Modularity

Philipp Reinecke

ReineckeP@cardiff.ac.uk

What is it?

- Breaking down of complex system into manageable pieces

Separately named

Separately addressable

"Modules"

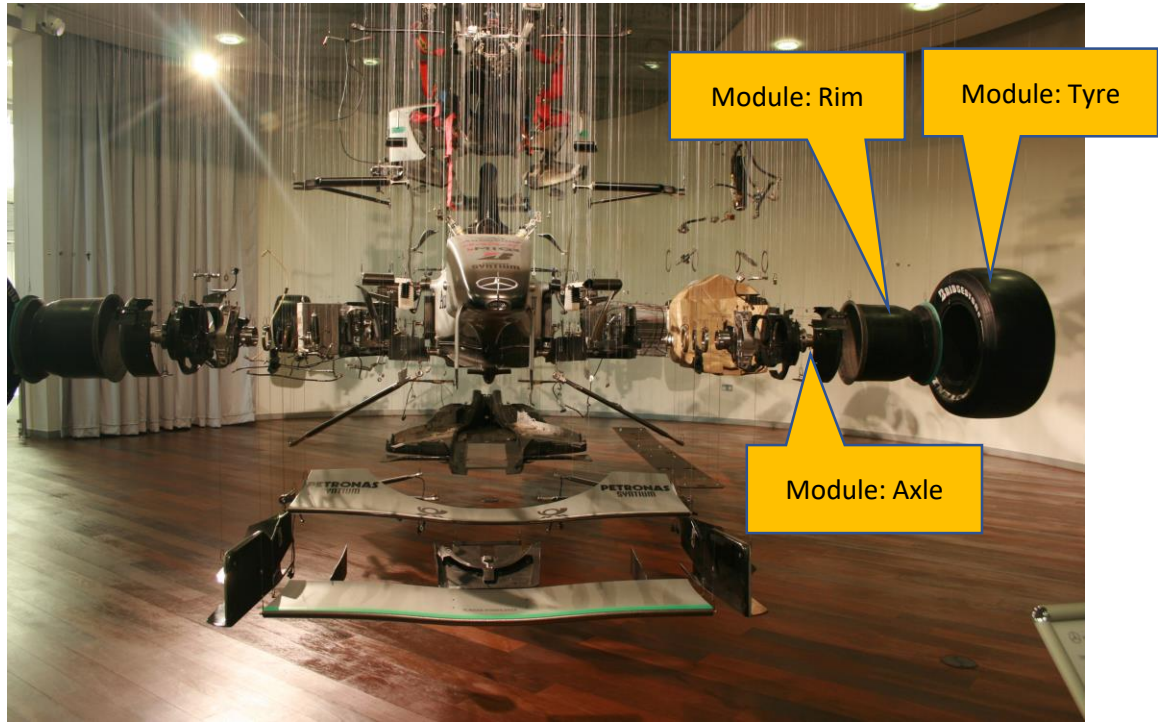
Clear interfaces

What is a Module?

- Logical unit of the system
- Interactions with other modules via interfaces
- Hides internal details
- May depend on level of abstraction – a system can be a module in a larger system

"[A] module is an identifiable unit in the design."

Hans van Vliet, Software Engineering. Principles and Practice (1993), p.182



Why Modularity?

- Helps manage system complexity
- Helps manage team-based development

"Modularity is the single attribute of software that allows a program to be intellectually manageable"
G. Myers (1978) Composite Structured Design, Van Nostrand Reinhold

Monolithic Design
- Difficult to understand
  - Need to keep whole program in mind
  - What is the code doing?
  - What is affected by what?
- Difficult to maintain
  - Mistakes are easy to make and hard to find
  - Effects of changes are unclear → Hard to change program
- Difficult to mange in team
  - → Who edits what?

Modular Design
- Easier to understand
  - Divