High level modules depend on lower level modules, depend on even lower level modules
- A change in a lower level module has a direct and negative impact on the higher level modules
  - policies (business rules) impacted by low level changes

Principle: dependency inversion

- aka: Hollywood: don't call us, we'll call you
- More simply: “depend on abstractions”
- Inverting the ownership of interfaces
  - It is not the provider who owns the interface but the client
  - the service provides the implementation to meet the client's need
- Knowledge of implementation captured by a creator (factory, etc)

Principle: selfish object

- Instead of focusing on what an object can use (or be given) focus on what it wants
- Higher impact on coupling than abstracting interfaces from implementations
- Consequences on architecture are to increase PfA, dependency inversion, separation

Principle: stable dependency

- Instable packages should depend on stable packages
  - Using instability metric to determine this
- Efferent coupling has a negative impact on stability
  - more axes of change
- The more afferent coupling the more stable a package
  - the harder it is to change

Principle: stable/abstraction

- A package should be as abstract as it is stable
  - Stability means change has a high impact on other packages
  - Concrete classes are more likely to change than abstract classes (or interfaces)
- Therefore aim for
  - Abstract and stable
  - Concrete and instable

Problem: über facade

- An interface or facade of an entire part of the system
- Mix of many concepts
  - a tendency to be used together
  - driven from the implementation
  - a kind of stamp coupling?
- Difficult to provide a mock
- Clients coupled to unneeded concepts

Principle: small interfaces

- Clients should only be forced to depend on what they need
- A combining of concepts in a single interface increases coupling
  - Impact of change of one concept impacts all clients of the interface, even if the concept is not used
- Separation of concepts for separate clients increases cohesion and reductions coupling

Wrap up

- Too much coupling has a negative effect on the ability to change software
- Simple design/coding problems are the visible evidence of the violation of good design principles
- Reducing coupling is a good thing, but only up to a certain point beyond which the design suffers for no real benefit

References

[Ball96] Ball, Thomas. Eick, Stephen G. *Software Visualization in the Large.* IEEE Computer Society Press, 1996

[Eckstein04] Eckstein, Jutta. *Agile Software Development in the large – Diving Into the Deep.* Dorset House Publishing, 2004.

[Evans04] Evans, Eric. *Domain Driven Design.* Addison-Wesley, 2004

[Feathers05] Feathers, Michael C. *Working Effectively with Legacy Code.* Prentice Hall, 2005.

[GoF95] Gamma, Eric.Helm, Richard. Johnson, Ralph. Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley 1995.

[Henney07] Henney, Kevlin. *The Selfish Object.* http://www.software-architect.co.uk/slides/sa07-KevlinHenney-Selfish_Object.pdf

[Hunt00] Hunt, Andrew. Thomas, David. *The Pragmatic Programmer.* Addison-Wesley, 2000.

References

[Larman02] Larman, Craig. *Applying UML and Patterns (second edition).* Prentice Hall, 2002.

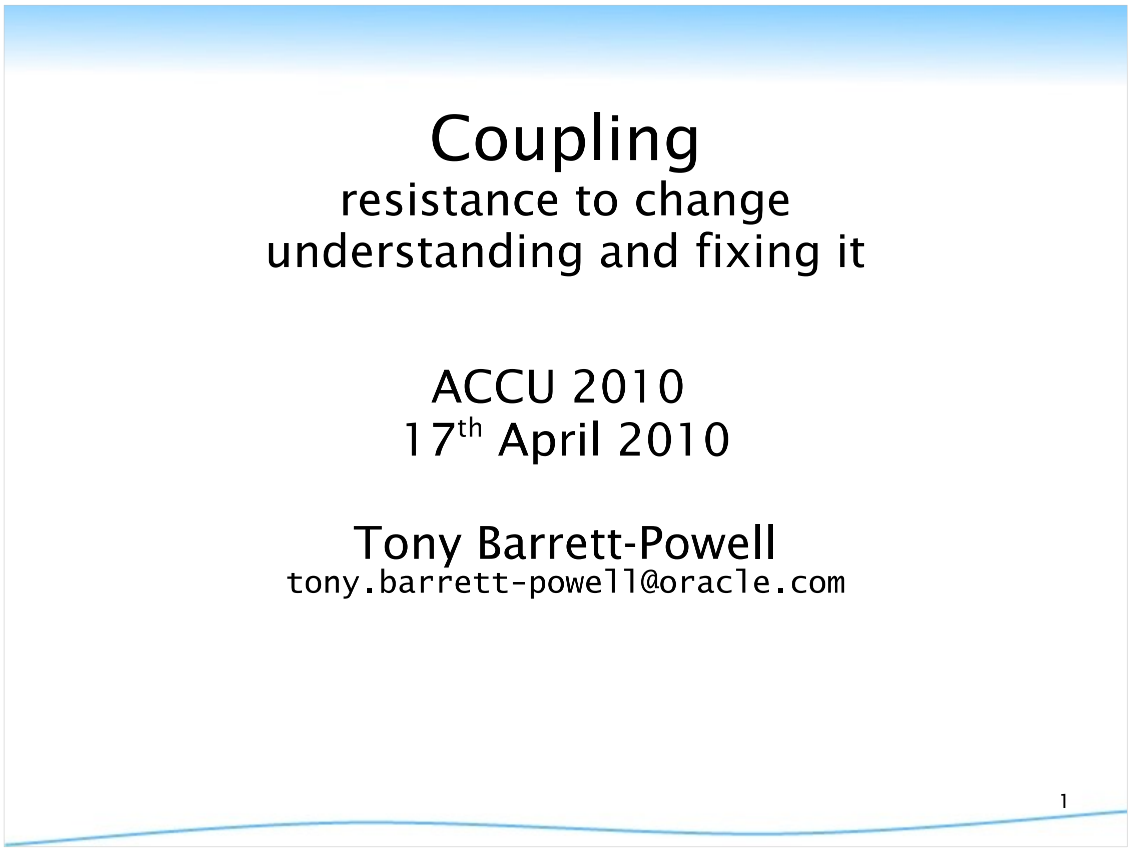
[Martin03] Martin, Robert C. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2003.

[Meyer97] Meyer, Bertrand. *Object Orientated Software Construction (second edition).* Prentice Hall, 1997.

[Myers78] Myers, Glenford J. *Composite/Structured Design.* Van Nostrand Reinhold Company, 1978.

[Page-Jones88] Page-Jones, Meilir, *The Practical Guide to Structured System Design (second edition).* Prentice Hall, 1998.

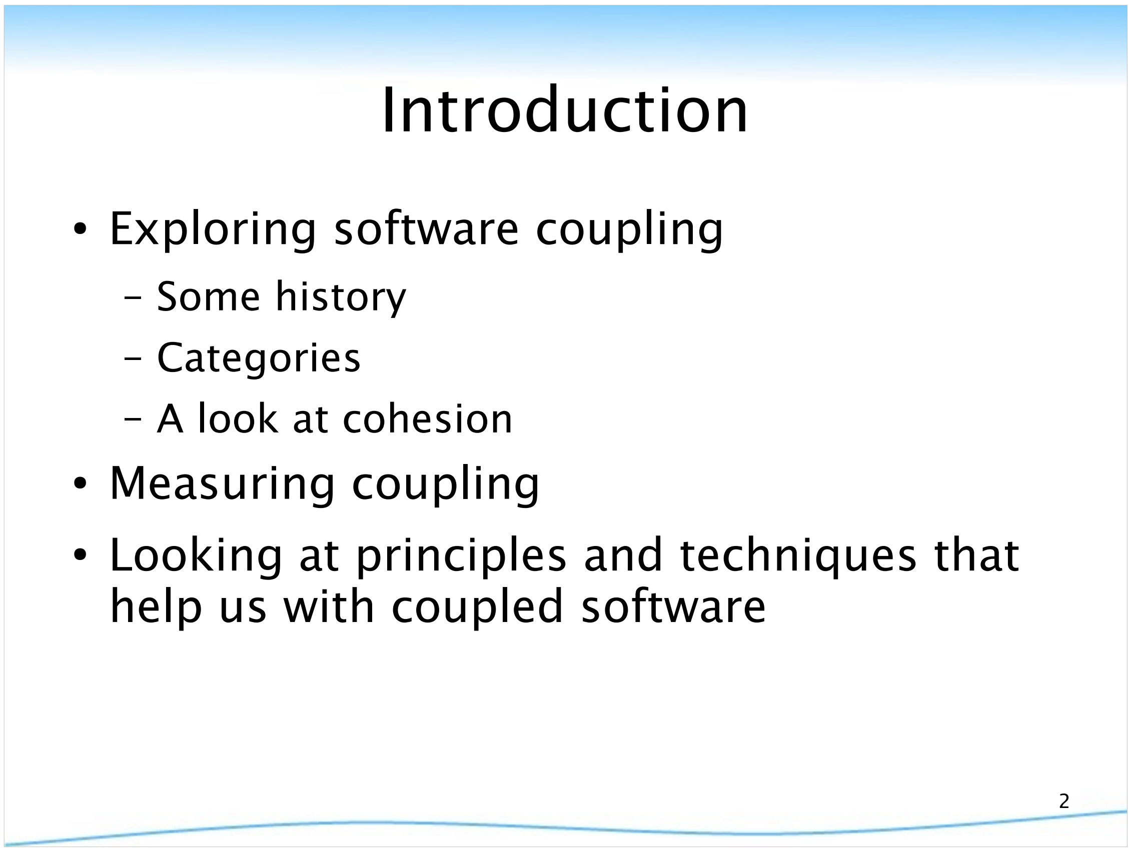
[Sharp97] Sharp, A. *Smalltalk By Example* . McGraw-Hill, 1997.



Coupling
resistance to change
understanding and fixing it

ACCU 2010
17th April 2010

Tony Barrett-Powell
`tony.barrett-powell@oracle.com`



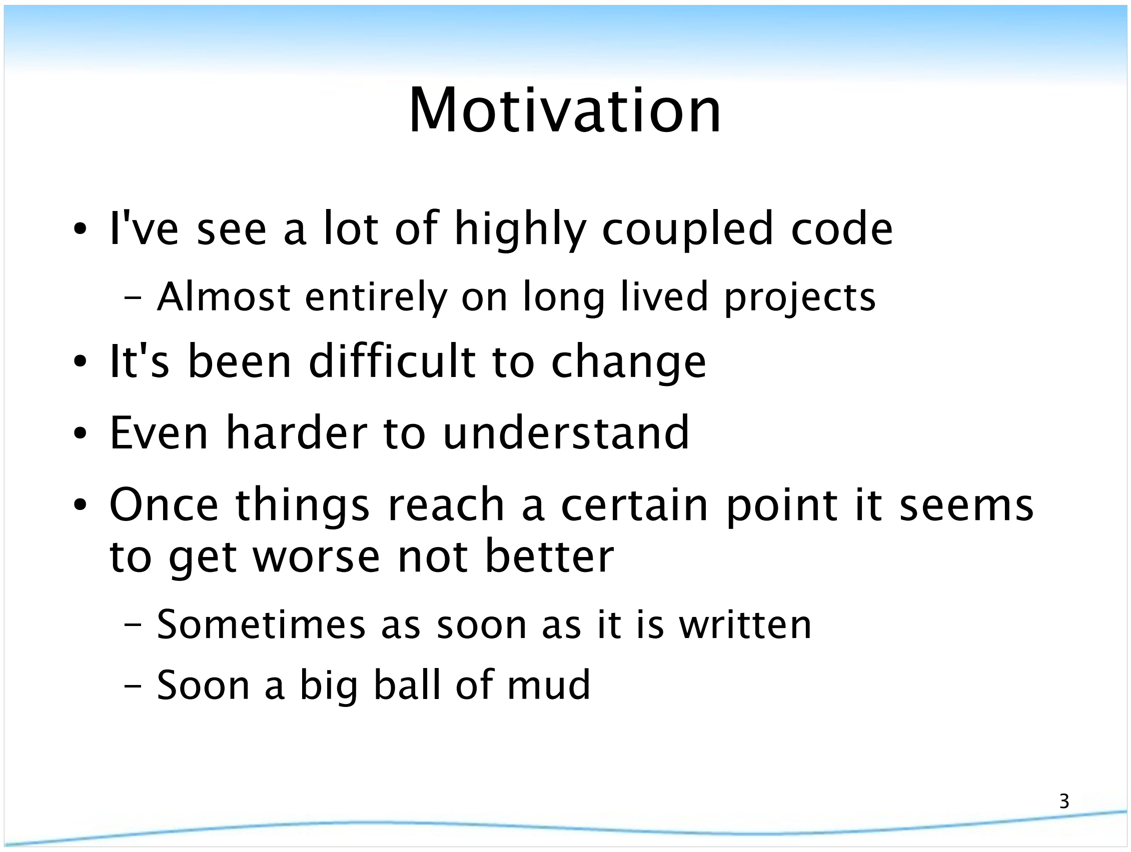
Introduction

- Exploring software coupling
  - Some history
  - Categories
  - A look at cohesion
- Measuring coupling
- Looking at principles and techniques that help us with coupled software

This session explores the categories used to describe software coupling from Structured Programming, OO and some related to higher level concepts such as packages. Along the way the session also covers categories of cohesion which are related to the categories of coupling.

This session will consider some of the metrics that can be used to understand the coupling of modules, classes and packages in a design.

Later in the session some archetypical coding problems which demonstrate bad coupling are described. These coding problems are related to good design principles that reduce coupling and should be considered when evaluating the design problems. Some practices which can be used to reduce coupling are also considered in this section of the session.



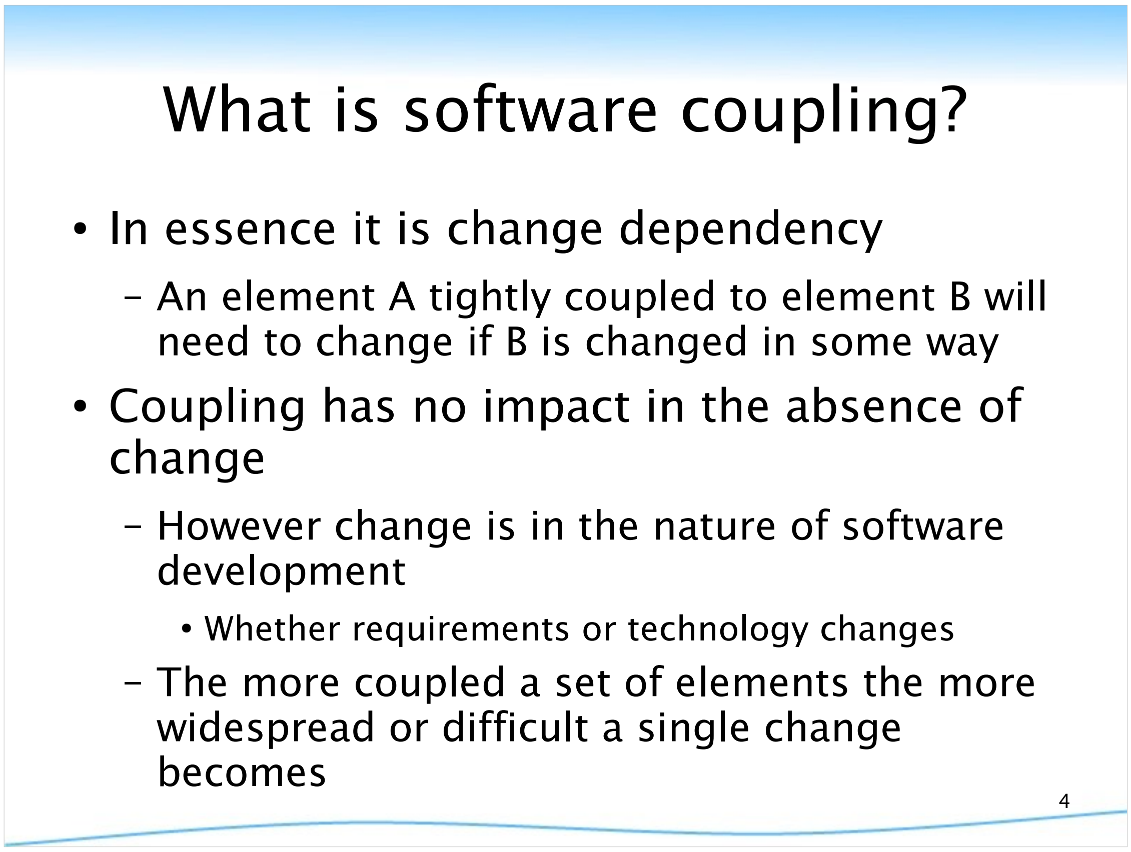
Motivation

- I've see a lot of highly coupled code
  - Almost entirely on long lived projects
- It's been difficult to change
- Even harder to understand
- Once things reach a certain point it seems to get worse not better
  - Sometimes as soon as it is written
  - Soon a big ball of mud

This session is motivated by my need to understand coupled designs that I have encountered during my career as a software developer. How small problems in the code are signs that there are coupling problems, how good design principles are not being applied during the development or maintenance of the software and what techniques can be used to bring the software back into a more maintainable form.

Software which is badly coupled is hard to change because a small change in a module can cause changes in many other modules which depend on it, this is generally termed the ripple effect. When a design is like this, it is also hard to understand as the design usually suffers from cohesion problems. To understand a single concept a large number of modules need to be examined. This compounds the maintenance problems as before a change can be made the code needs to be understood.

Once a code base exhibits bad coupling the coupling problems appear to grow worse more quickly, as a consequence of the lack of cohesion. Thus it is important to keep software in a decoupled state to ensure that ongoing maintenance does not make the software more difficult to change later.


What is software coupling?

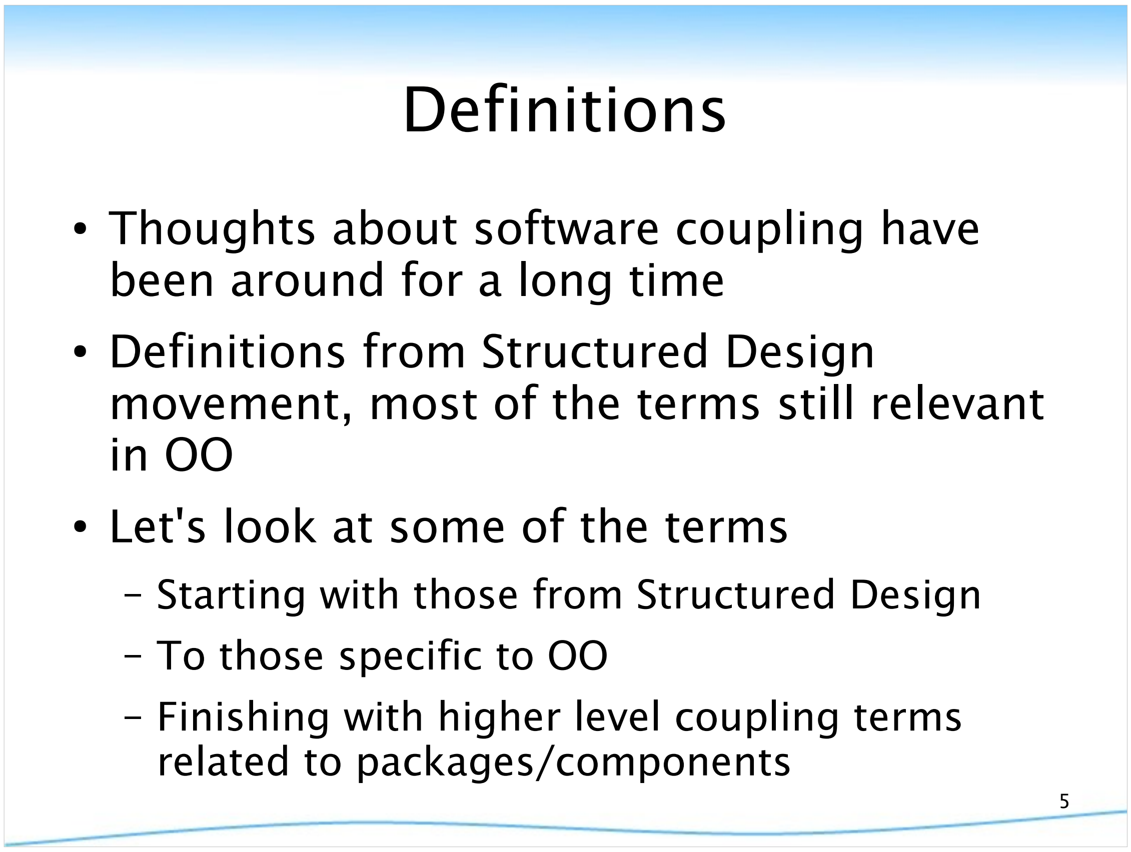
- In essence it is change dependency
  - An element A tightly coupled to element B will need to change if B is changed in some way
- Coupling has no impact in the absence of change
  - However change is in the nature of software development
    - Whether requirements or technology changes
  - The more coupled a set of elements the more widespread or difficult a single change becomes

4

Software coupling is a dependency between elements of a software design, for example functions or classes. This dependency is benign until a change is required to the software which causes the element which is depended upon to be modified and this will generally require the dependent element to change as well. The more dependants an element has the more impact a change has, the more dependencies an element has the more likely it is to be effected by change.

Change is in the nature of software development whether through changes in requirements, technology or change driven by our better understanding of a problem and its solution it will have an impact on the software already in progress.

Thus changes will have an impact on the software we develop and maintain but the extent of the change to the software depends to some extent on how coupled this software is, the more coupled the software the more impact a single change will have.



Definitions

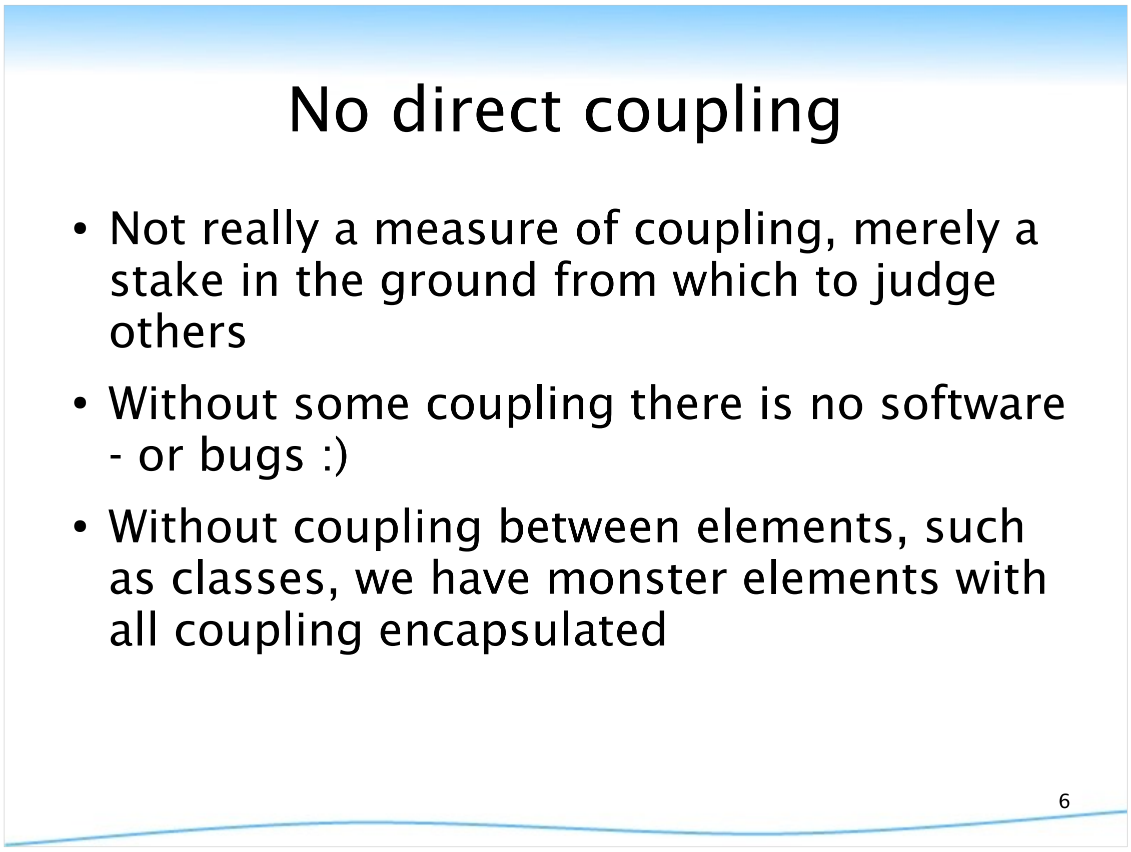
- Thoughts about software coupling have been around for a long time
- Definitions from Structured Design movement, most of the terms still relevant in OO
- Let's look at some of the terms
  - Starting with those from Structured Design
  - To those specific to OO
  - Finishing with higher level coupling terms related to packages/components

5

There is a large amount of literature on software coupling, some of the most widely known being that from the structured design movement of the 1970s. Of the books from this approach the ones most usually cited are *Composite/Structured Design* [Myers79] and *The Practical Guide to Structured System Design* [Page-Jones88]. Both these books have large sections dedicated to the study and consideration of software coupling.

Most of the terms used in these books are still used today to categorize software coupling in OO designs.

The following section explores these terms from structure design, continues to look at those specific to OO design and then looks are those used to describe package level coupling.



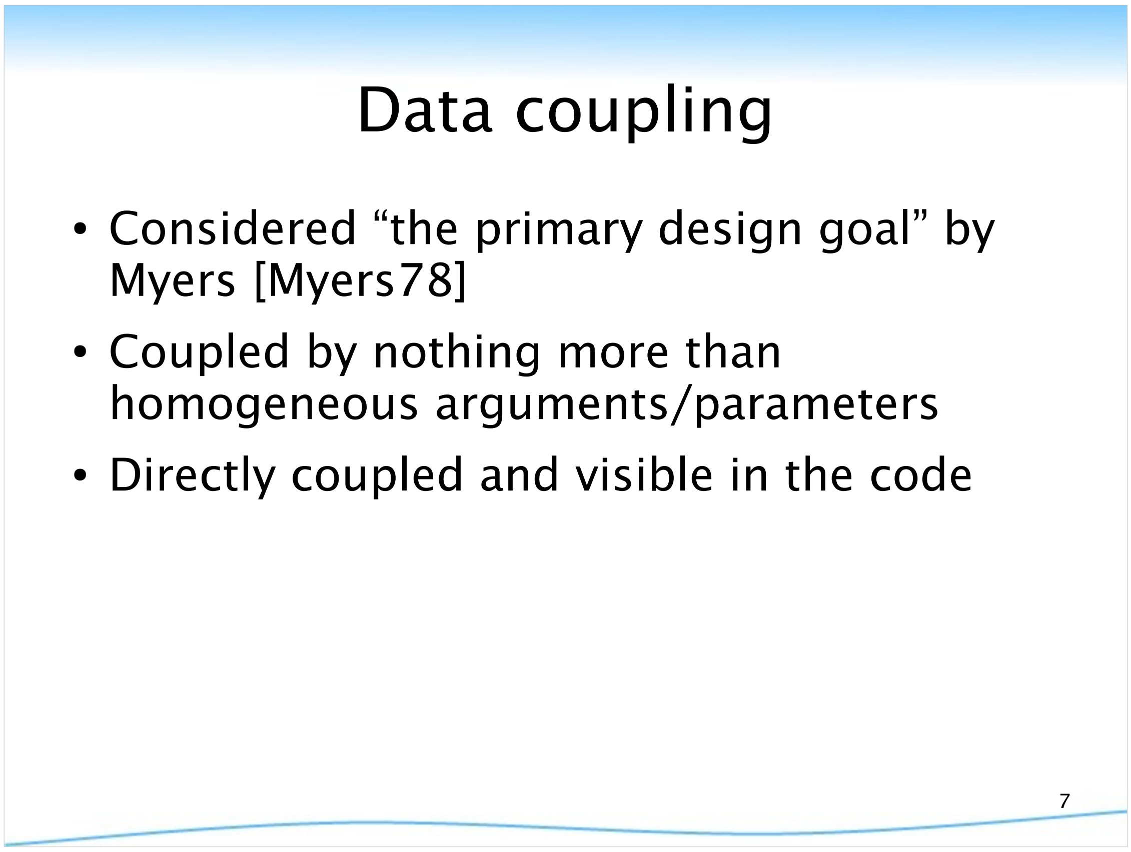
No direct coupling

- Not really a measure of coupling, merely a stake in the ground from which to judge others
- Without some coupling there is no software - or bugs :)
- Without coupling between elements, such as classes, we have monster elements with all coupling encapsulated

6

The first measure of coupling is no coupling between 2 elements at all, this is presented in *Composite/Structured Design* [Myers78] as a stake in the ground from which to consider the various coupling definitions. Thus this definition is not useful as a term for coupling, however it does acknowledge that in any system there are elements which have no relationship at all.

Of course it is possible to produce a system where there is no coupling between elements, if there is only one element, but this just hides the coupling, moving it rather than reducing it.

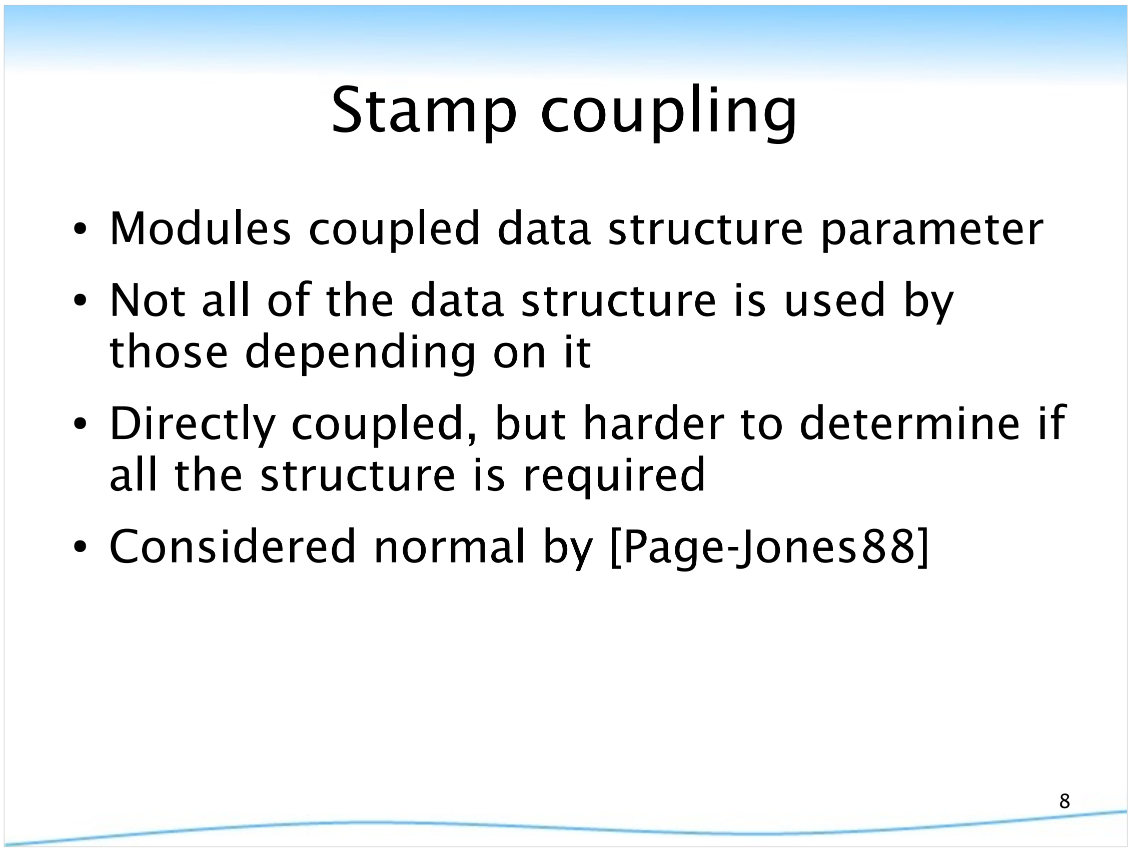


Data coupling

- Considered “the primary design goal” by Myers [Myers78]
- Coupled by nothing more than homogeneous arguments/parameters
- Directly coupled and visible in the code

Data coupling is the first level of coupling considered by Myers [Myers78] and he termed this the “primary design goal” of structure design. The elements are coupled with nothing more than simple homogeneous parameters, such as scaler values or strings. The coupling is directly visible in the code as a function call with the parameters clearly visible.

This kind of coupling allows software to be easily comprehended as a minimum amount of information needs to be studied in order to understand the use of the parameters by the function being called.



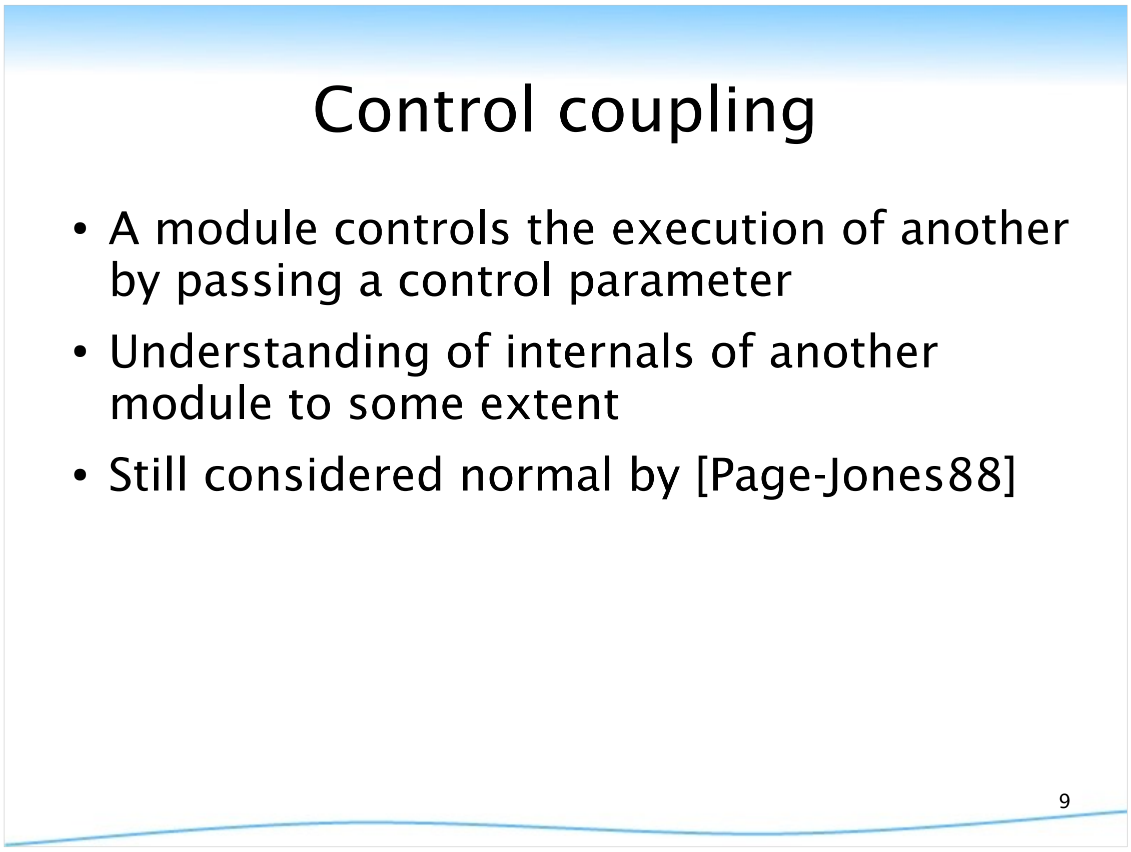
Stamp coupling

- Modules coupled data structure parameter
- Not all of the data structure is used by those depending on it
- Directly coupled, but harder to determine if all the structure is required
- Considered normal by [Page-Jones88]

Stamp coupling introduces a new level of complexity of coupling compared to data coupling as a composite type, a structure, is used as a parameter. This is increased coupling as both elements and the structure need to be comprehended.

Stamp coupling introduces a new coupling issue, being coupled to information which is not required. The called function may not need all parts of the structure in order to achieve its aim, thus more information than required is being passed in. When trying to understand this code it is not possible to understand what parts of the structure are used without looking into the function being called.

In *The Practical Guide to Structured System Design* [Page-Jones88] this level of coupling is considered normal.

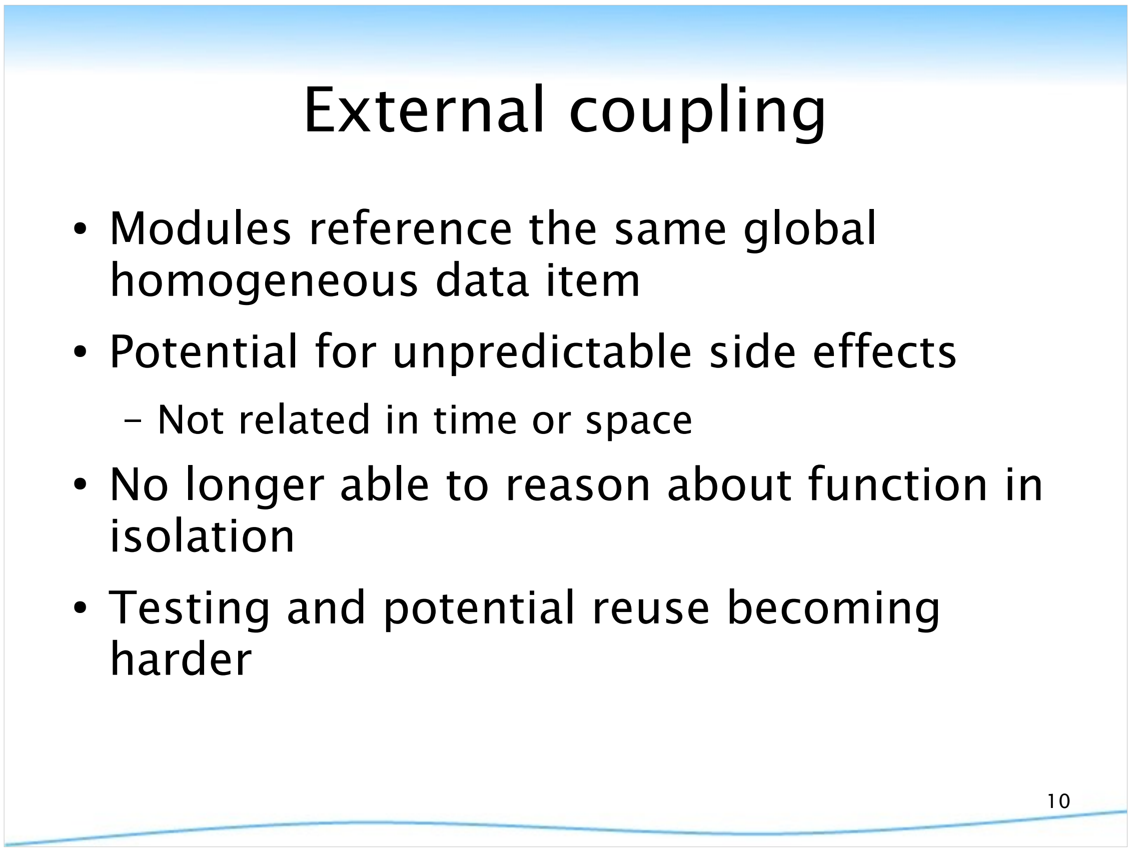


Control coupling

- A module controls the execution of another by passing a control parameter
- Understanding of internals of another module to some extent
- Still considered normal by [Page-Jones88]

Control coupling introduces a level of coupling where the caller understands something of the internals of the called function. Thus the caller is able to control the processing of the called function by passing a variable design for that purpose, another way to look at this would be to say that the called function has leaked information about its implementation to the outside.

Whilst this is more tightly coupled than the previous data and stamp coupling this is still considered normal in *The Practical Guide to Structured System Design* [Page-Jones88].



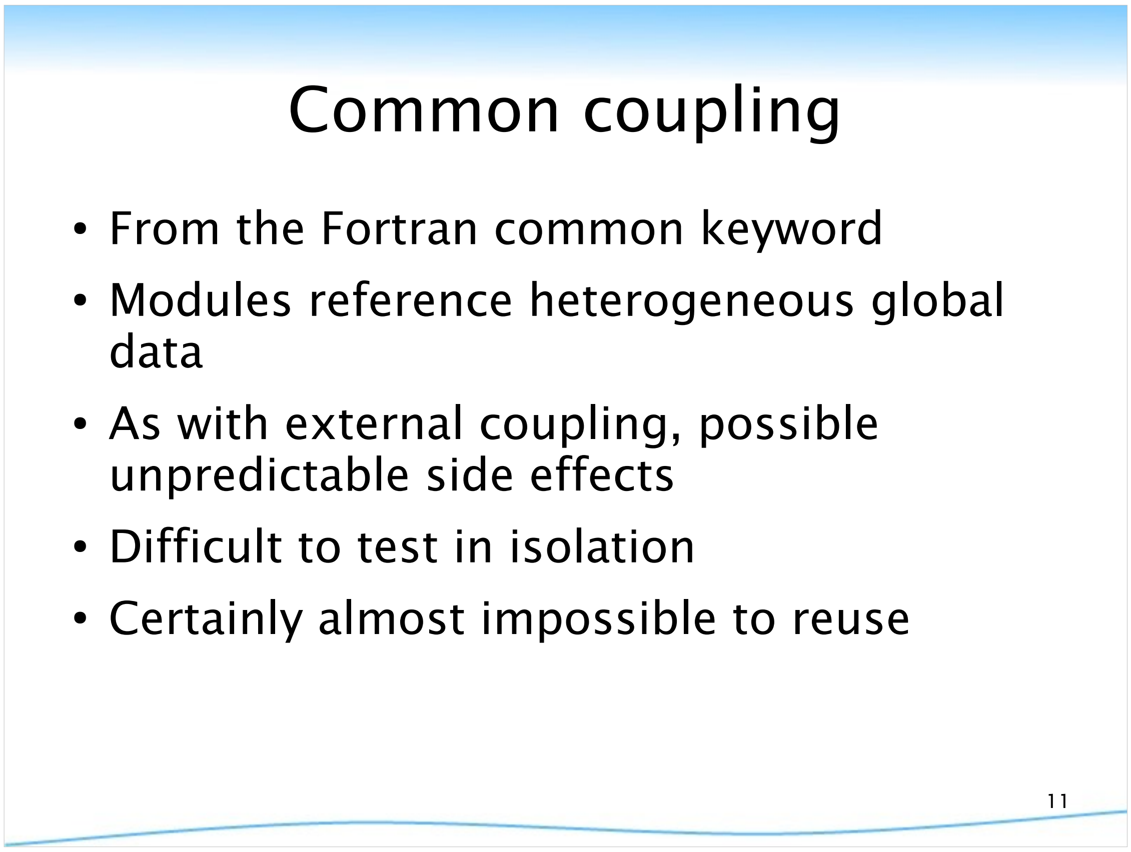
External coupling

- Modules reference the same global homogeneous data item
- Potential for unpredictable side effects
  - Not related in time or space
- No longer able to reason about function in isolation
- Testing and potential reuse becoming harder

10

External coupling is a dependency on a global homogeneous data item, such as a scaler or string. This level of coupling ensures that a function does not advertise the full extent of it input or output data, as the global data item may serve as either. Thus a user of a function with external coupling must examine the function in order to understand what data will be modified.

The most difficult aspect of this coupling is the unpredictable nature of the data in a global data item, as either an input which could cause a function to change the processing performed, or as an output, which may change the processing of other functions which depend on the same item. Thus testing becomes more difficult, as it is hard to predict the state of the global data item and thus hard to specify a valid set of tests for the function.



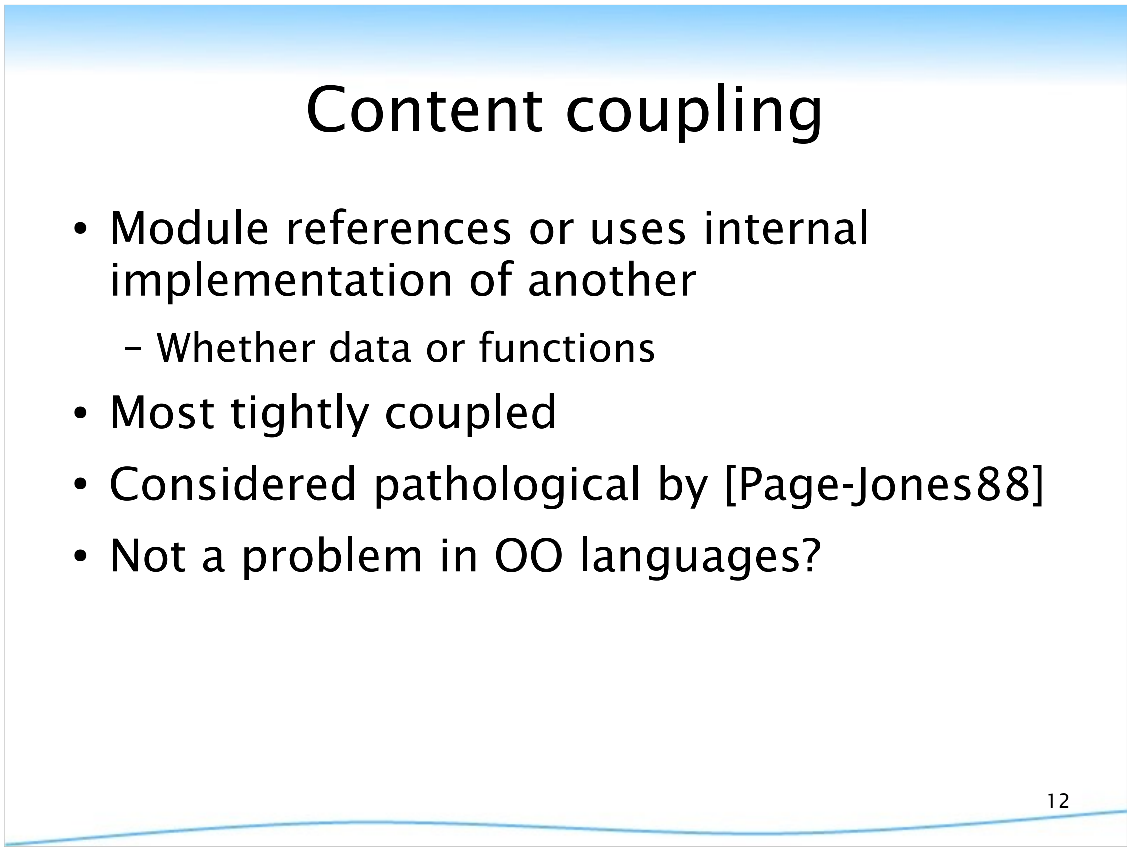
Common coupling

- From the Fortran common keyword
- Modules reference heterogeneous global data
- As with external coupling, possible unpredictable side effects
- Difficult to test in isolation
- Certainly almost impossible to reuse

Common coupling is a dependency on a global heterogeneous data, i.e. a composite type such as a structure. Like external coupling this coupling impacts the understandability of a function, and makes prediction about the processing hard.

More importantly with this level of coupling is that the dependency on the structure may only be partial, in that only part of the structure is required by the function to perform its processing, yet the dependency is on the entire structure.

We should expect, like external coupling, to find it more difficult to predict the processing performed by a function with common coupling as the data in the structure is beyond the control of the caller and function being called. Thus as before testing is more difficult.



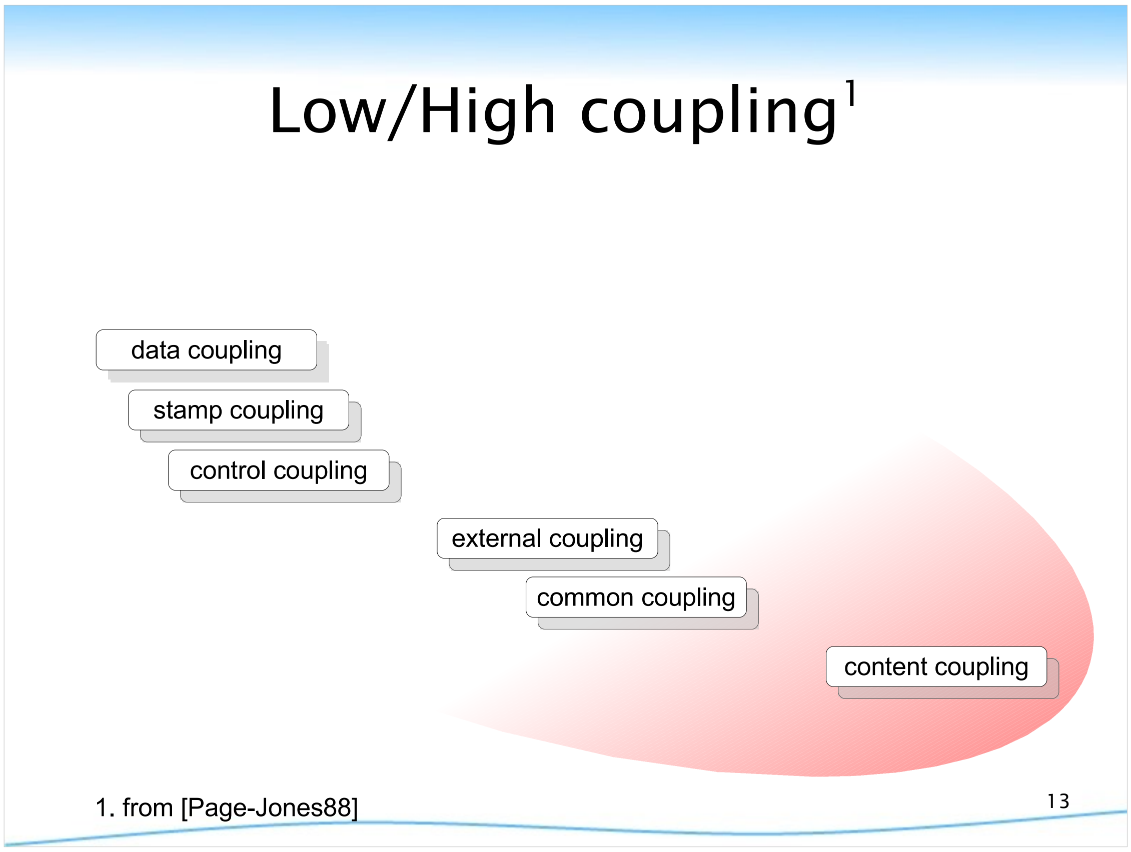
Content coupling

- Module references or uses internal implementation of another
  - Whether data or functions
- Most tightly coupled
- Considered pathological by [Page-Jones88]
- Not a problem in OO languages?

12

Content coupling is a use of the internal implementation of a element breaking the encapsulation of the element, the example given in *The Practical Guide to Structured System Design* [Page-Jones88] is a jump into the middle of a function to perform some processing. This is considered “pathological” coupling [Page-Jones88] as the coupling is the tightest possible, making change to the called function very difficult as the point at which the caller jumps to cannot be controller using the usual language mechanisms.

In OO access to a function in such a way is usually not possible, though it is possible to gain access to nominally private data of a module if it is made public.

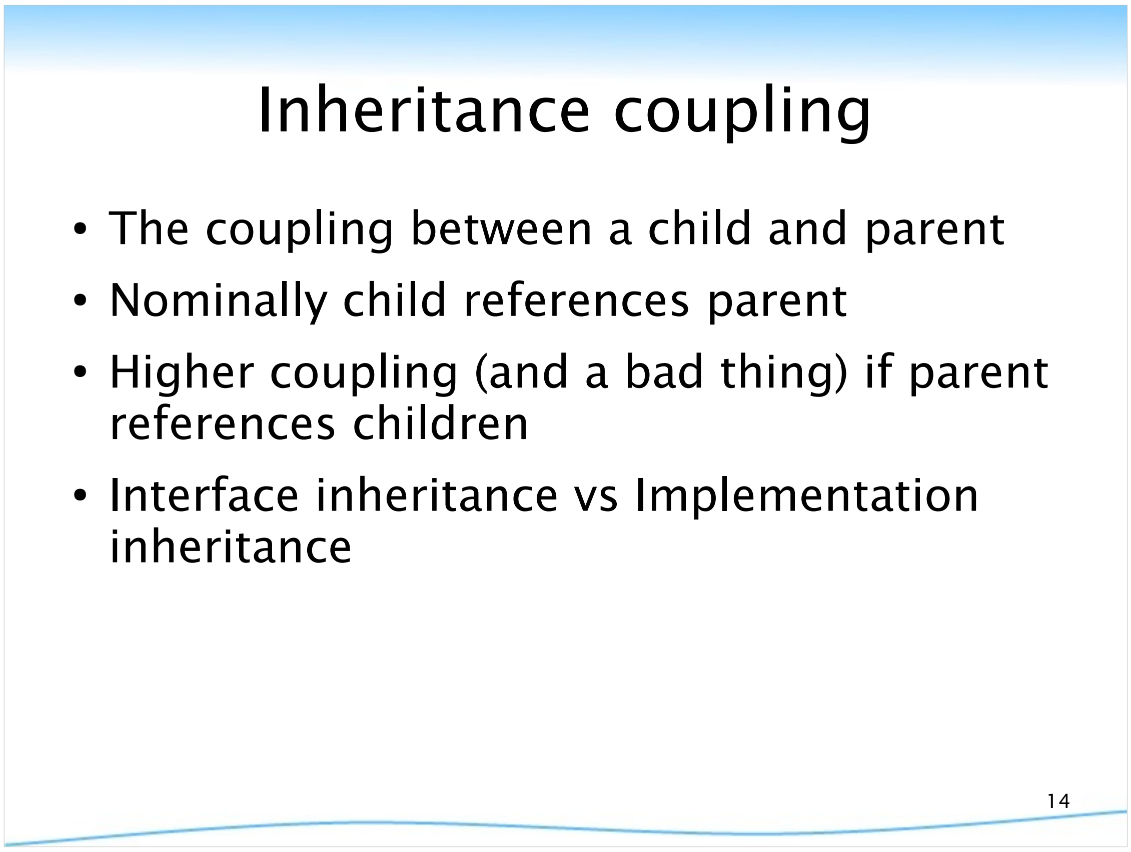


Low/High coupling[1]

data coupling
stamp coupling
control coupling
external coupling
common coupling
content coupling

1. from [Page-Jones88]

In *Composite/Structured Design* [Myers78] the various levels of coupling were presented as an overlapping continuum, but in *The Practical Guide to Structured System Design* [Page-Jones88] the above distinction was made between the various levels, with the “normal” coupling on the left and the “pathological” on the right (in the zone of danger, or embarrassment depending on how red this is on your monitor).



Inheritance coupling

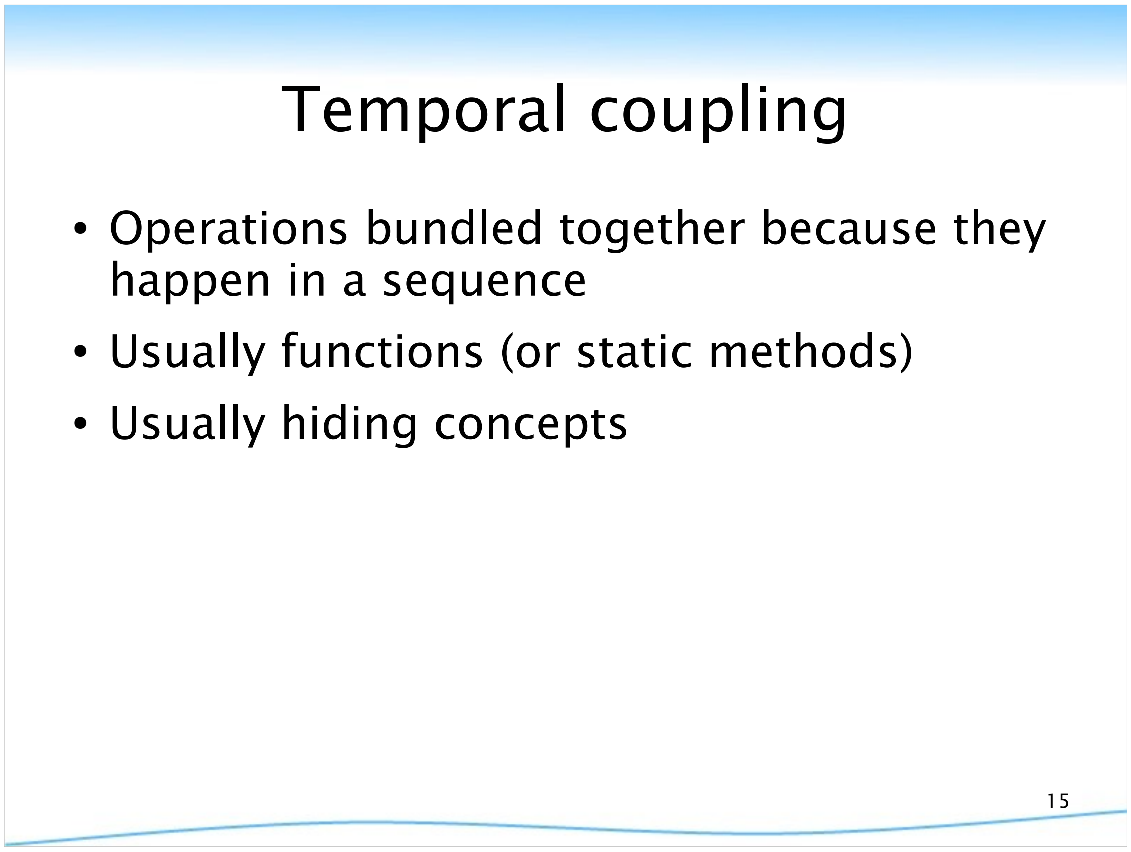
- The coupling between a child and parent
- Nominally child references parent
- Higher coupling (and a bad thing) if parent references children
- Interface inheritance vs Implementation inheritance

14

OO adds additional coupling terms compared to those in procedural code, the first being related to inheritance. Any inheritance relationship between elements is a tight coupling as the child element is dependent on the full implementation of the parent, depending on the nature of the inheritance.

Of course this coupling can be increased in the parent element has knowledge of the child (or children) but such a dependency is discouraged as it interferes with a design meeting the open/closed principle (discussed later).

It is important to draw the distinction between interface inheritance, or “is-a”, and implementation inheritance, a means of sharing code. Whilst some languages make this distinction clear, others do not, and it is important to understand the impact on coupling of the different forms of inheritance. Interface inheritance is usually deemed less coupled than implementation inheritance as the dependency of interface inheritance is based on an abstract (or pure interface) which is more stable in a design.



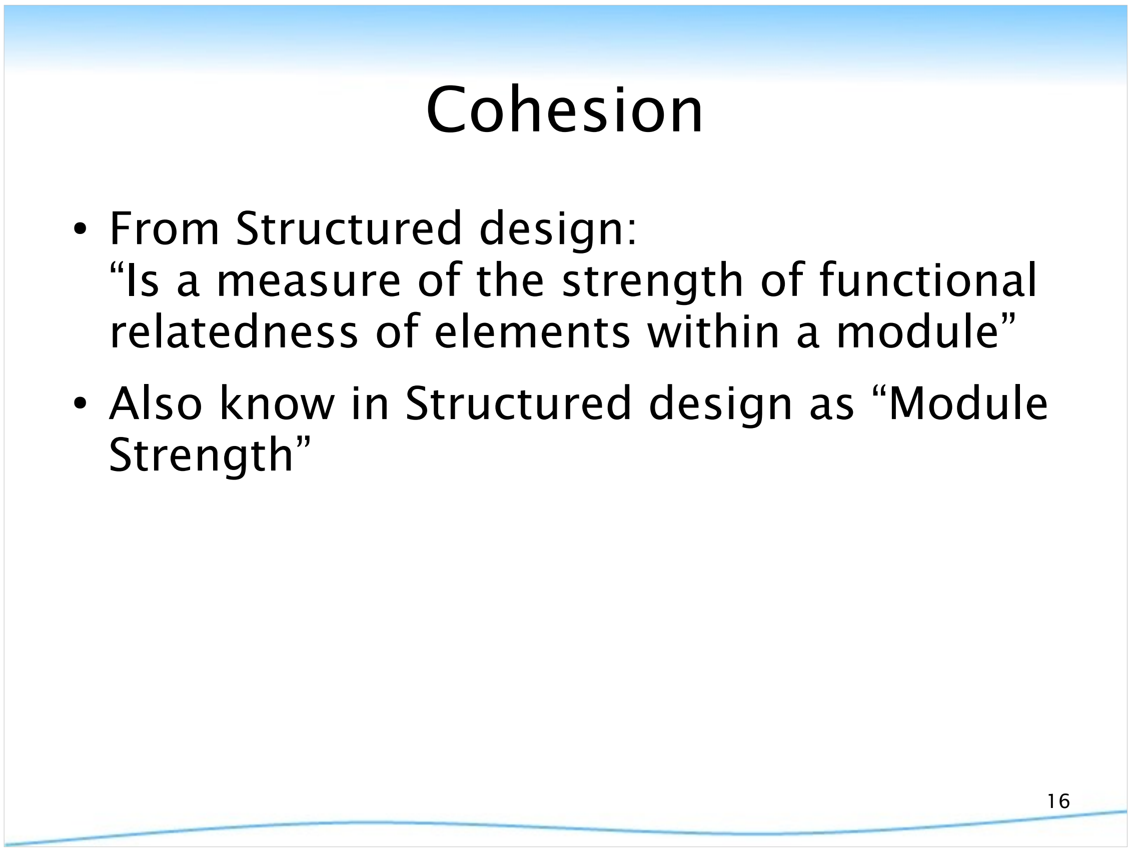
Temporal coupling

- Operations bundled together because they happen in a sequence
- Usually functions (or static methods)
- Usually hiding concepts

Whilst often associated with OO design, temporal coupling applies equally to procedural designs too. The important fact is that processing is coupled together because it happens in a particular order, rather than for any other criteria such as domain based concepts.

The problem with this form of coupling is that it is usually unclear, as compared to a design based around collaborating objects. That this sequence is apt to change causes this kind of design to be fragile as the coupling tends to re-enforce the order of the original sequence.

It is interesting to note that the term temporal coupling is also sometimes used to describe those objects which require multiple calls to be completely initialised. For example, a constructor does not initialise all fields of the object and a second call to a set method is required to finish the initialisation.



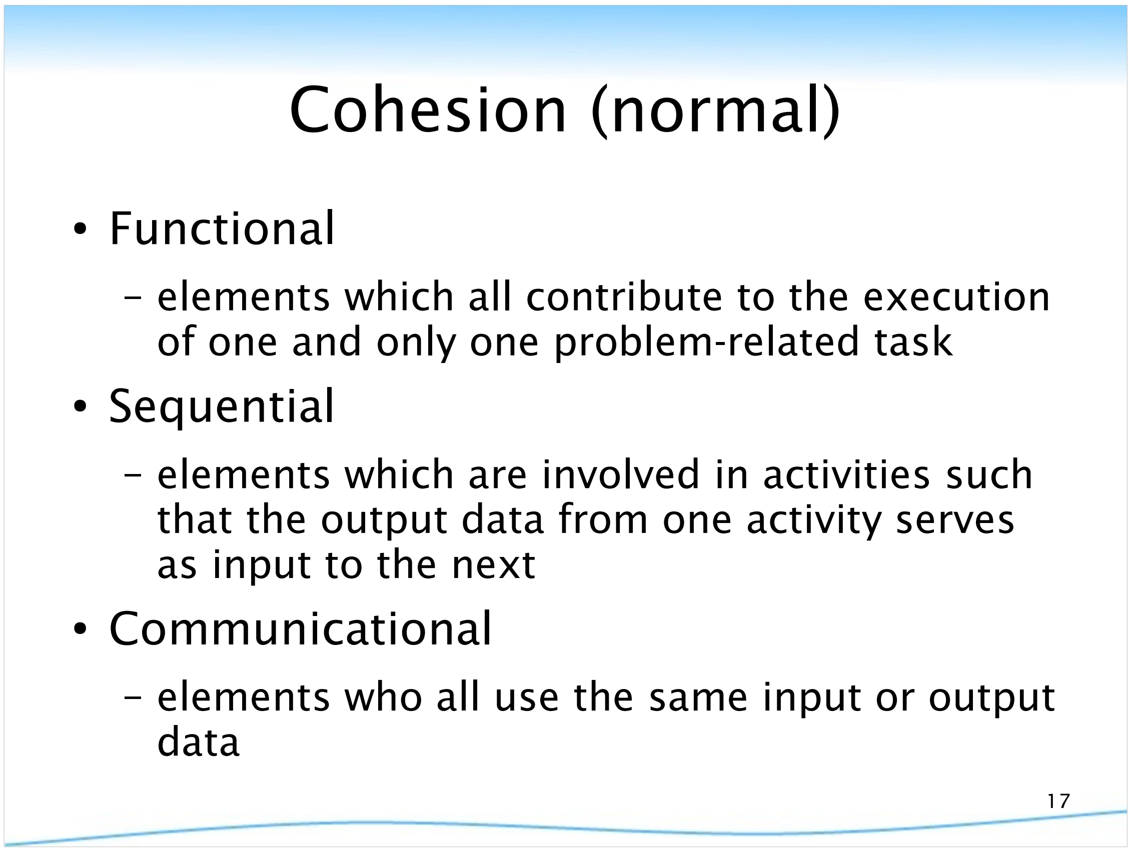
Cohesion

- From Structured design:
  "Is a measure of the strength of functional relatedness of elements within a module"
- Also know in Structured design as "Module Strength"

16

Cohesion describes how related a module is to a specific function or concept in the design. Thus it provides a way of describing how focused a module is on a particular set of operations related to a single aim. It also allows us to reason about what changes may impact a module, for example a highly cohesive module is only likely to change for a single reason, as it supports a single concept. A module which is not cohesive is likely to change for many reasons, as it supports many concepts, albeit in all likelihood only partially.

In structured design cohesion was also known as Module Strength, rather than meaning string and brittle, it reflects on the cohesion to a single concept and thus the likelihood the module would remain stable even while changes in the design occurred.



Cohesion (normal)

- Functional
  - elements which all contribute to the execution of one and only one problem-related task
- Sequential
  - elements which are involved in activities such that the output data from one activity serves as input to the next
- Communicational
  - elements who all use the same input or output data

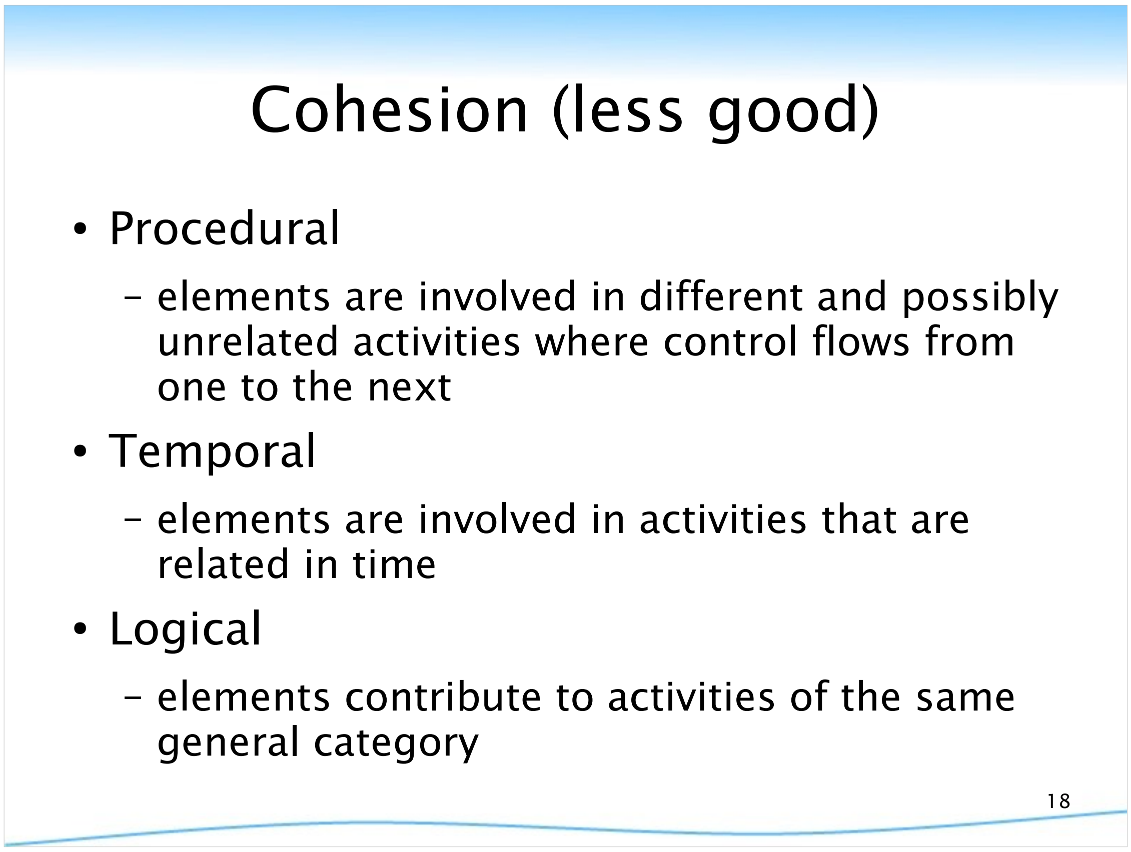
17

These terms for cohesion are taken from *The Practical Guide to Structured System Design* [Page-Jones88] but these terms are less widely used as the more general term cohesion with some qualification, such as high or bad. These terms, with some small variations, are also used in *Composite/Structured Design* [Myers78].

These levels of cohesion could be considered normal, functional cohesion being the most cohesive, where functions support a single concept or task. System that contain normally coupled, functionally cohesive, modules are the most easy to maintain.

The second, sequential cohesion, is where functions are related in terms of a data flow, from one to the next, but where the functions cannot be described by a single concept. This cohesion resembles an assembly line, where an overall task is performed by the individual actions are only related to each other in order and the object they act on.

Finally communication cohesion is where elements share the same input and output data and thus are cohesive to the concept the data represents. For example operations which perform actions on a customer data element.



Cohesion (less good)

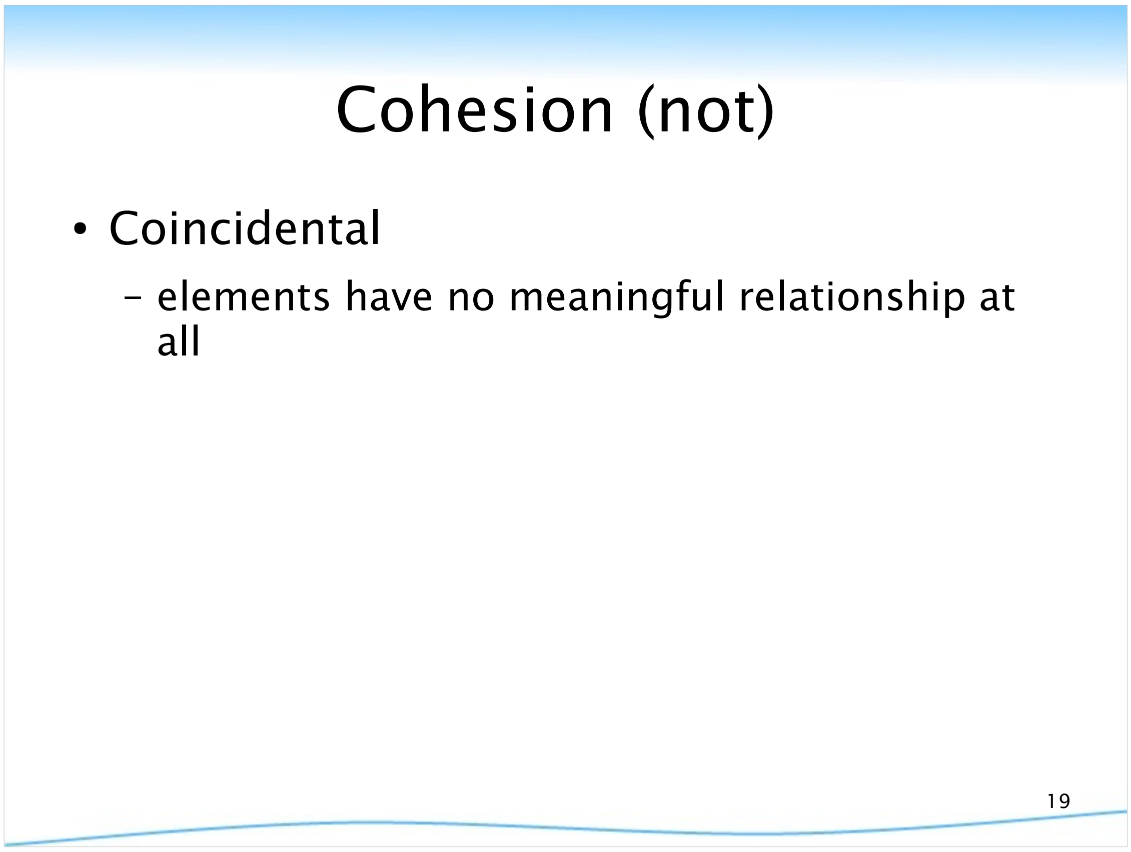
- Procedural
  - elements are involved in different and possibly unrelated activities where control flows from one to the next
- Temporal
  - elements are involved in activities that are related in time
- Logical
  - elements contribute to activities of the same general category

These terms are reserved for cohesion which is less good, though exhibiting some qualities of cohesion, in a system elements with these forms of cohesion will be less maintainable.

Procedural cohesion is where functions are related by a control flow, but where the functions are not necessarily related in any other way, such as sharing the same data as sequential cohesion, etc.

Temporal cohesion is when functions are involved in activities related in time, for example a start of day process, which has activities like, switch on computer, make coffee, etc.

Logical cohesion is where elements are based on activities in the same general category, for example initialisation, or something related to travel, but where there is no other relationship.

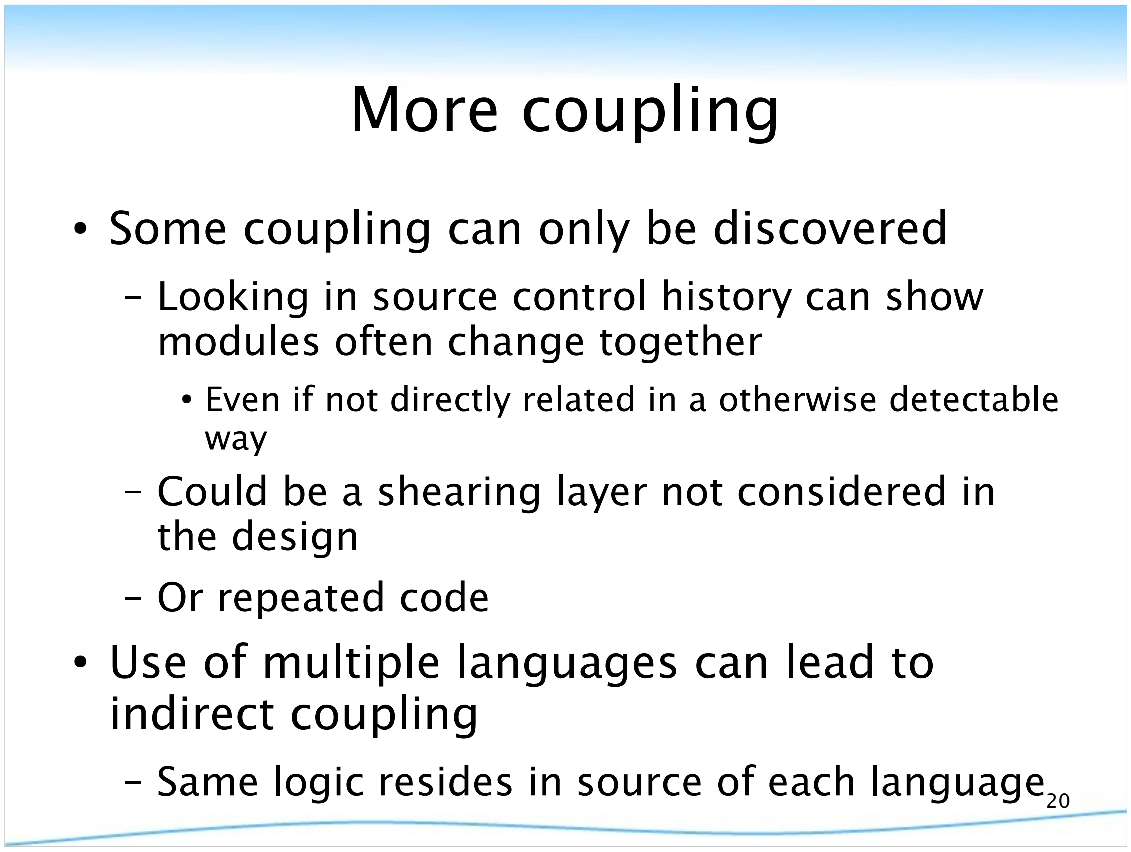


Cohesion (not)

- Coincidental
  - elements have no meaningful relationship at all

The final category of cohesion is coincidental, where there is no relationship between elements at all. This kind of cohesion is rare, and is usually considered a result of breaking up monolithic code in an ah-hoc manner into separate elements.

Whilst these terms for cohesion are interesting from a historical point of view, it is unusual to hear them used to describe cohesion in OO systems. The general usage seems to be binary in nature, either cohesive or not, this may be related to the inability to place a value on cohesion beyond the qualitative good or bad.



More coupling

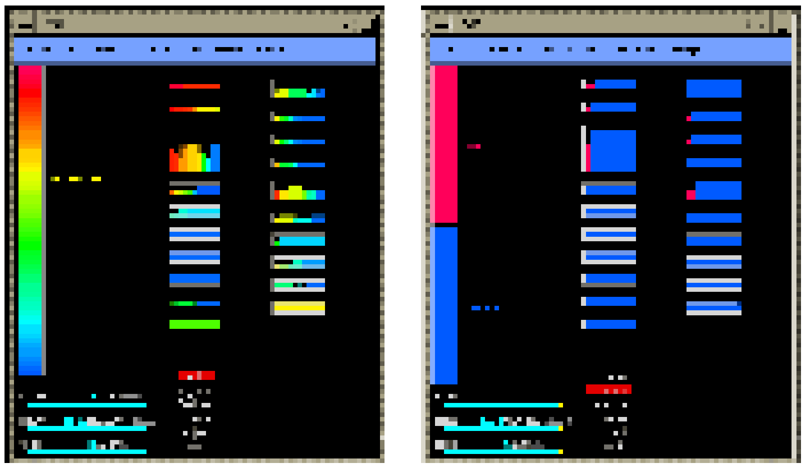
- Some coupling can only be discovered
  - Looking in source control history can show modules often change together
    - Even if not directly related in a otherwise detectable way
  - Could be a shearing layer not considered in the design
  - Or repeated code
- Use of multiple languages can lead to indirect coupling
  - Same logic resides in source of each language

All the coupling categories we have already seen can be seen directly in code as the dependencies are easily apparent by artefacts in the code, such as #include or similar imports. Some coupling however cannot be seen in this way and can only be seen when looking at some other information, for example bug fixes which change a similar set of files, or seeing files which often change in the same transaction.

These kind of effects are likely a consequence of some indirect dependency, whether though a mutual coupling on an unstable element, some axis of change which impacts some elements in the same way (a shearing layer), or whether some repeated code. Of these the repeated code would be harder to find in this way, as it is unlikely to modified at the same time, causing a drift in functionality of the copies from each other.

Use of multiple languages can cause some code to be repeated, for example values and the validation logic that ensures these values are correct.

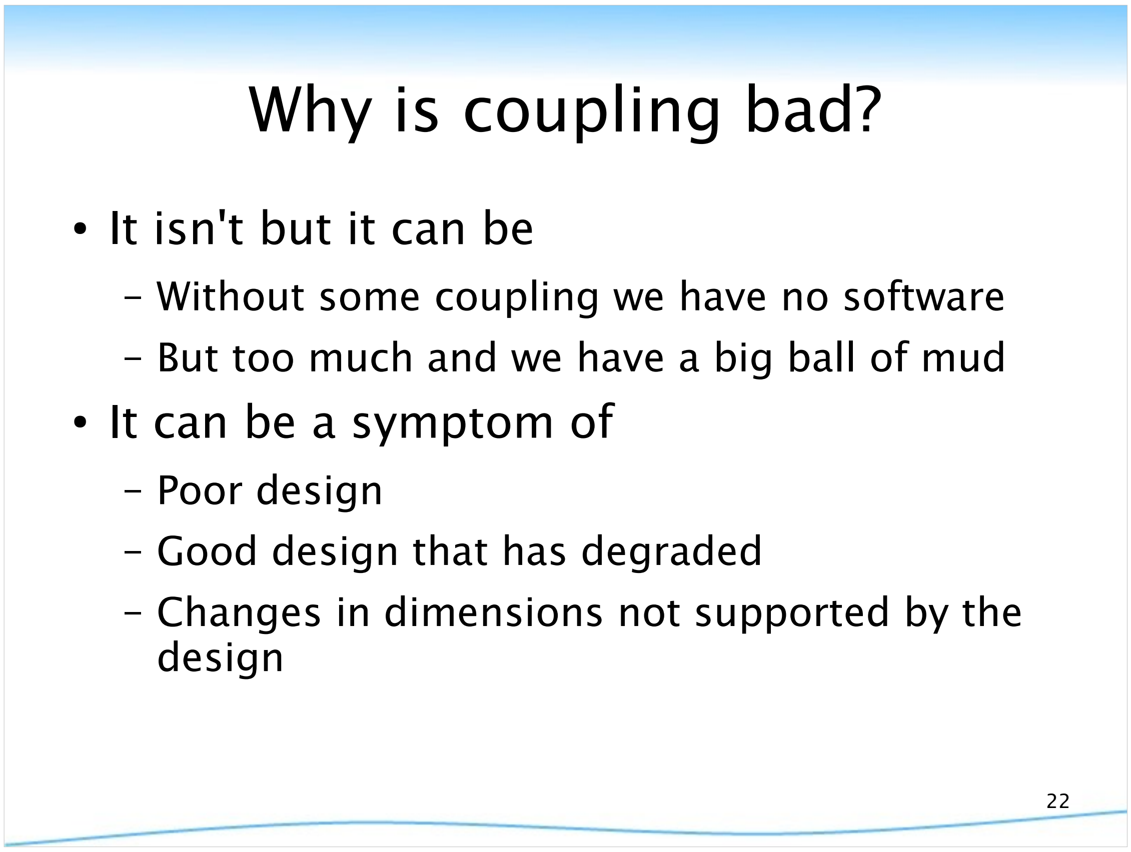
Finding coupling



[illegible]

This diagram is taken from a paper *Software Visualization in the Large* [Ball03] and the right hand pane shows the changes to files related to bug fixes. This kind of visualization illustrates a way to show coupling of file changes to bug fixes. This style of visualization can therefore be used to identify indirect coupling.

Of course, this kind of analysis will not be useful until a good amount of change data is available, but it can indicate files which should be investigated as they may be indirectly coupled.



Why is coupling bad?

- It isn't but it can be
  - Without some coupling we have no software
  - But too much and we have a big ball of mud
- It can be a symptom of
  - Poor design
  - Good design that has degraded
  - Changes in dimensions not supported by the design

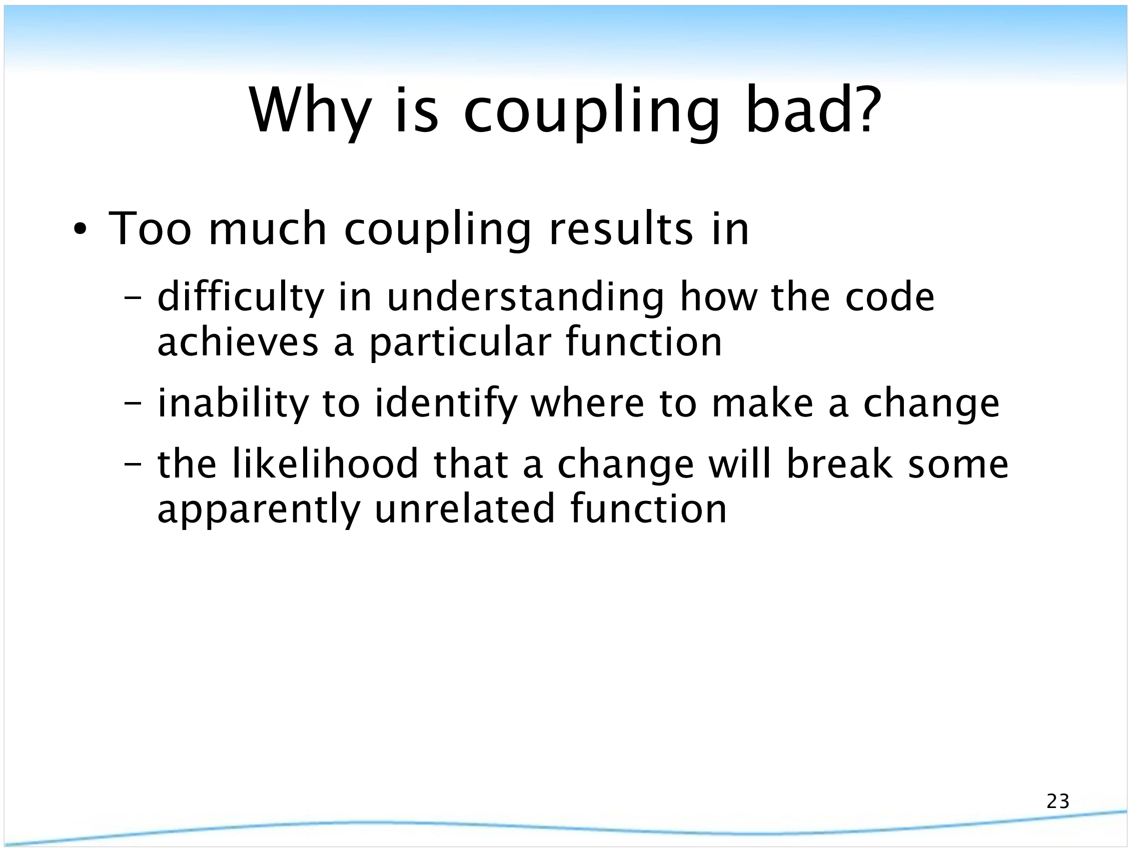
22

So far we have discussed what coupling is and have touched on cohesion, but we have not considered the impact on a design of coupling beyond bad things happen here.

The most important consideration of coupling is that we must acknowledge all software has it, without some coupling between elements we cannot have the best form of design with highly cohesive elements collaborating to achieve a desired task.

So how is it that some coupling is good and some bad, how do we draw the distinction? The most obvious measure of coupling is how many elements are required to change to meet a change in requirements, if a large number of elements are changed for every change in the software then there is a coupling problem. Another way to look at the impact of coupling is how much a programmer is required to understand in order to determine what to change. This kind of consideration falls between the concepts of coupling and cohesion as badly coupled code is usually badly cohesive and so changes are not isolated within defined boundaries.

Bad coupling is a sign that a design is not good, whether poor from its inception, or whether degraded over time due to lack of care, the impact is the inability of the design to adapt to change.



Why is coupling bad?

- Too much coupling results in
  - difficulty in understanding how the code achieves a particular function
  - inability to identify where to make a change
  - the likelihood that a change will break some apparently unrelated function

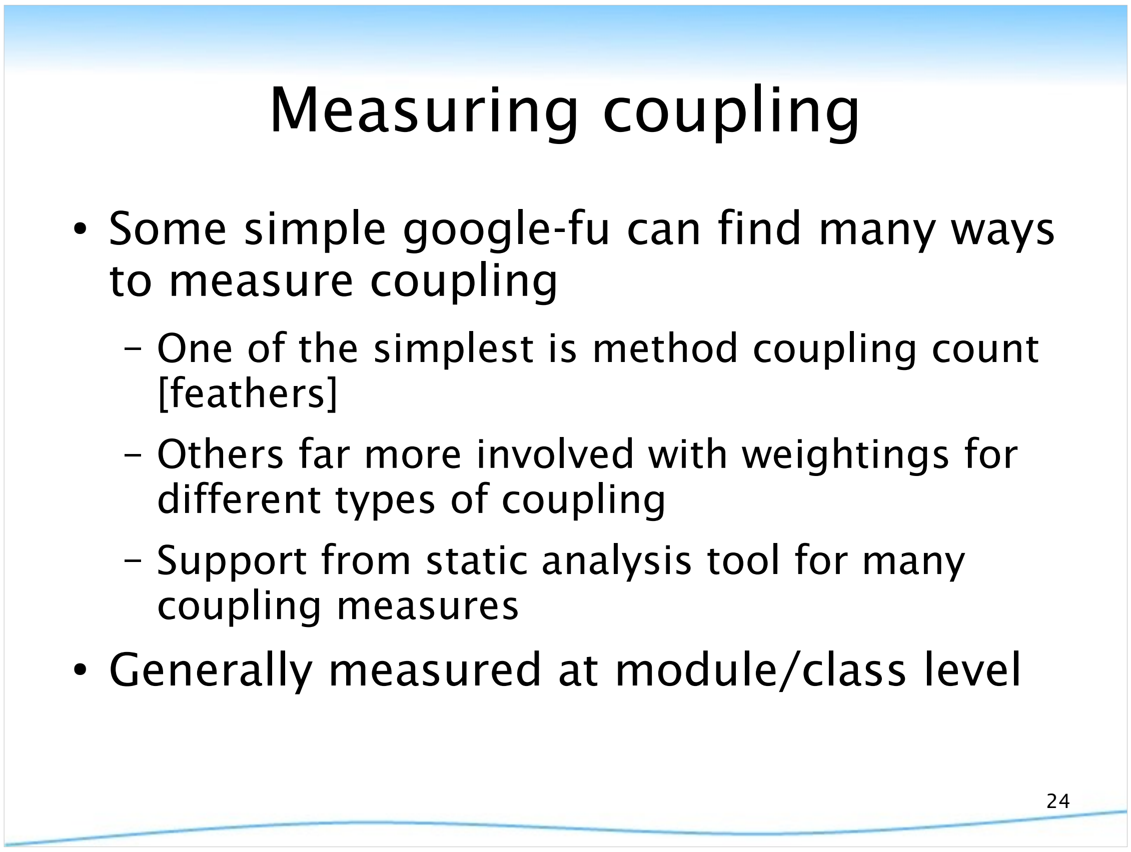
23

As noted above bad coupling in a design impacts understandability and increases that likelihood of changes impacting unrelated functionality.

Code which is highly coupling usually exhibits poor cohesion, and this means that there is no single concept which captures a reason for change, thus any change results in a number of elements being changed. In order to make the change the programmer must find and understand the elements that need to be changed and anything these elements depend on. The more widespread the change is, the more difficult the code is to comprehend as a whole and the more likely a programmer is to make a mistake, which may or may not be detected in testing.

The more coupled a design the further away from a change the dependency horizon is and the greater the chance that a change may impact unrelated code causing it to fail.

Thus highly coupled code, which is likely to be poorly cohesive, is difficult to change correctly compare to code with low coupling and good cohesion.



Measuring coupling

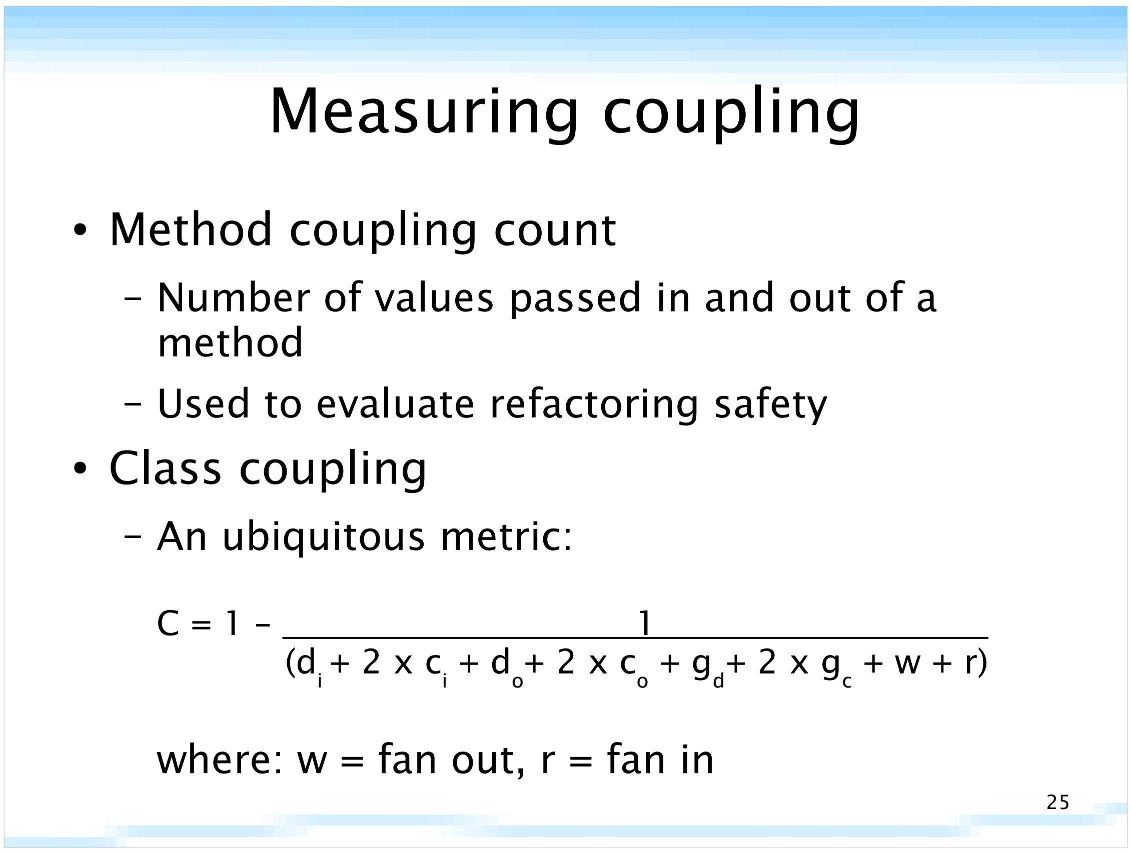
- Some simple google-fu can find many ways to measure coupling
  - One of the simplest is method coupling count [feathers]
  - Others far more involved with weightings for different types of coupling
  - Support from static analysis tool for many coupling measures
- Generally measured at module/class level

24

There are many ways to measure coupling between elements in a design, some quick google-fu and a number of ways to calculate a value for coupling can be found.

As with all measures care must be taken to understand what the meaning of the value is, in general any such measure should only be used to guide the programmer to areas of the code which should be examined for problematic coupling. Reducing the complexity of coupling to a scaler value means that information is lost, for example an element which depends on stable abstract elements could have the same coupling value as one which depends on concrete unstable elements.

Many calculations for coupling are based on module/class level as these are simple to apply and are to some extent the most obviously useful as a tool to a programmer.



Measuring coupling

- Method coupling count
  - Number of values passed in and out of a method
  - Used to evaluate refactoring safety
- Class coupling
  - An ubiquitous metric:

$$C = 1 - \frac{1}{(d_i + 2 \times c_i + d_o + 2 \times c_o + g_d + 2 \times g_c + w + r)}$$

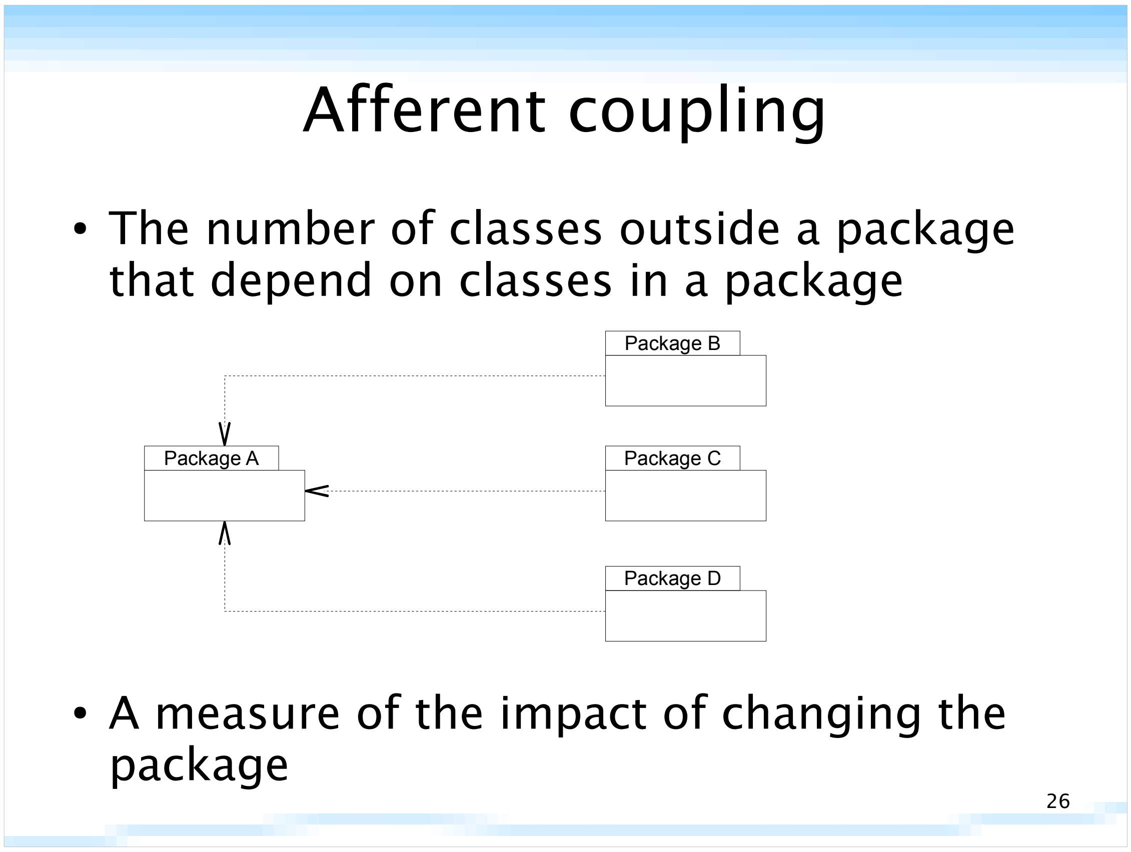
where: w = fan out, r = fan in

One of the simplest ways to measure coupling is one suggested by Michael Feathers [Feathers05], the coupling count, which is the total number of incoming and outgoing parameters. Michael's suggested use for this measure is to guide the programmer in choosing safe refactorings, i.e. to choose refactorings which have a low coupling count.

Other measures are more complex, one often mentioned is the one above, the measures are:

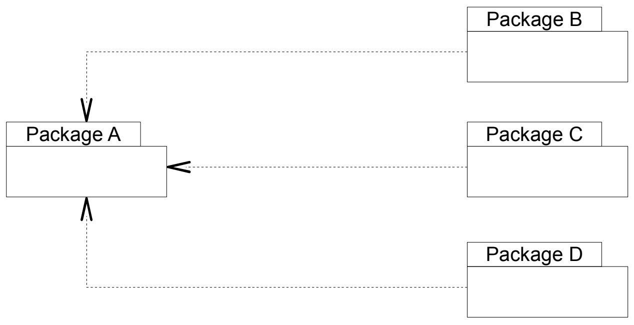
- d_i: number of input data parameters
- c_i: number of input control parameters
- d_o: number of output data parameters
- c_o: number of output control parameters
- g_d: number of global variables used as data
- g_c: number of global variables used as control
- w: number of modules called (fan-out)
- r: number of modules calling the module under consideration (fan-in)

The resultant value is in the range 0.67 to 1 (low to high coupling) with some coupling types (for example control coupling) weighted more than others.



Afferent coupling

- The number of classes outside a package that depend on classes in a package

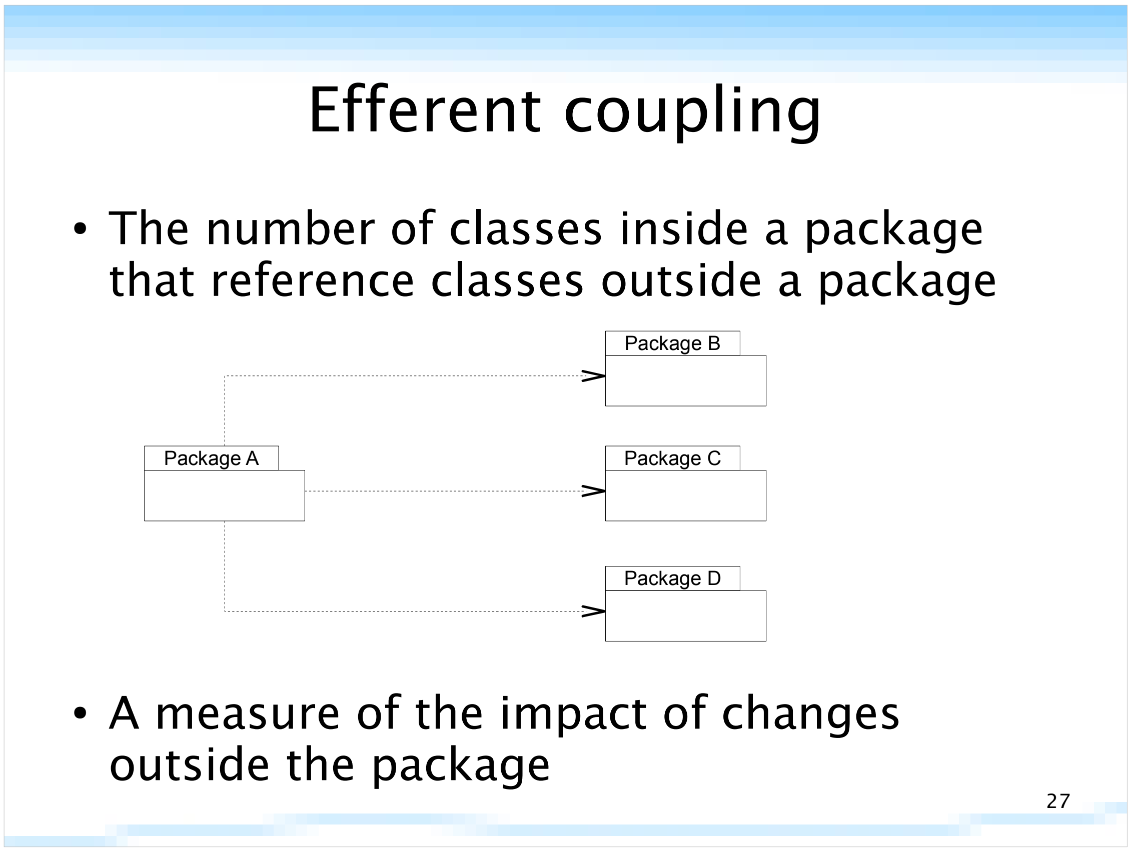


- A measure of the impact of changing the package

There are also measures for package coupling, these are very similar in nature to those simple measures used in module/class coupling. The first is Afferent coupling, simply the number of classes outside a package that depend on classes in that package.

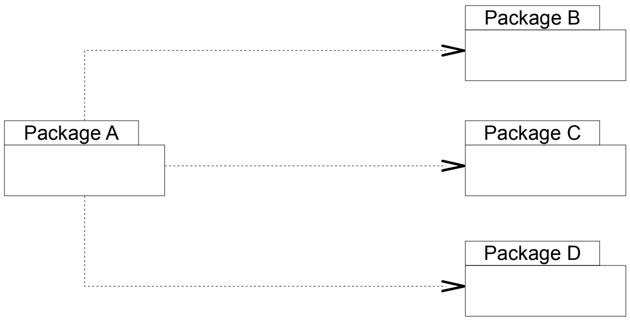
This measure provides us with an indication of the impact of a change in a package, the higher the number of dependencies, the more likely a change is to impact other packages. Of course, it is also possible that a change will not impact any other packages, as the measure is an aggregation of dependencies and therefore cannot tell us what percentage of classes in a package are depended upon.

With this in mind, we should also expect that classes in a package are related in some way and thus a change to a concept may impact man classes in the package, in this case the measure becomes more relevant



Efferent coupling

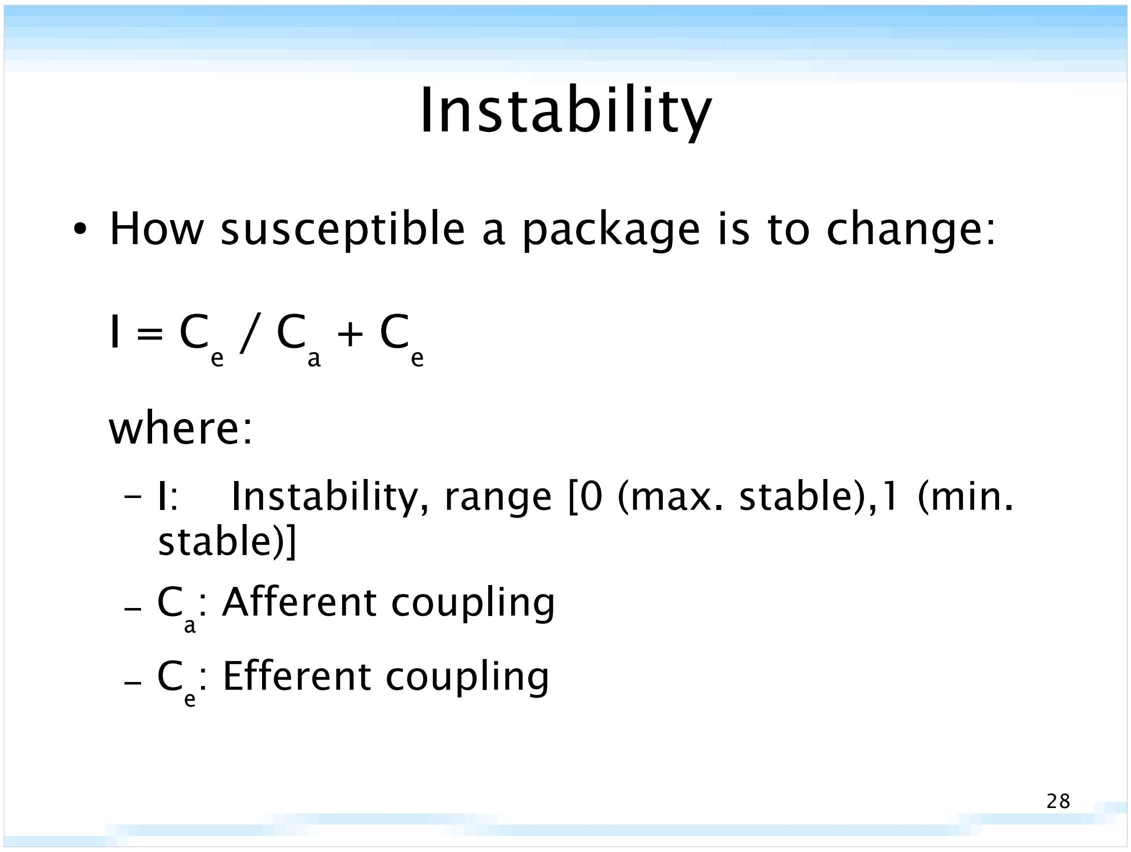
- The number of classes inside a package that reference classes outside a package



- A measure of the impact of changes outside the package

Like afferent coupling, Efferent coupling measures coupling at the package level, but this time the measure is the number of classes from other packages the classes within a package depend on.

This measure provides us with an insight into the likelihood that a package will be impacted by changes in other packages. Like afferent coupling this measure cannot tell the whole story of coupling, but is a useful indicator, nonetheless.



Instability

- How susceptible a package is to change:

  $I = C_e / C_a + C_e$

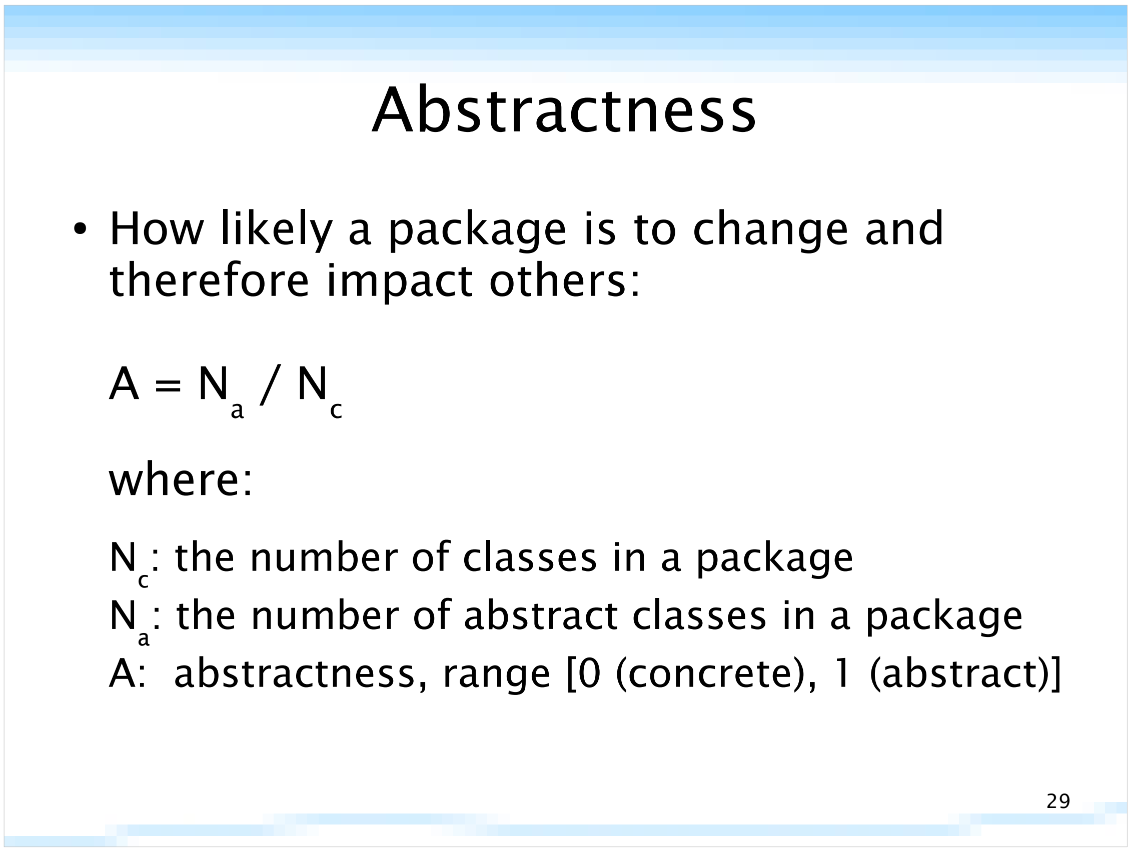
  where:

  - I: Instability, range [0 (max. stable),1 (min. stable)]
  - C_a: Afferent coupling
  - C_e: Efferent coupling

We can derive another useful measure from afferent and efferent coupling measures, instability, in essence the likelihood a change will impact a package. The range of this measure is from 0, where the package is stable, to 1, where the package is unstable.

For a package to be maximally stable it must have no dependencies on any other package, thus any change made to this package will have an impact on other packages. Given this level of impact, we deem this package to be responsible to others, so we should aim to make such a package stable in terms of the design (as well as dependencies).

For a package to be maximally instable it must have only dependencies on other packages, with no other package having a dependency on it. This means any change to a package would have no impact on any other package. Thus we deem this type of package, irresponsible. Thus we expect to find the most unstable design elements in this kind of package. It should be noted that with high efferent coupling such a package will be susceptible to changes in other packages as noted above.



Abstractness

- How likely a package is to change and therefore impact others:

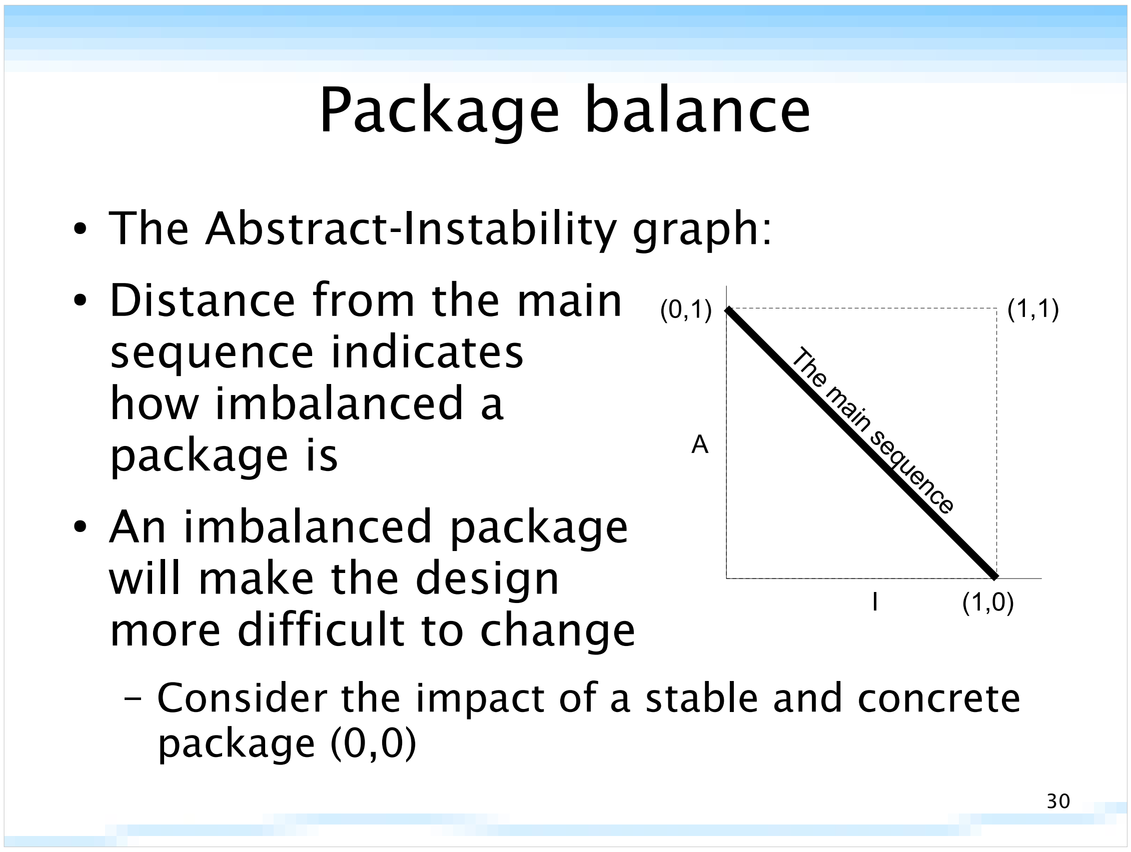
  $A = N_a / N_c$

  where:

  N_c: the number of classes in a package
  N_a: the number of abstract classes in a package
  A: abstractness, range [0 (concrete), 1 (abstract)]

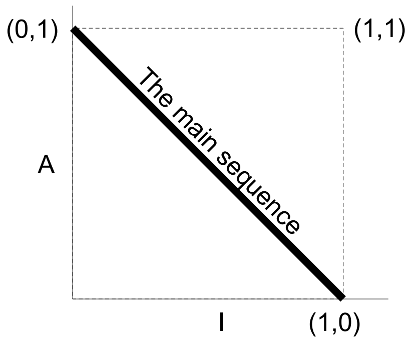
Another measure we can use to judge a package is the abstractness, i.e. the ratio of abstract class to concrete classes. This provides a guidance on the likelihood of a change in a package, based on the assumption that abstract types are less likely to change than concrete elements.

The measure is in the range 0, where all classes in a package are concrete, to 1 where all classes are abstract. Thus it is expected that a package with an abstractness measure of 1 is much less likely to change than a package with a abstractness measure of 0.



Package balance

- The Abstract-Instability graph:
- Distance from the main sequence indicates how imbalanced a package is
- An imbalanced package will make the design more difficult to change
  - Consider the impact of a stable and concrete package (0,0)

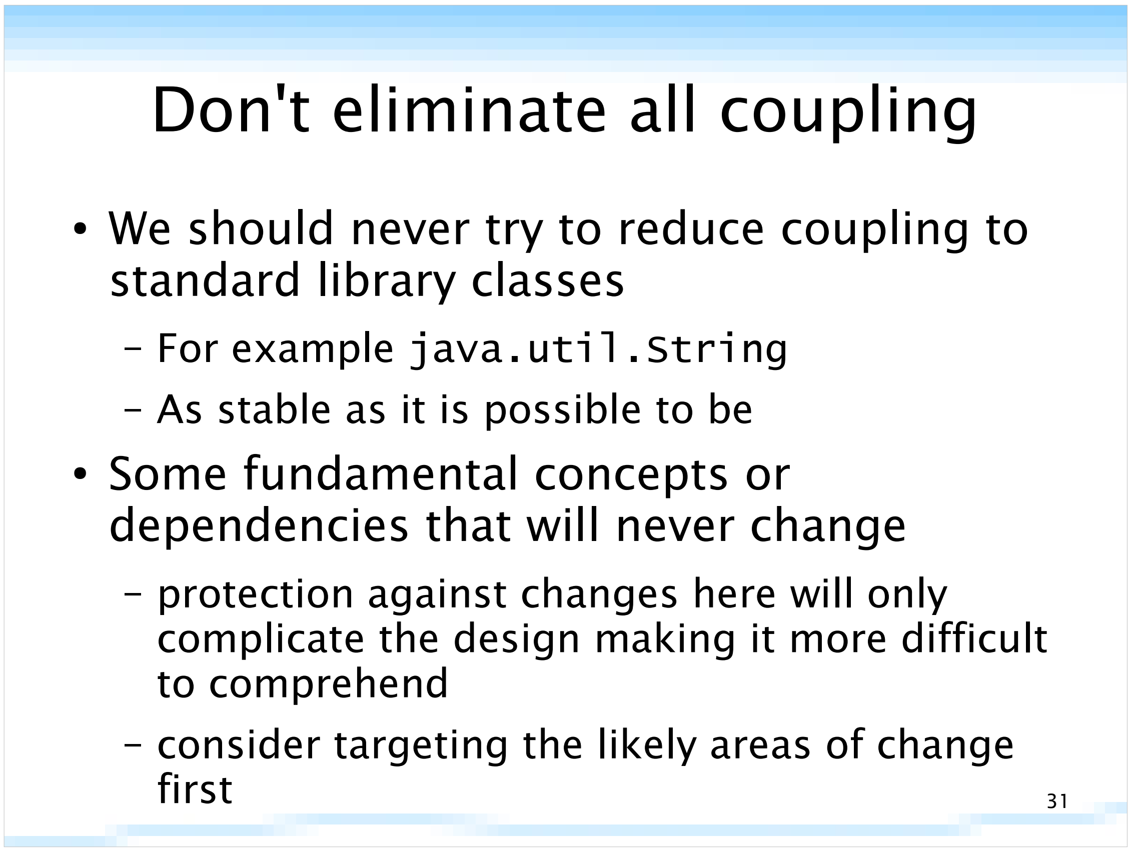


30

We can combine the package measure we have into a final measure, which we can use to determine the usefulness of a package and judge its impact on a design. We do this by mapping the abstractness and instability measures on a graph (as above). Our aim to have the combination of abstractness and instability of a package place it on or near the main sequence.

A package which has a good balance between abstractness and instability will contribute to a design which is easy to change. It can be seen that the book ends of the main sequence are either a stable/abstract package or an instable/concrete package. However it is unlikely we have many of these types of packages, thus we expect a balance that places the package somewhere towards the middle of the graph.

If a package is found to be a long way from the main sequence it is either likely to be both stable and concrete or both instable and abstract. A stable and concrete package is likely to change more often than one which is abstract, but this package will have a high cost of change as it is reference by many other elements. Such a package in a design will decrease the ability of the design to change. An instable and abstract package is unlikely to prove a useful artefact in a design, it depends on many elements, but is mostly abstract, thus has no implementation.



Don't eliminate all coupling

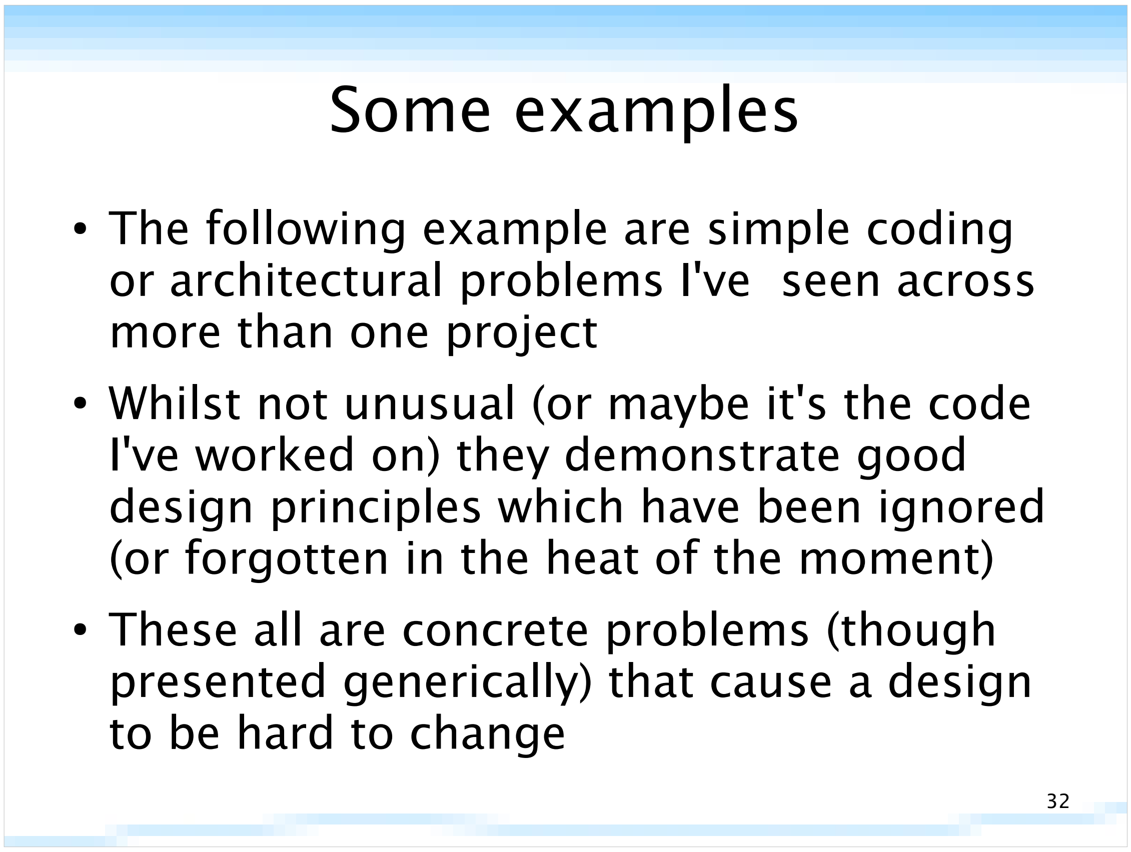
- We should never try to reduce coupling to standard library classes
  - For example `java.util.String`
  - As stable as it is possible to be
- Some fundamental concepts or dependencies that will never change
  - protection against changes here will only complicate the design making it more difficult to comprehend
  - consider targeting the likely areas of change first

31

Whilst we want to reduce unnecessary coupling in our designs so a designs may more easily adapt to change, we should not try to protect designs against all forms of change. For example, it would make a design much more complicated to protect against a change in the java.util package. This is likely to be true for other parts of a design too, for example some parts of a design are fundamental to the purpose of the system and will not change.

Our aim is to reduce dependency on the elements of a design that we can predict will change, or where uncertainty remains, so we reduce the impact when a change occurs.

Thus it is better to target effort on areas of likely change rather than a broad reduction in coupling, this will maximize the design's ability to cope with change, whilst remaining as simple as possible. Of course, we cannot also predict where the changes will occur, but equally we cannot protect a design against all possible changes without making it more complicated and therefore harder to change.



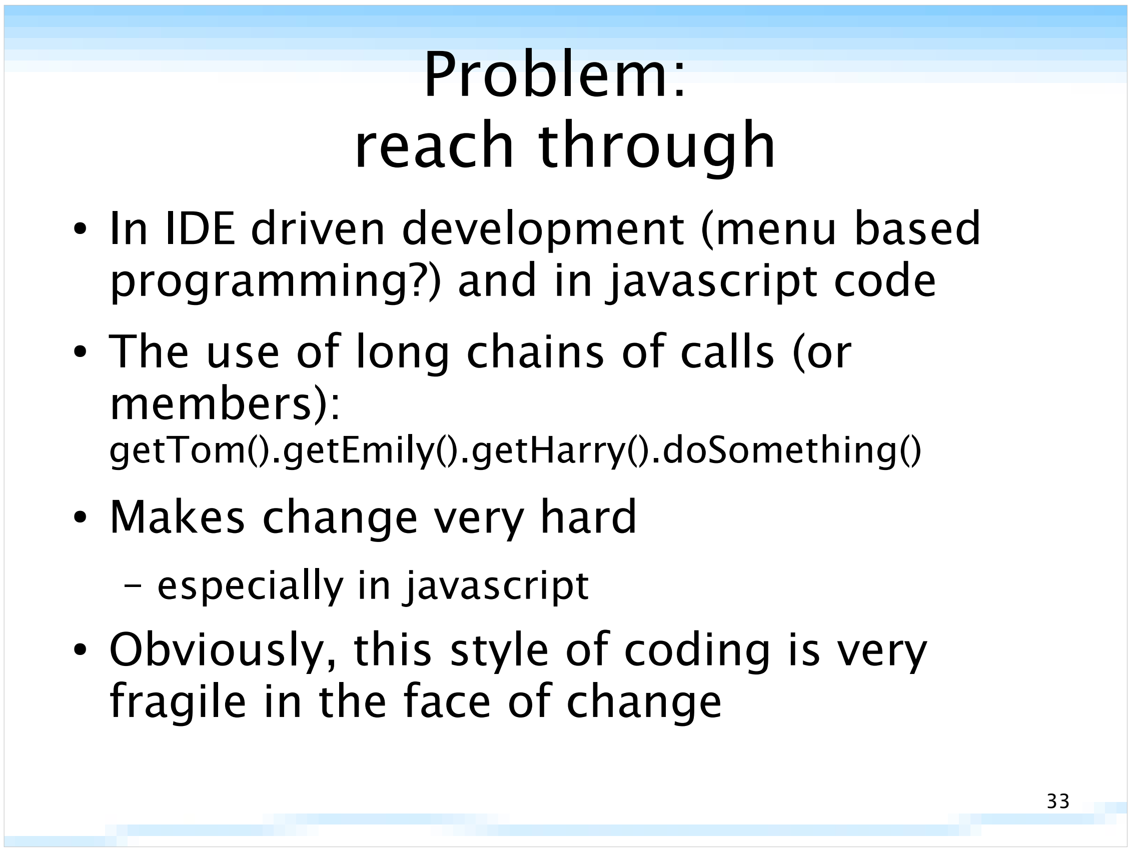
Some examples

- The following example are simple coding or architectural problems I've seen across more than one project
- Whilst not unusual (or maybe it's the code I've worked on) they demonstrate good design principles which have been ignored (or forgotten in the heat of the moment)
- These all are concrete problems (though presented generically) that cause a design to be hard to change

This next section explores some archetypal code issues which are symptoms of badly coupled design (which also demonstrate poor cohesion). These are drawn from a number of sources, some from projects I have worked on, others from online examples.

Each example provides us with an opportunity to explore the good design principles that should be applied to deal with the problem.

Whilst these examples are presented generically I would expect most programmers to have seen these kinds of problems in a design.



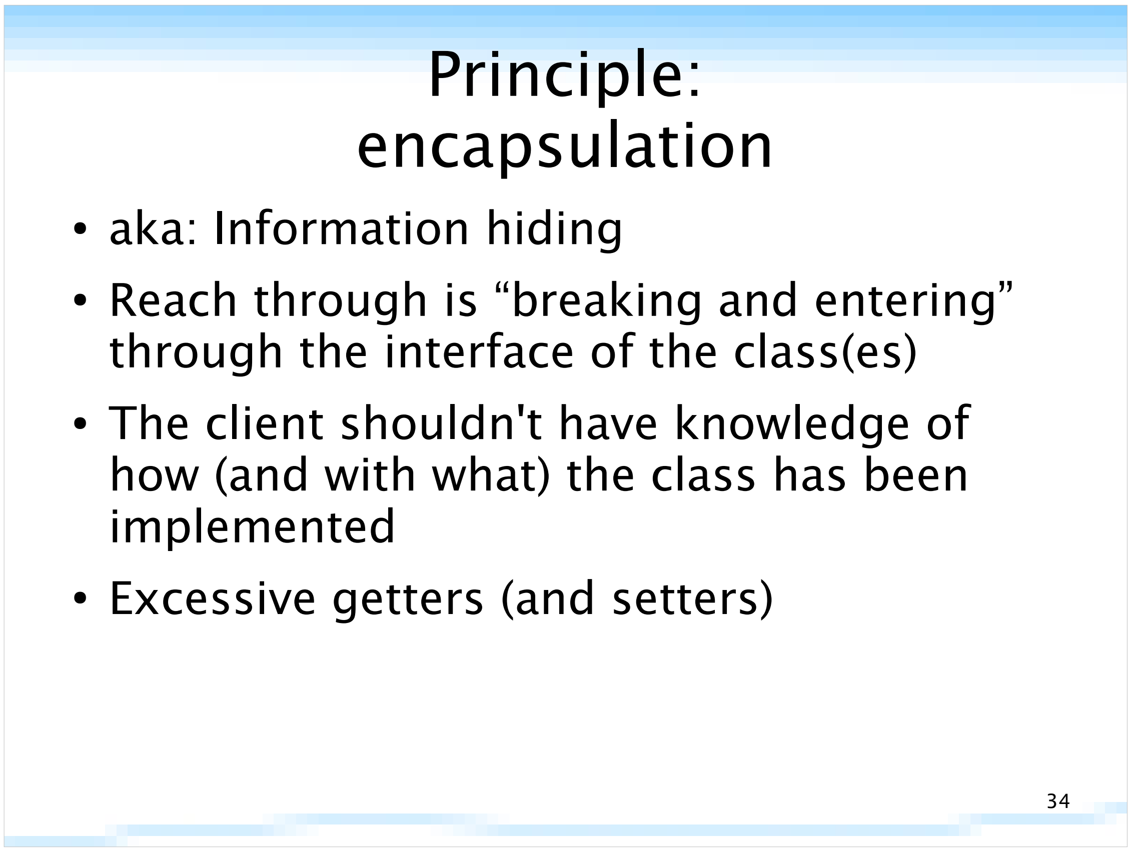
Problem: reach through

- In IDE driven development (menu based programming?) and in javascript code
- The use of long chains of calls (or members):
  getTom().getEmily().getHarry().doSomething()
- Makes change very hard
  - especially in javascript
- Obviously, this style of coding is very fragile in the face of change

The first problem is something often seen in software, sometimes examples in programming literature. It is generally termed *reach through*, where a query method is called on an element returned from a method in a long chain, as above.

This problem can be seen in many languages though typically those with IDE code insight support (allowing a programmer to select from a list of methods the one that looks about right). Also seen in languages with a community that have a tendency to use get methods.

This kind of coding style increases the dependency horizon quickly, in languages without compilation a change to an api may not be noticed for some time. The end result is a design which is hard to change as many dependencies ripple through the design.

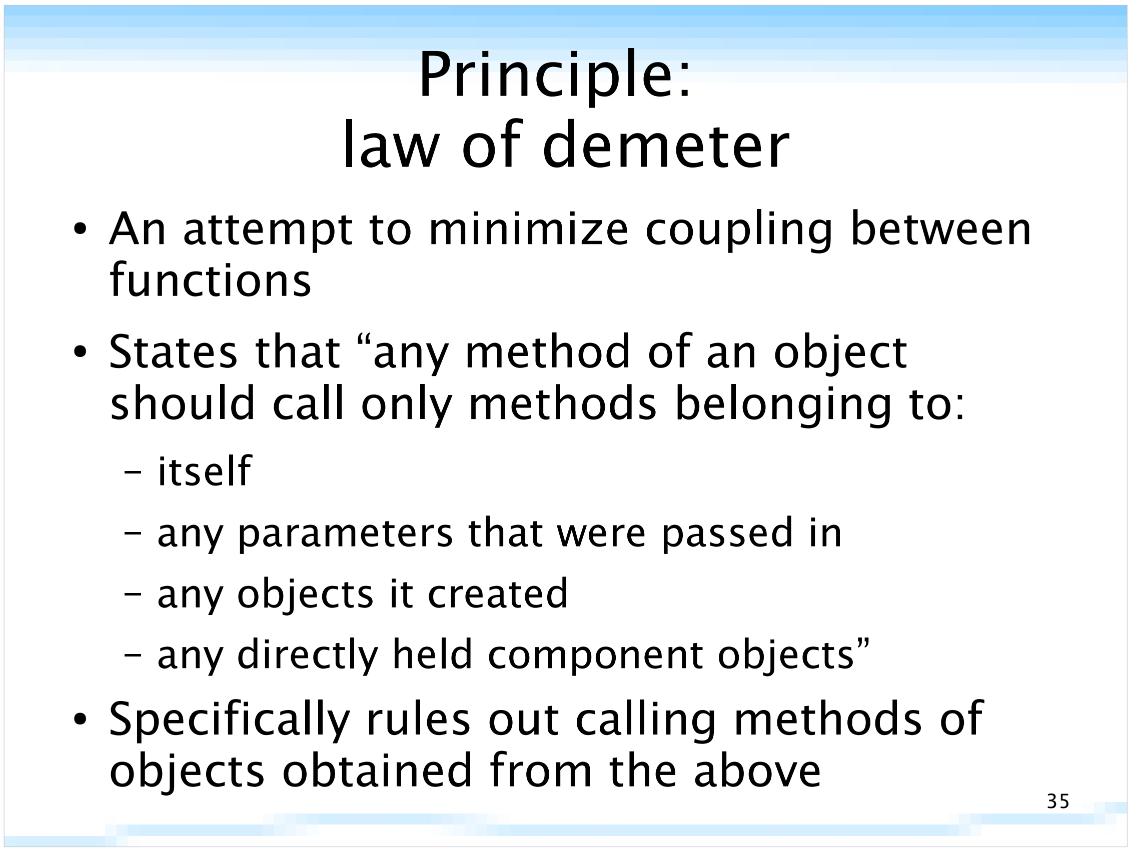


Principle: encapsulation

- aka: Information hiding
- Reach through is “breaking and entering” through the interface of the class(es)
- The client shouldn't have knowledge of how (and with what) the class has been implemented
- Excessive getters (and setters)

We can immediately judge reach through as having flouted the encapsulation principle (or information hiding). The client of the class has been given understanding of exactly how and with what a class has been implemented. The excessive use of get methods leads to this breaking of encapsulation as it is, more often then not, another way of implementing public data.

To some extent abstractions are leaky (see The Law of Leaky Abstractions – Joel on software), but we can minimize reach through by providing the API at the level of abstraction of the class exposing the API. Thus we wrap those dependencies which provide the implementation for an element and protect a client from exposure to these implementation details.



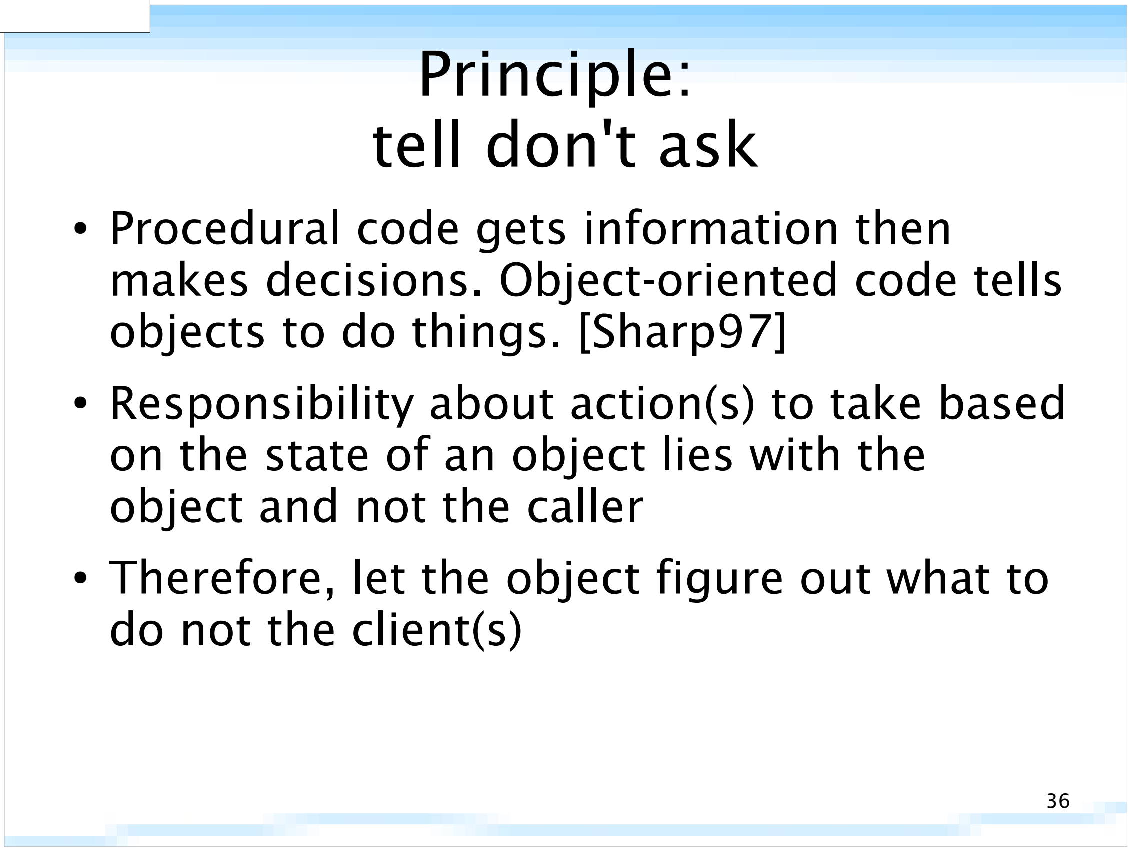
Principle: law of demeter

- An attempt to minimize coupling between functions
- States that “any method of an object should call only methods belonging to:
  - itself
  - any parameters that were passed in
  - any objects it created
  - any directly held component objects”
- Specifically rules out calling methods of objects obtained from the above

Encapsulation can be enforced by using the Law of Demeter, the principle is described above, but the intent is to minimize coupling between functions by restrictions of the use of the elements passed in and used within a method.

In order to implement the law of demeter every class would completely wrap any implementation with a method at the right level of abstraction. The consequence, whilst having minimized coupling, would be methods in the class doing no more than delegating to the original implementation. This is a complication of the design, so we need to be careful to use this principle to minimize coupling where we can usefully protect the design from expected changes.

ClassName

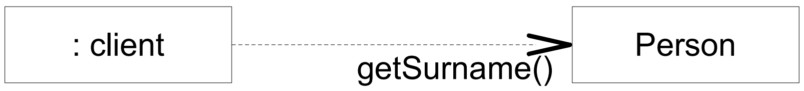


Principle: tell don't ask

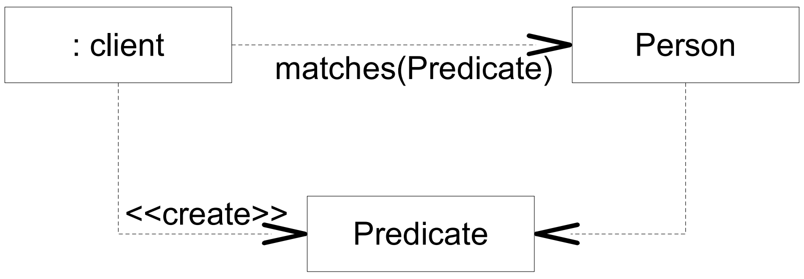
- Procedural code gets information then makes decisions. Object-oriented code tells objects to do things. [Sharp97]
- Responsibility about action(s) to take based on the state of an object lies with the object and not the caller
- Therefore, let the object figure out what to do not the client(s)

36

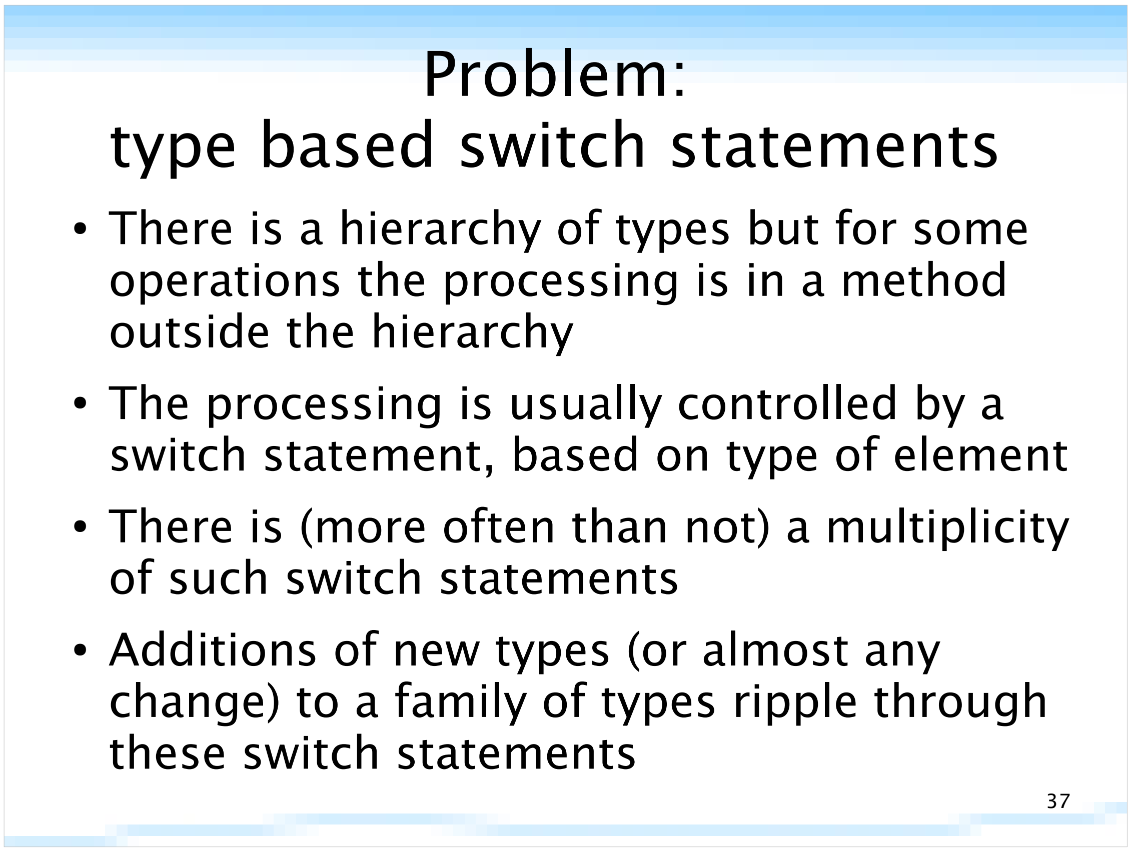
Another approach to enhance encapsulation is to use the principle of tell don't ask, this addresses the tendency to externalise functionality from a class which is not cohesive. The best example would be a search across a collection of Person objects, if such a search was based on the surname, we could use a getSurname() method on Person, so that for each Person object we could compare the search criteria with the value returned. So the class layout would be something like (ignoring the interaction with the collection):



The problem with this style is that the search processing is entirely performed in the client, this does not seem cohesive. Another approach might be to encapsulate the search criteria as a Predicate object, for example:



Thus we have encapsulated the concept of the criteria in the Predicate, we have told the Person object to determine whether it matches the Predicate, we have increased the cohesion of the search processing.



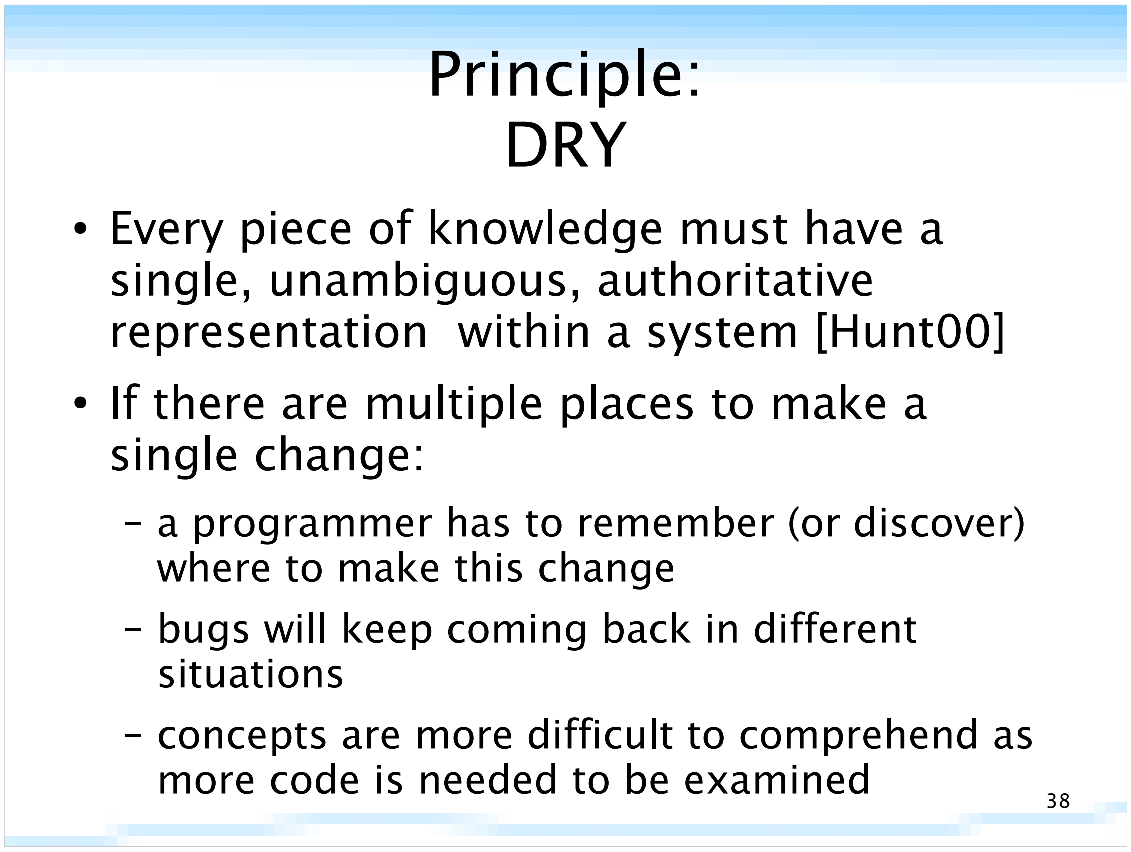
Problem: type based switch statements

- There is a hierarchy of types but for some operations the processing is in a method outside the hierarchy
- The processing is usually controlled by a switch statement, based on type of element
- There is (more often than not) a multiplicity of such switch statements
- Additions of new types (or almost any change) to a family of types ripple through these switch statements

This is another typical breaking of encapsulation seen in code bases, even though there is a viable polymorphic hierarchy, there is some code which operates on the types of this hierarchy contained in switch (or if-else) statements. It is to be expected that if there is one such switch statement, then there are likely to be others.

The obvious problem with such code, is the difficulty of added a new type to the hierarchy, as it is not simply a matter of added the new type, the programmer needs to examine all the switch statements based on the members of the hierarchy to determine how to add the new type.

This style of coding obviously is a break in encapsulation, on encountering such code the following principles can guide us.

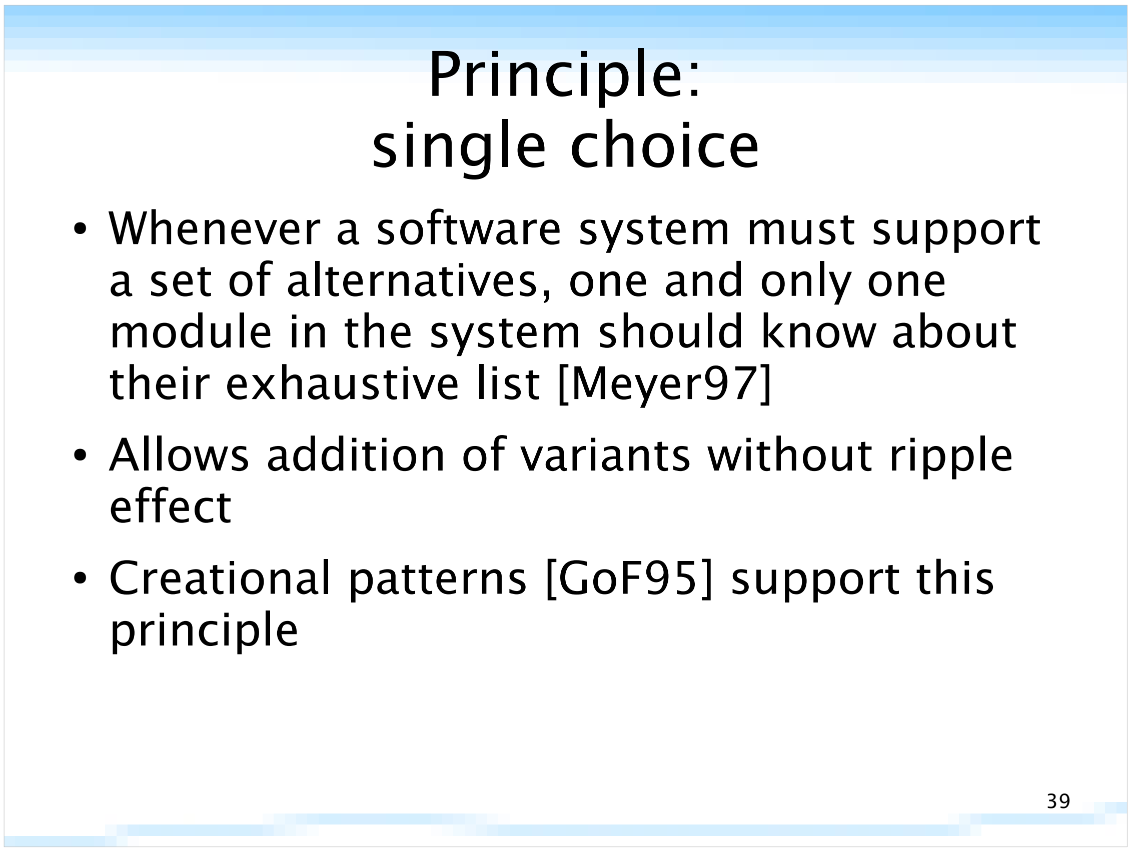


Principle: DRY

- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system [Hunt00]
- If there are multiple places to make a single change:
  - a programmer has to remember (or discover) where to make this change
  - bugs will keep coming back in different situations
  - concepts are more difficult to comprehend as more code is needed to be examined

DRY, simply don't repeat yourself, is a principle to guide us to not to repeat information in a design. The symptoms this addresses are myriad, but the basic problem is that having repeated information places a burden on a programmer. This burden is to locate and modify consistently any repeating information in the code when making a change. This burden is difficult to maintain, and over time a design with repeating information will see the various copies of the same information drift apart. Thus over time the design becomes fragmented and inconsistent, leading to bugs generated solely by these inconsistencies.

To combat this potential inconsistency of copies, we avoid having copies of any information is the design, whether this is a particular piece of processing, or knowledge of the members of a class hierarchy, we encapsulate all this information into a single place in the design. The great benefit of such an approach is the design is much easier to understand and change correctly.



Principle: single choice

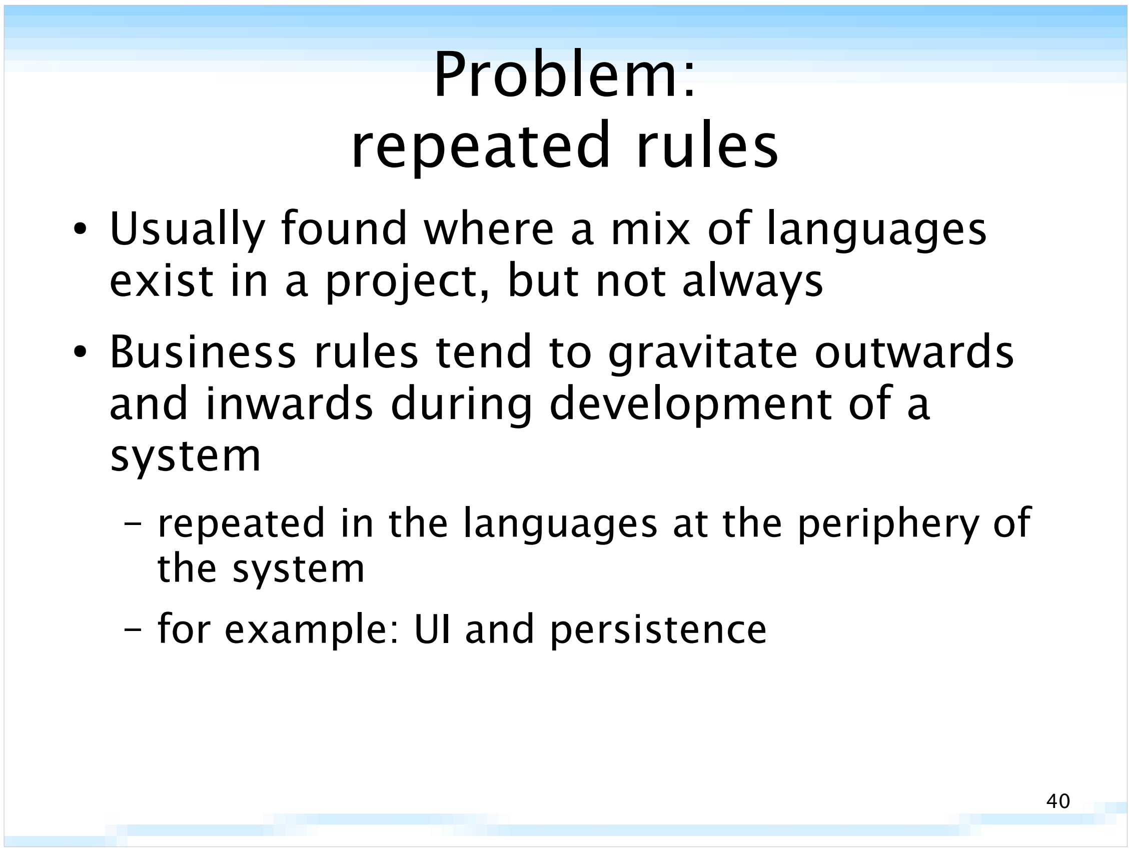
- Whenever a software system must support a set of alternatives, one and only one module in the system should know about their exhaustive list [Meyer97]
- Allows addition of variants without ripple effect
- Creational patterns [GoF95] support this principle

39

Single choice, is another way of stating DRY, in *Object Orientated Software Construction* [Meyer97] this specifically addresses choice. For example, the choice of which class in a hierarchy to use for a particular function. In our example, the knowledge about which class to use is repeated in a number of places in the design.

From the discussion related to DRY, we can see that having repeating information can cause a design to degrade over time as changes occur. Reducing the knowledge of the available classes to a single place, usually creation, and careful allocation of the processing to the members of the hierarchy themselves we can successfully provide this single choice.

The advantages of such a design is the ease of adding a new member of the hierarchy, reduced to adding the new class, and adding the creation of this instance. A good way to handle this creation would be to use one of the creational patterns [GoF95].



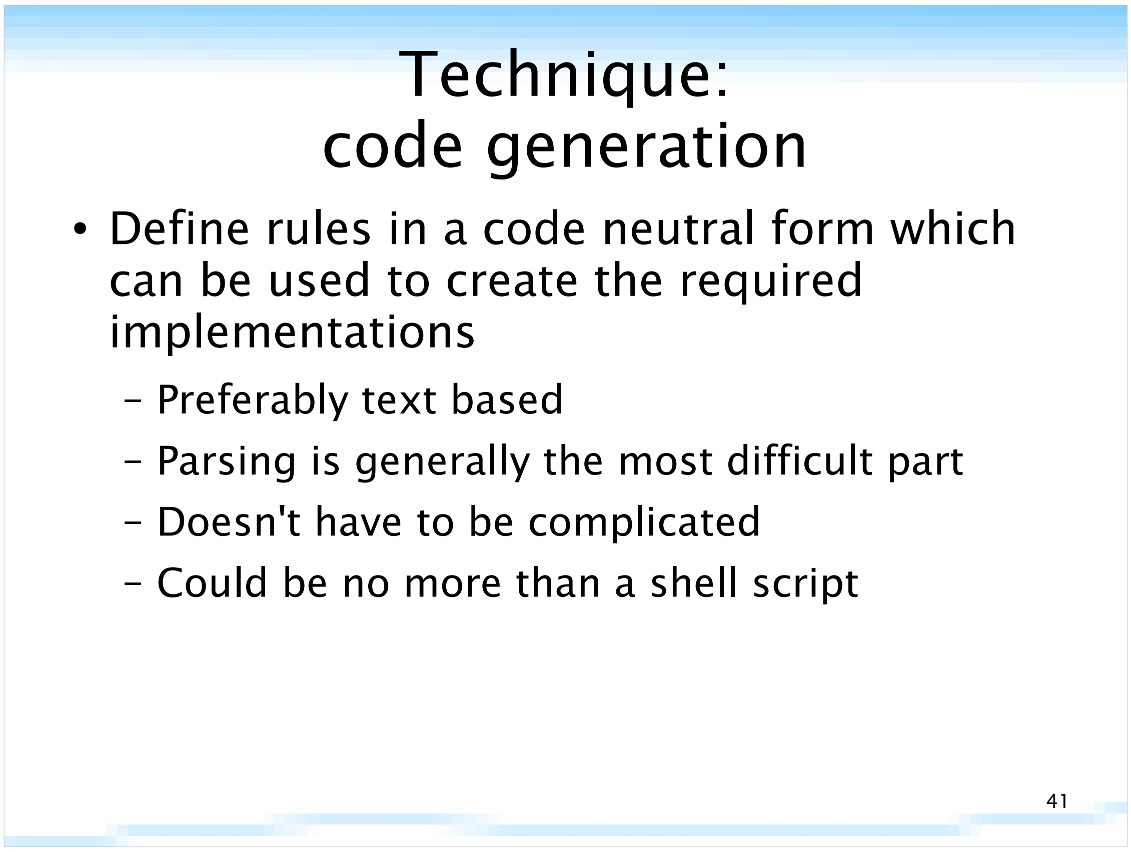
Problem: repeated rules

- Usually found where a mix of languages exist in a project, but not always
- Business rules tend to gravitate outwards and inwards during development of a system
  - repeated in the languages at the periphery of the system
  - for example: UI and persistence

Another form of repetition in designs is when a design is made up of more than one language, for example a database backend with SQL, a server component in C++ or Java, and a web front end in Javascipt. This kind of language mix is not unusual, but it leads to a particular kind of repetition.

This repetition is around the business rules of the domain concepts, which tend to gravitate to the external parts of the system, for example the database and the UI. The need for consistency in the persistence layer and the need to increase responsiveness of the UI to user entry errors (and increased scalability of any server components) are the drivers for this.

The repetition comes from the need for these business rules to be provided in each component in the language of that component. The problem is that the rules are repeated, and as they are reimplemented in each language tend to be inconsistent, or become so over time.



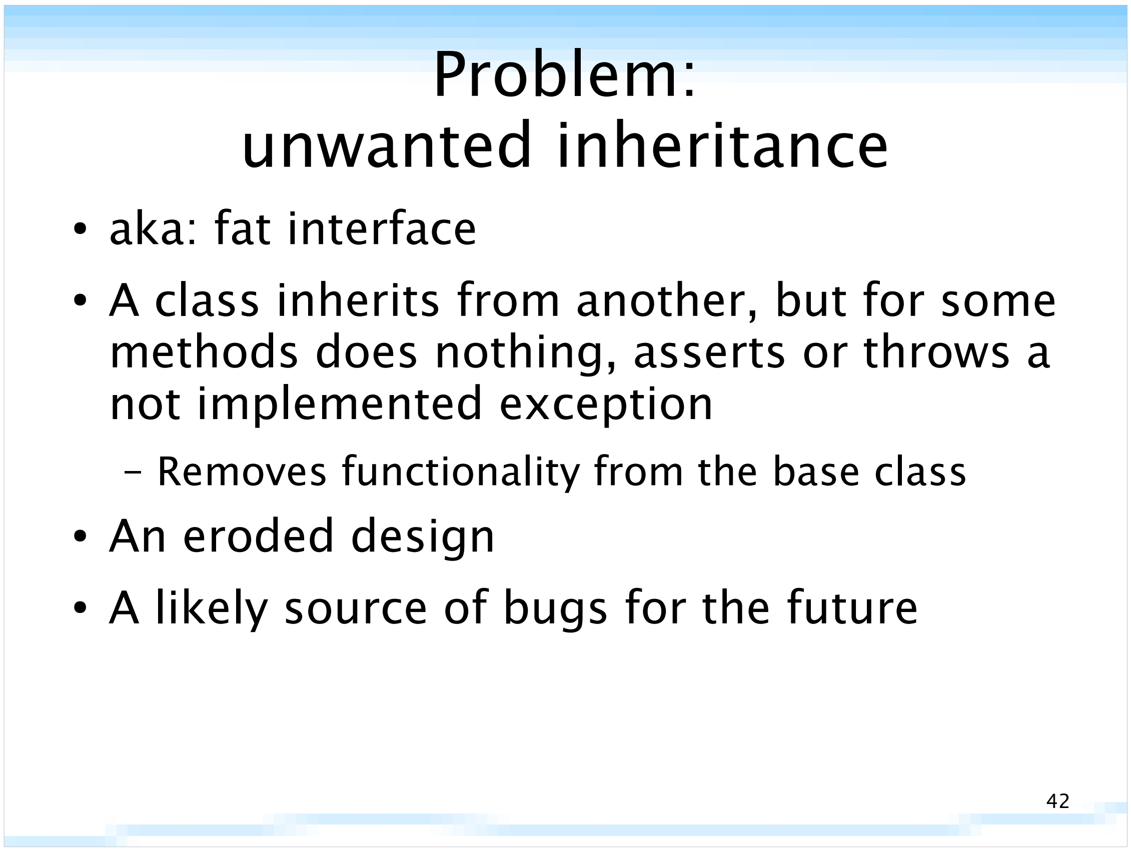
Technique: code generation

- Define rules in a code neutral form which can be used to create the required implementations
  - Preferably text based
  - Parsing is generally the most difficult part
  - Doesn't have to be complicated
  - Could be no more than a shell script

A simple technique we can use to address the repetition of business rules is to generate the code for each language from some other representation, i.e. a code neutral form. The *Pragmatic Programmer* [Hunt00] suggest that a simple text based format is preferable to any other, as it can be read by both a program and a programmer without additional tools.

In such an approach, complexity should be avoided, the aim is to reduce the burden of knowledge on a programmer and the inconsistency of the design.

In this category, Google's GWT is a good example of providing a single source for 2 languages, though this is only targeted at the server component and the UI.

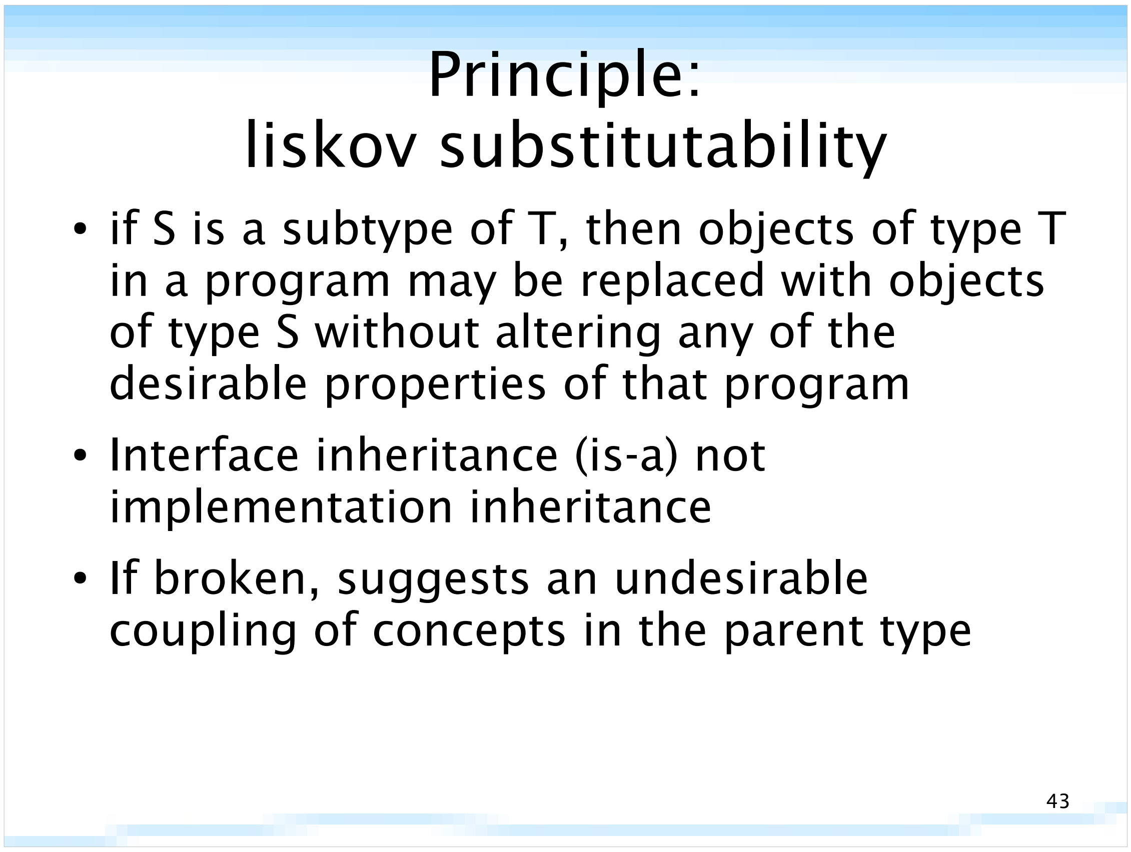


Problem: unwanted inheritance

- aka: fat interface
- A class inherits from another, but for some methods does nothing, asserts or throws a not implemented exception
  - Removes functionality from the base class
- An eroded design
- A likely source of bugs for the future

Another problem likely to be seen in a design is that of unwanted methods provided by an interface or abstract type to a realisation of that type. This is visible in a design where a subtype either provides no implementation of a function, asserts or throws an exception which states, “do not call”, or “not implemented”. The basic flaw is that the subtype is removing functionality of the base class.

The source of such code is the likelihood of the base class being more than a single concept, or a fat interface, it has too much responsibility. The subtype does not want to provide the implementation for all the responsibilities of the base class, and thus we see the artefacts of removed methods in the design.



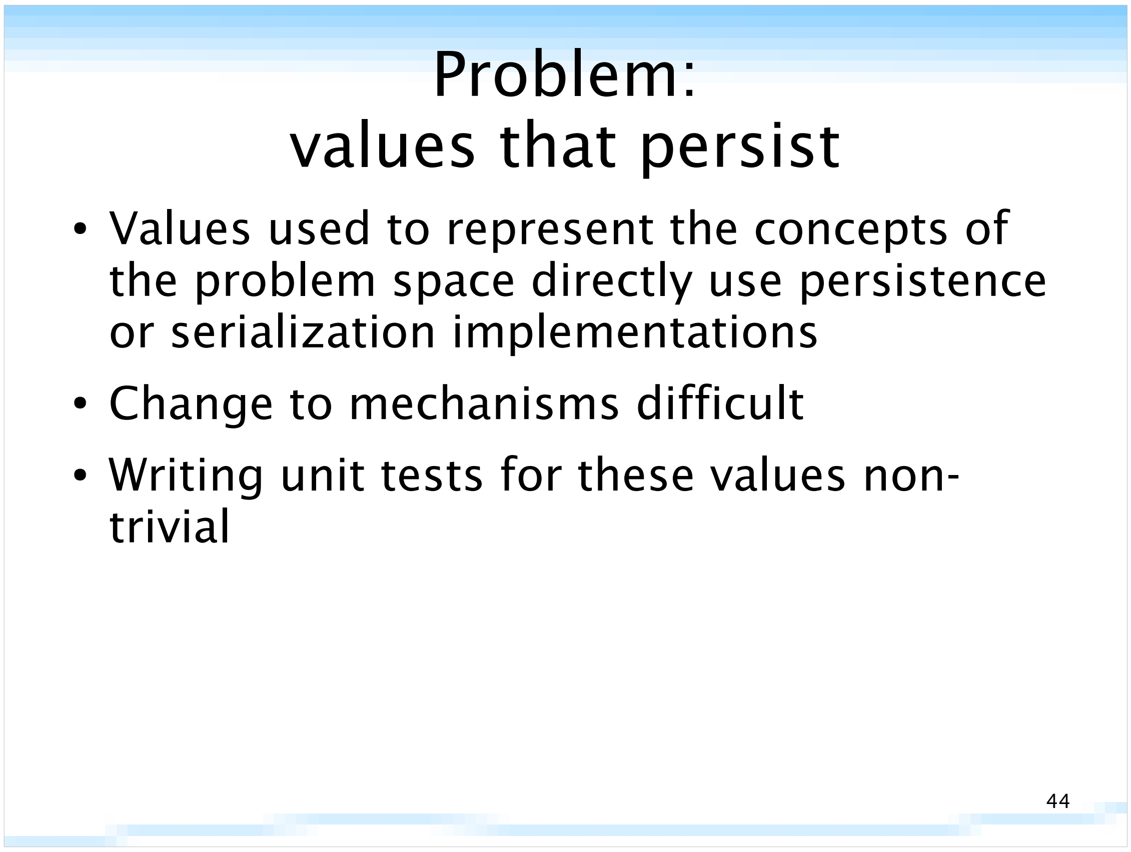
Principle: liskov substitutability

- if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program
- Interface inheritance (is-a) not implementation inheritance
- If broken, suggests an undesirable coupling of concepts in the parent type

The problem exhibits a major problem the subtype is not a subtype of the base class at all. It cannot be used in place of the base class without altering the behaviour of the program. This is a breakage of the Liskov Substitutability principle, which states (paraphrased): “using a sub type T of S in place of S has no impact on the functionality of a program”.

Thus a subtype has responsibilities to exactly meet the contract of the parent type, which means not just the interface of the parent type, but also any boundaries, invariants, and error behaviour (i.e. the subtype should throw no additional exceptions). Of course, this does not mean the subtype cannot add additional behaviour, but it must meet the contract of the parent.

As stated above, if the subtype cannot meet the contract of the parent, the parent must be examined for possible combination of concepts which, of course, should be separated.

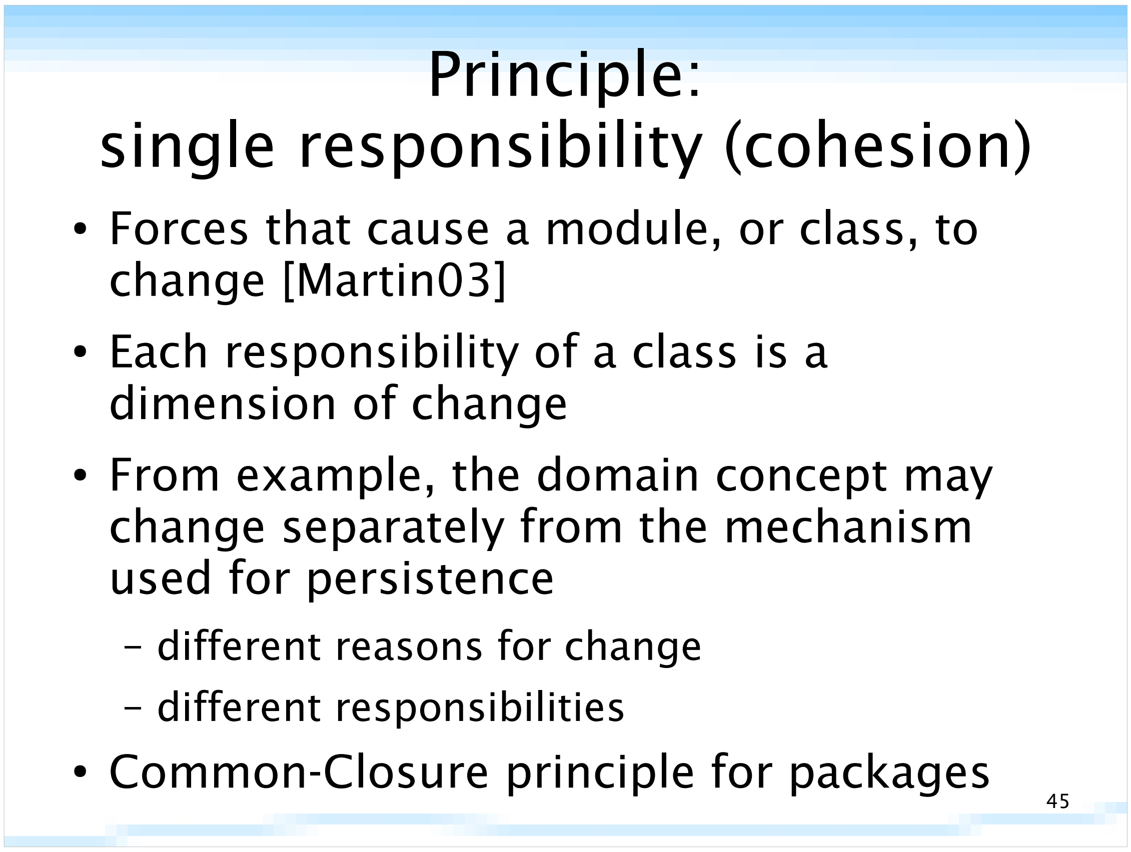


Problem: values that persist

- Values used to represent the concepts of the problem space directly use persistence or serialization implementations
- Change to mechanisms difficult
- Writing unit tests for these values non-trivial

Another problem seen in some designs is for classes that represent domain concepts to be coupled to some form of persistence mechanism. This is not unusual in smaller projects which have grown over time as this coupling is not seen as a problem in a small project. Of course this coupling to the persistence mechanism can be very difficult to change at a later date.

The most difficult problem is that any tests written for these domain concept classes require the persistence framework to be included, which increases the burden on the build of the tests.

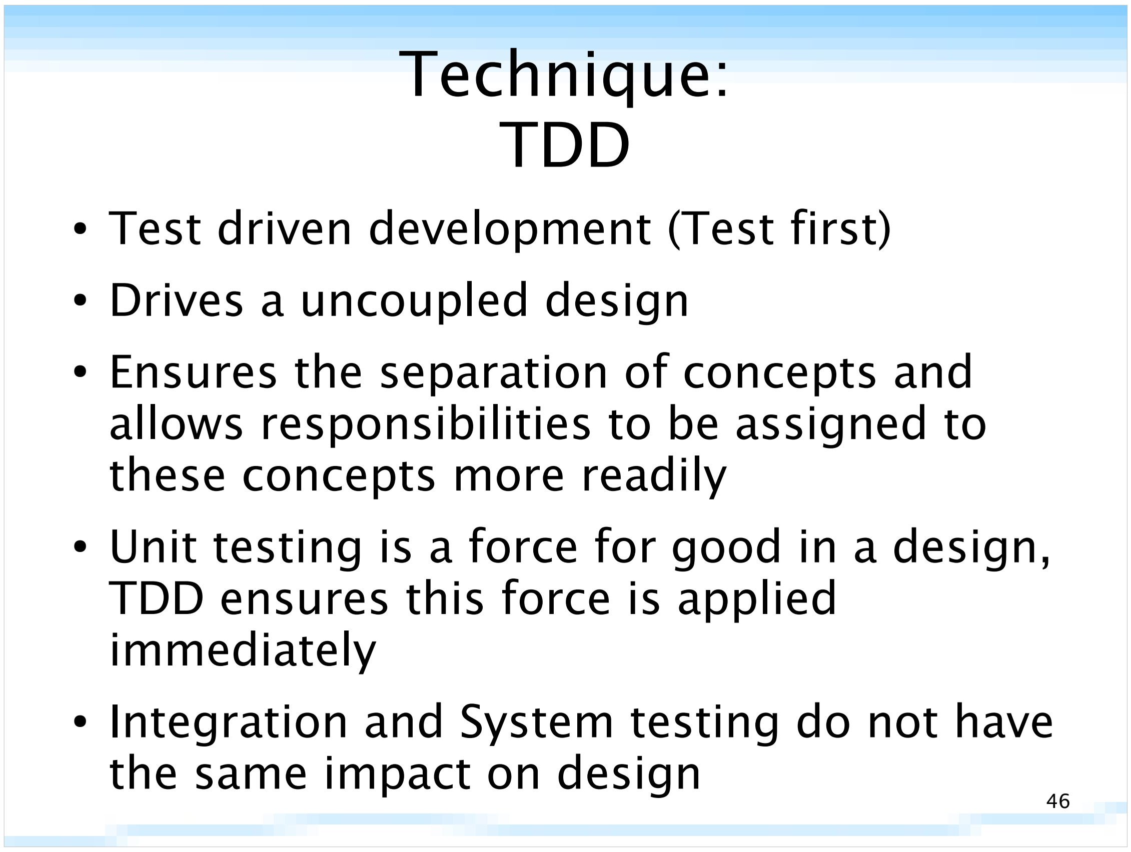


Principle: single responsibility (cohesion)

- Forces that cause a module, or class, to change [Martin03]
- Each responsibility of a class is a dimension of change
- From example, the domain concept may change separately from the mechanism used for persistence
  - different reasons for change
  - different responsibilities
- Common-Closure principle for packages

In *Agile Software Development* [Martin03] discusses the single responsibility principle as being the forces the cause a module or class to change. A class should only contain a single responsibility, i.e. a single dimension of change. Thus a coupling of a domain concept with the persistence mechanism is a coupling of responsibilities, the domain concept itself and the persistence of concept. Such a coupling in a single class produces a class with more than a single dimension of change, as the domain concept may change separately from the persistence mechanism, thus this class should be changed to separate the persistence mechanism, or this mechanism should be encapsulated by an interface.

This is similar to another principle applied at the package level, the common closure principle, which states that classes in a package should change for the same reason. For example if the persistence mechanism changes in a design, the impact should be contained to a single package.



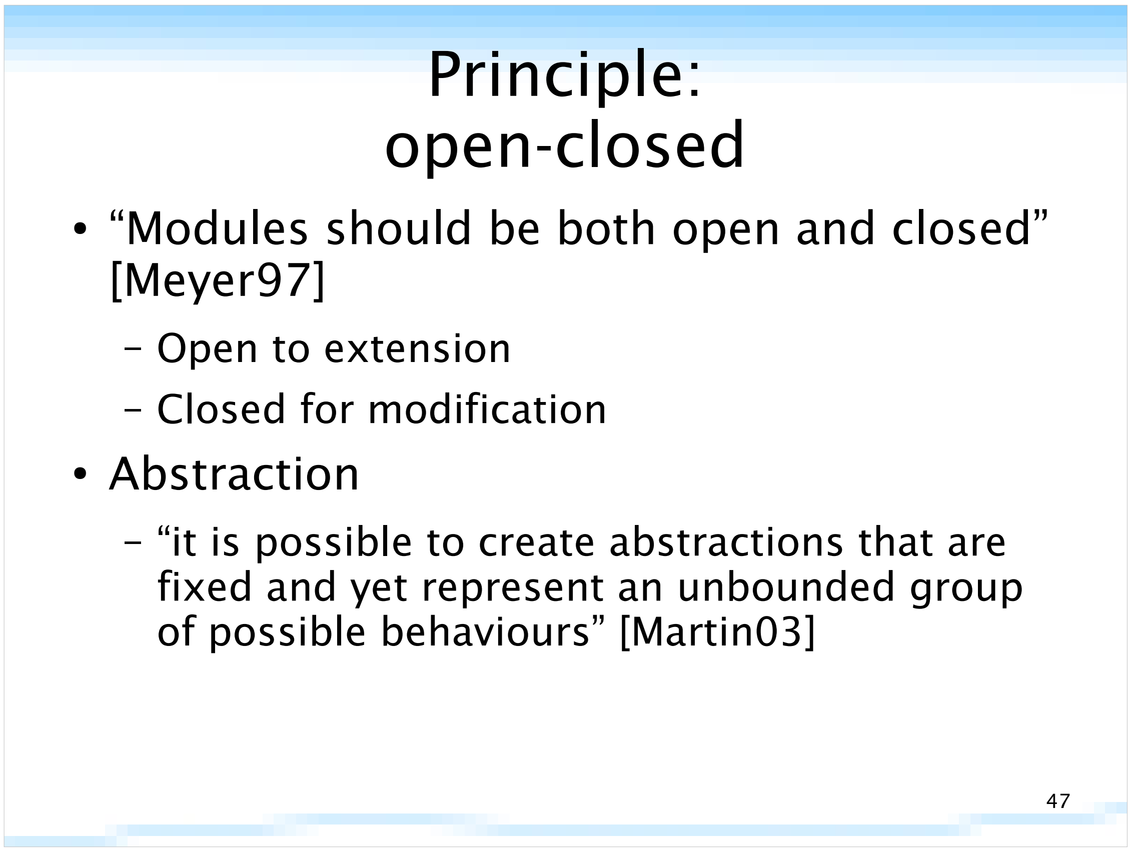
Technique: TDD

- Test driven development (Test first)
- Drives a uncoupled design
- Ensures the separation of concepts and allows responsibilities to be assigned to these concepts more readily
- Unit testing is a force for good in a design, TDD ensures this force is applied immediately
- Integration and System testing do not have the same impact on design

A simple technique for driving a decoupled design is TDD (Test driven development), as we drive the design with tests. Thus if we provide tests for the domain concepts we would not expect to introduce the persistence mechanism in these classes, as this would increase the burden on the tests. We would expect to introduce an interface for persistence which would allow the tests to mock the persistence mechanism to allow the tests to be written quickly.

The impact of TDD cannot be anything other than a force for good in a design as it strongly encourages separation of concepts in a design, the use of interfaces to allow mocking of concepts not being tested by a test, etc. The end result is a decoupled design, which will be more adaptable and will respond to change more readily.

Integration and system testing do not have the same impact on a design, only unit testing can provide such benefits. Though in the absence of any testing, introducing automated systems tests will be valuable.



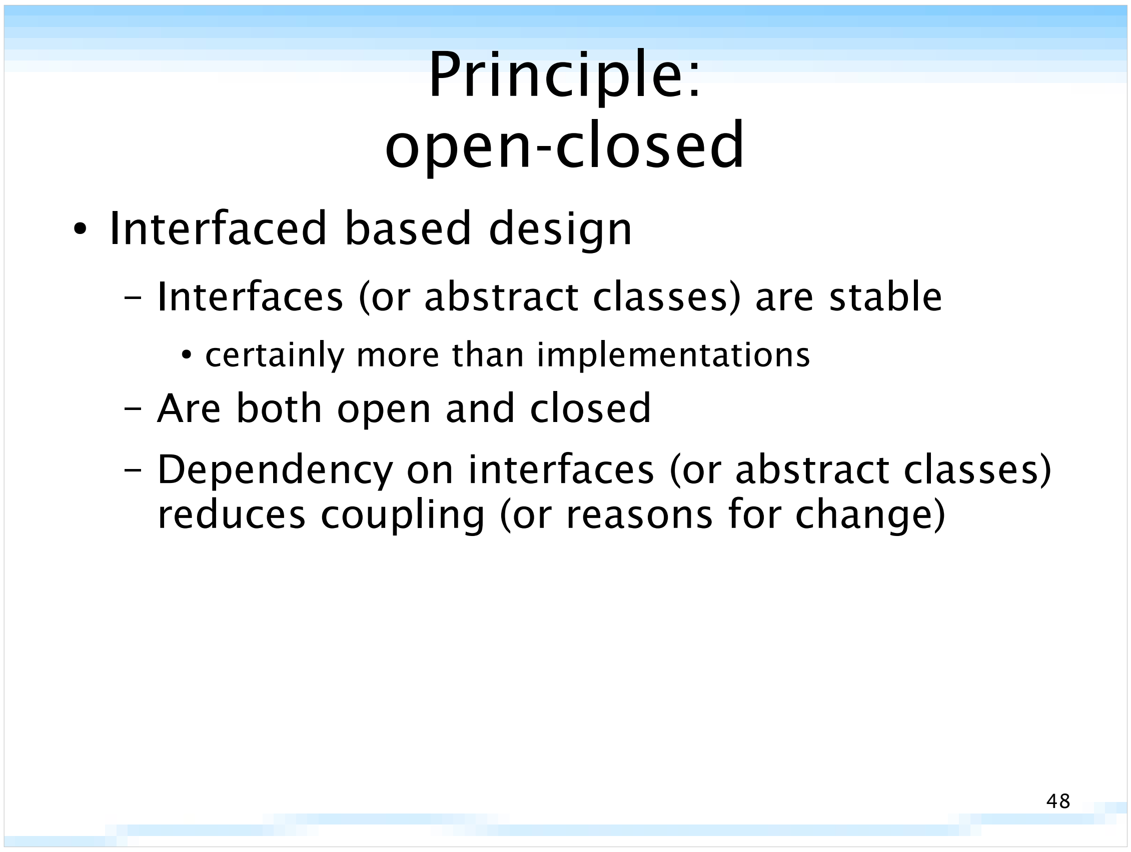
Principle: open-closed

- "Modules should be both open and closed" [Meyer97]
  - Open to extension
  - Closed for modification
- Abstraction
  - "it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviours" [Martin03]

A principle related to the reduction of the coupling of concepts is the open-closed principle, though this is not immediately obvious. In essence the open closed principle states 2 requirements which are seemingly at odds with each other, open to extension (allowing additional behaviour), whilst at the same time closed for modification (allowing no change in behaviour).

The intention of this principle is to reduce (or eliminate) changes to existing behaviour, this being considered at the interface of a class. Thus we should not change the existing methods of a class, but are free to add additional methods. In keeping with this principle we are reducing the impact of change, as existing code using existing methods will not change as we add new behaviour.

However this approach is restrictive, and may lead to many overloaded methods. Thus in *Agile Software Development* [Martin03] the focus is on the interface, and the abstraction represented by the interface. An interface and realisations represent a fixed concept and a range (unbounded as new realisations can be created) of behaviours. Thus the use of an abstract type (or interface) provides a convenient mechanism for the delivery of the open closed principle.



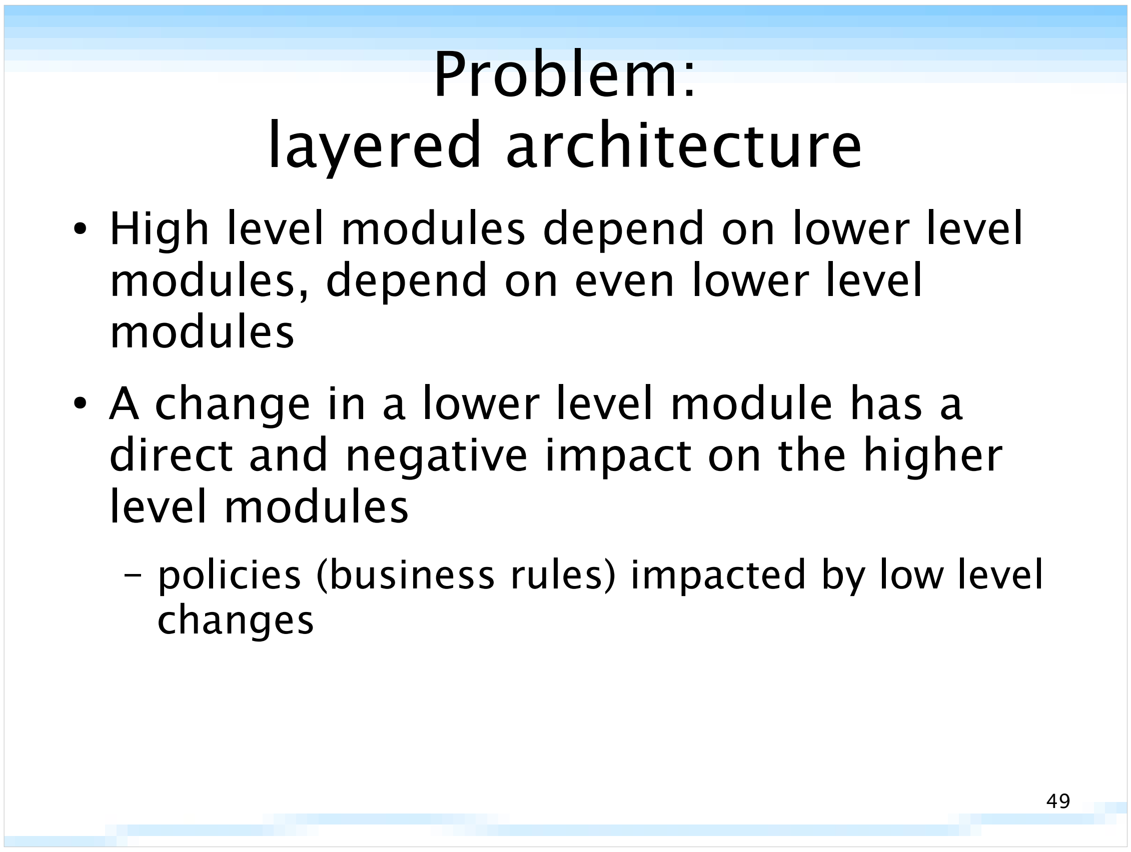
Principle: open-closed

- Interfaced based design
  - Interfaces (or abstract classes) are stable
    - certainly more than implementations
  - Are both open and closed
  - Dependency on interfaces (or abstract classes) reduces coupling (or reasons for change)

Using abstract types (or interfaces) to provide the open closed principle gives us 2 benefits in a design:

Abstract types (or interfaces) are more stable then an implementation of the interface, thus dependency on an interface (compared to an implementation) reduces the impact of change.

Abstract types (or interfaces) are both open and closed, the abstract type represents a bounded concept, which is unchanging and is cohesive. The implementations of this abstract type provide us with the opportunity to extend the functionality of the abstract type (keeping in mind Liskov) without any change to the abstract type and the clients which depend on it.



Problem: layered architecture

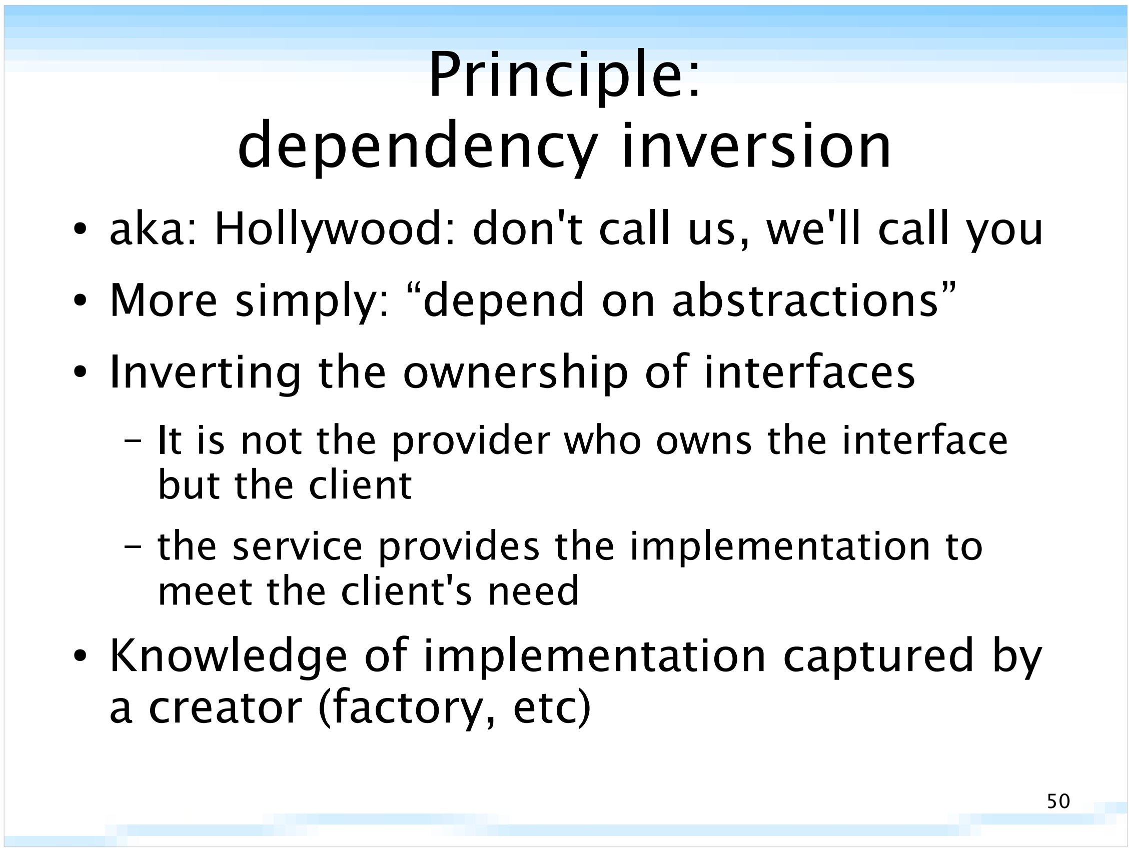
- High level modules depend on lower level modules, depend on even lower level modules
- A change in a lower level module has a direct and negative impact on the higher level modules
  - policies (business rules) impacted by low level changes

49

Traditionally literature describes software architecture as a series of layers (like lasagne) with the high level concepts in the top layer, with details beneath, and so on. It is not unusual to see designs implemented in this way, but there is a major drawback to such a design, the high level concepts depend on the implementation details which are apt to change. Such a design is unstable by its very nature, due to the dependencies on the detailed implementation by the domain concepts.

A more recent take on software architecture places the domain concepts at the code of a design, with the implementation details, such as persistence, on the outside. Designs based on this arrangement tend to be better than those based on the those based on the layered architecture.

To have the best possible design (those with the least coupling to implementation detail and most adaptable to change) we need to consider the following principles.



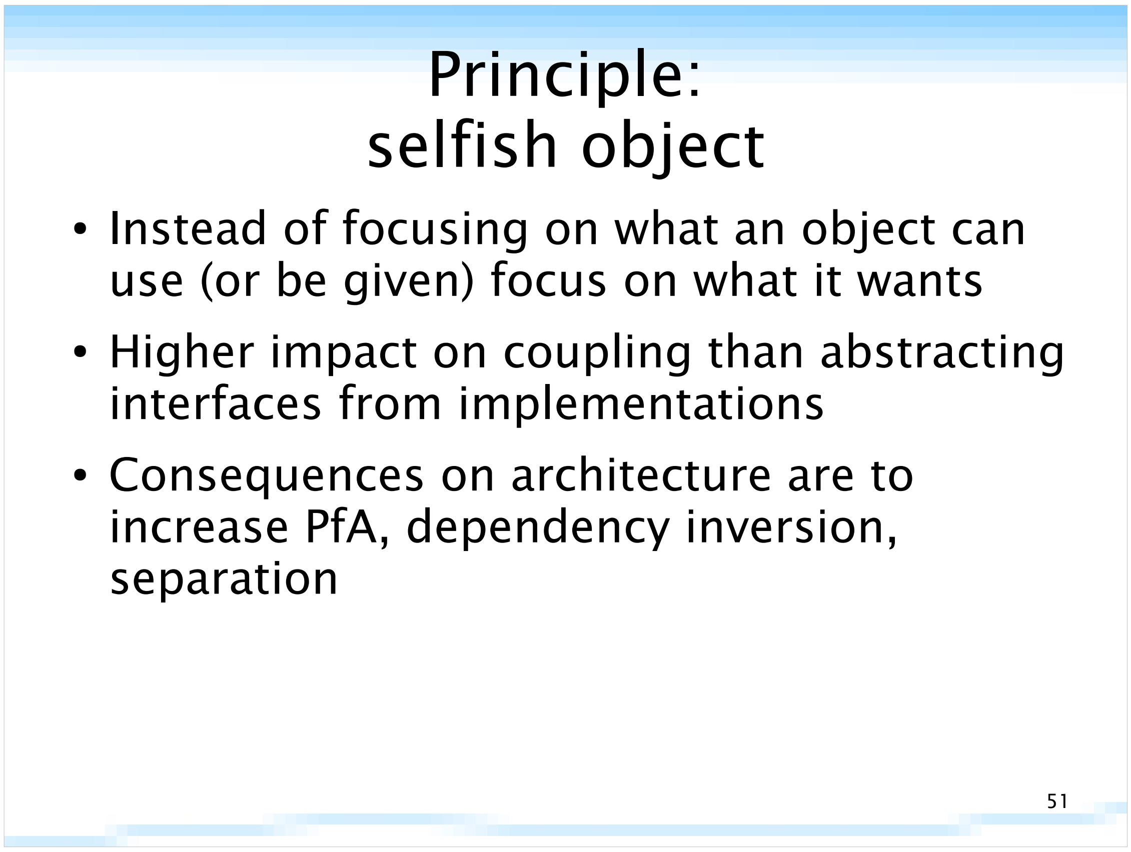
Principle: dependency inversion

- aka: Hollywood: don't call us, we'll call you
- More simply: “depend on abstractions”
- Inverting the ownership of interfaces
  - It is not the provider who owns the interface but the client
  - the service provides the implementation to meet the client's need
- Knowledge of implementation captured by a creator (factory, etc)

This session has hinted at this principle earlier, but the dependency inversion principle, helps us to control the coupling of high level concepts on detailed implementation. We provide an abstract of the detailed implementation, though an interface, but we do this form the point of view of the client. The client owns the interface, which states what services it requires, and the service (the implementation details) provides the realisation.

Thus the client is dependent on a interface and not implementation, which has been discussed earlier. The main concept here is that the interface is not owned by the implementation, interfaces generated from an implementation tend not to be as suitable as one generated from the point of view of the client. The interface is owned by the client, the implementation provides the service in terms required by the client, this is source of the inversion.

The client, does not necessarily need to know anything about the implementation of the interface, as this can be handled using a creational pattern [GoF95] or by Parametrization from Above (PfA) where the implementation of the interface is passed to the client. This approach is the basis of dependency inversion frameworks, where the information about the implementation of interfaces is maintained outside the code in some text form and the wiring of the implementations is provided by generating code from this representation.



Principle: selfish object

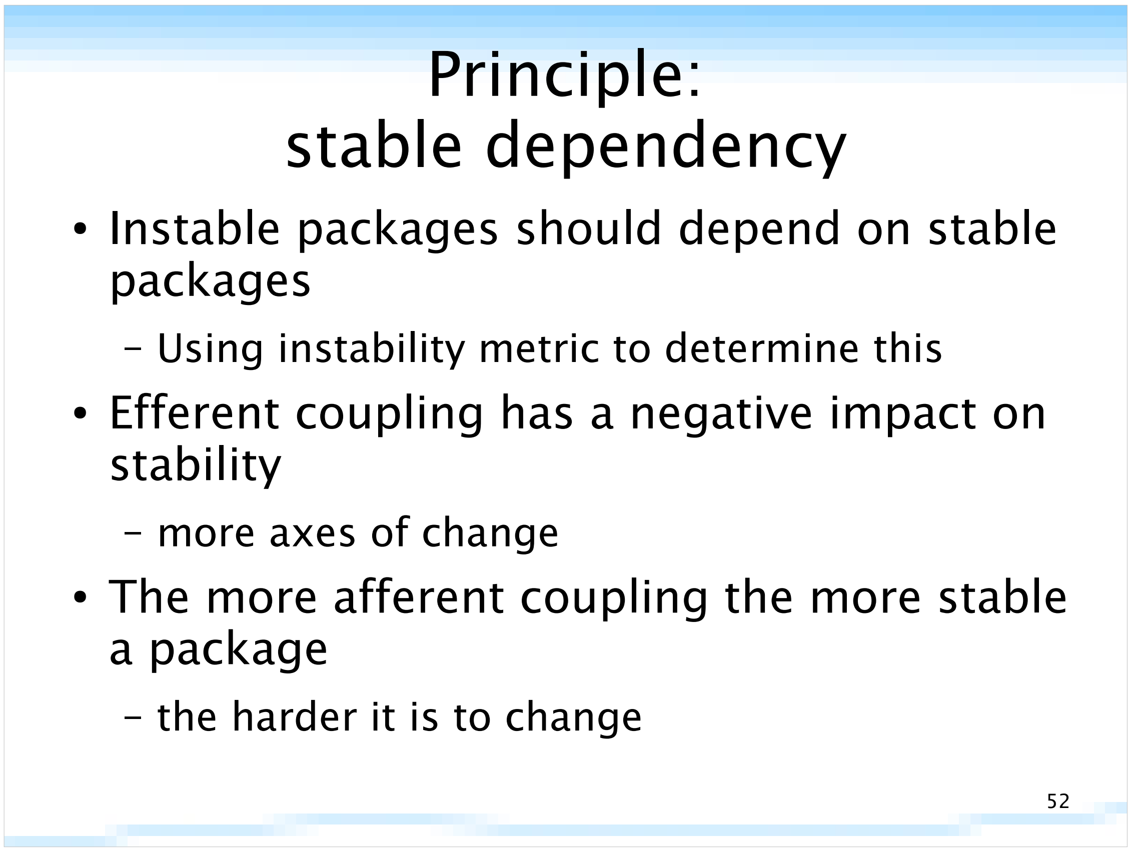
- Instead of focusing on what an object can use (or be given) focus on what it wants
- Higher impact on coupling than abstracting interfaces from implementations
- Consequences on architecture are to increase PfA, dependency inversion, separation

51

Kevlin Henney's presentation of the selfish object, covers much of the same ground as the dependency inversion, but the term is much more compelling. The object in question states its needs to the service in the form of an interface, selfishly demanding the service meet the contract it defines.

As stated above, this approach to defining services required by objects, rather than those provided by the implementation, tends to exhibit a much better cohesion (or focus). It improves the ability to unit test and design by test (TDD) and also exhibits increase use of PfA.

Finally, as stated before, this approach produces designs with low coupling because of the focus on the dependency on interfaces, rather than implementations.



Principle: stable dependency

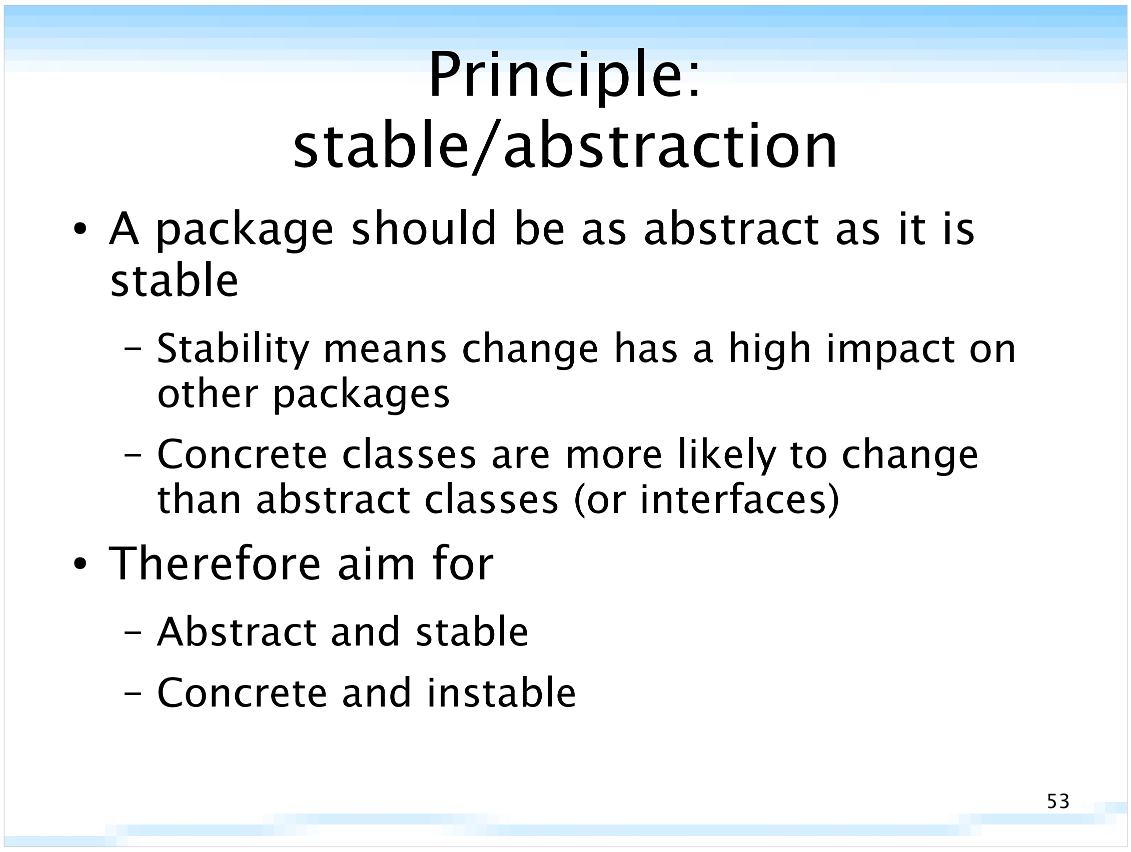
- Instable packages should depend on stable packages
  - Using instability metric to determine this
- Efferent coupling has a negative impact on stability
  - more axes of change
- The more afferent coupling the more stable a package
  - the harder it is to change

52

Another principle that can be applied to control dependency in an architecture is the stable dependency principle, which requires that packages should always depend on more stable packages. Thinking back to the metrics discussed earlier in the presentation, instability was a measure of the incoming compared to the outgoing dependencies. We know that efferent coupling has a negative impact on stability as there are more reasons for a package to change if it has many dependencies.

This principle re-enforces the use of interfaces in a architecture, interfaces will generally only depend on other interfaces, abstract types or value types, thus will have low efferent coupling. If a package depends on another package which contains interfaces then this dependency will be towards something more stable, in terms of both the metric and the concepts in the package.

Thus as described in dependency inversion and selfish object, we can reduce coupling with the introducing of interfaces, from the clients requirement on the service. These interfaces are stable and the dependency from both the client and the service are to the interfaces, thus if the client was more unstable than the service the dependencies, after the introduction of the interfaces, the dependencies would be to something more stable.



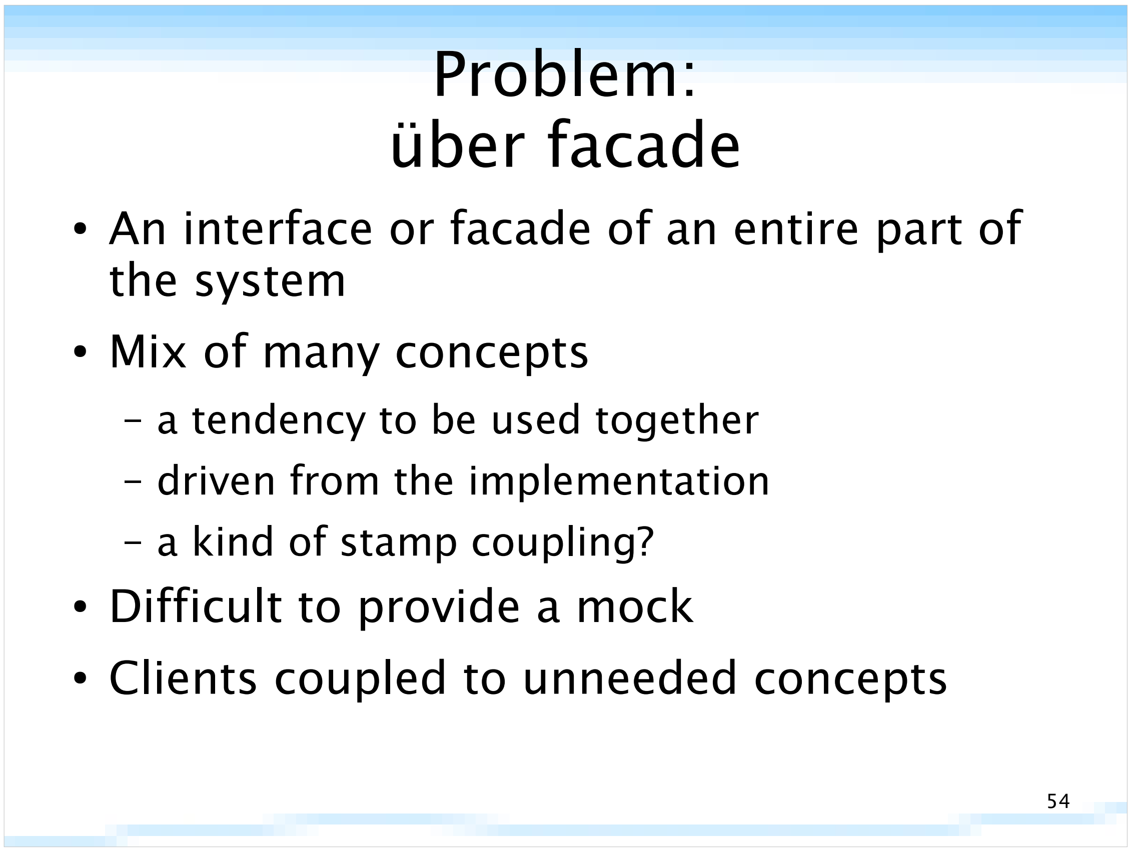
Principle: stable/abstraction

- A package should be as abstract as it is stable
  - Stability means change has a high impact on other packages
  - Concrete classes are more likely to change than abstract classes (or interfaces)
- Therefore aim for
  - Abstract and stable
  - Concrete and instable

Another principle with a high impact in handling dependencies in an architecture is the stable/abstraction principle, thus if a package is highly stable (i.e. it has a high afferent coupling) then it should also be mainly abstract. This would be a consequence of the previous dependency inversion, selfish object and the stable dependency principles.

This principle focuses on the impact of a change in a package which has many afferent dependencies, which would impact a large number of other packages. As concrete classes are more likely to change than abstract classes or interfaces, the more a package with large afferent coupling is abstract the less likely it will change and impact other packages.

The other consequence of this principle is that we aim to make packages which are instable as concrete as they are instable.

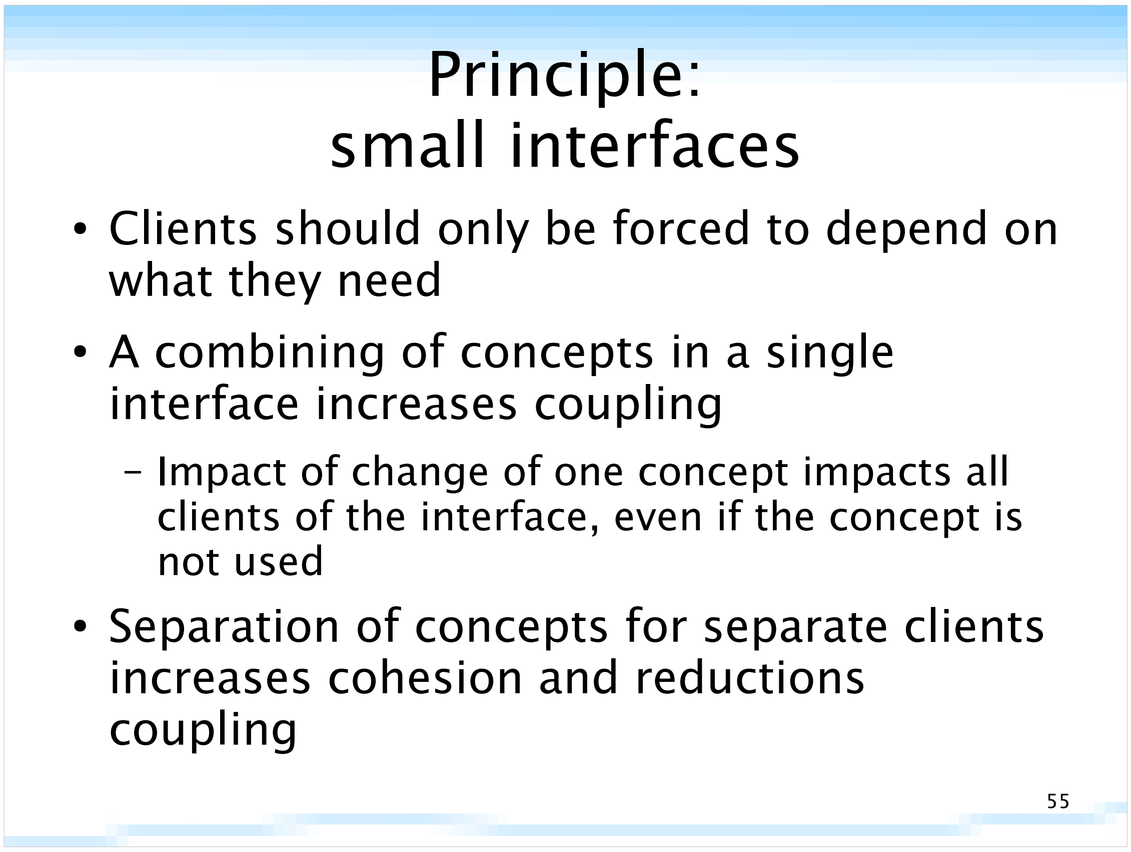


Problem: über facade

- An interface or facade of an entire part of the system
- Mix of many concepts
  - a tendency to be used together
  - driven from the implementation
  - a kind of stamp coupling?
- Difficult to provide a mock
- Clients coupled to unneeded concepts

It is not unusual to see a facade, or even an interface, which encapsulated an entire subsystem of a design. This is intended to provide to minimize dependency on the implementation of the subsystem and thus coupling in the design general. Unfortunately as the design changes and grows, the number of concepts introduced into the facade grow, thus any client requiring a small number of services from the facade have to depend on everything that the facade provides. This in fact increases the coupling of the design, as the number of reasons for a client to change are increased by the use of such a large facade.

As such a facade grows, the ability to mock the implementation for unit testing purposes becomes more difficult, ultimately leading to an inability to test the design sufficiently.



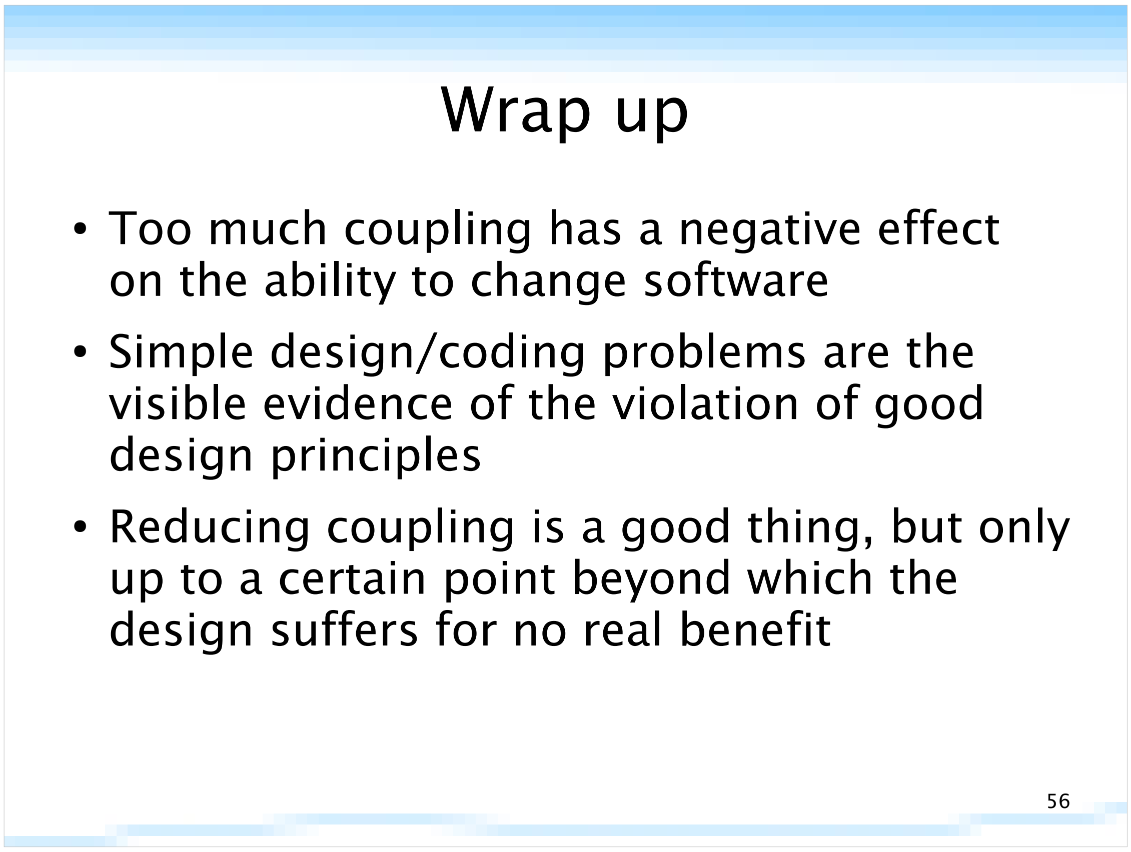
Principle: small interfaces

- Clients should only be forced to depend on what they need
- A combining of concepts in a single interface increases coupling
  - Impact of change of one concept impacts all clients of the interface, even if the concept is not used
- Separation of concepts for separate clients increases cohesion and reductions coupling

This principle is very similar to the selfish object, in that a client should only depend on what it needs, in fact it is best to state this need by publishing the interface it wants, such an interface would be small, by its nature as it is focused on the need of the client.

A facade or interface which has many concepts will inevitably lead to high coupling in a design, as the impact of change such an artefact will be widespread in the design.

Separation of concepts into interfaces which support only those concepts will reduce the coupling in a design whilst at the same time increasing the cohesion of these interfaces.



Wrap up

- Too much coupling has a negative effect on the ability to change software
- Simple design/coding problems are the visible evidence of the violation of good design principles
- Reducing coupling is a good thing, but only up to a certain point beyond which the design suffers for no real benefit

We have discuss the impact of bad coupling on a design, what it is, how to measure it and to some extent how to fix it, and where to focus effort in a coupled design. We have spent a good amount of the presentation looking at design problems as exhibited by seemingly simple code issues and we have discussed the principles that we should consider as we make changes to the design to decrease the coupling and increase the cohesion.

In the end reducing coupling in a design is a good thing, but we should not waste effort on areas of the design we cannot predict will change, in fact doing so would increase the complexity of the design and we should not waste effort in this way. Thus coupling is something that can be reduced in a design, but we should only do so where this makes for a better design.

References

[Ball96] Ball, Thomas. Eick, Stephen G. *Software Visualization in the Large.* IEEE Computer Society Press, 1996

[Eckstein04] Eckstein, Jutta. *Agile Software Development in the large – Diving Into the Deep.* Dorset House Publishing, 2004.

[Evans04] Evans, Eric. *Domain Driven Design.* Addison-Wesley, 2004

[Feathers05] Feathers, Michael C. *Working Effectively with Legacy Code.* Prentice Hall, 2005.

[GoF95] Gamma, Eric.Helm, Richard. Johnson, Ralph. Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley 1995.

[Henney07] Henney, Kevlin. *The Selfish Object.* http://www.software-architect.co.uk/slides/sa07-KevlinHenney-Selfish_Object.pdf

[Hunt00] Hunt, Andrew. Thomas, David. *The Pragmatic Programmer.* Addison-Wesley, 2000.

References

[Larman02] Larman, Craig. *Applying UML and Patterns (second edition).* Prentice Hall, 2002.

[Martin03] Martin, Robert C. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2003.

[Meyer97] Meyer, Bertrand. *Object Orientated Software Construction (second edition).* Prentice Hall, 1997.

[Myers78] Myers, Glenford J. *Composite/Structured Design.* Van Nostrand Reinhold Company, 1978.

[Page-Jones88] Page-Jones, Meilir, *The Practical Guide to Structured System Design (second edition).* Prentice Hall, 1998.

[Sharp97] Sharp, A. *Smalltalk By Example* . McGraw-Hill, 1997.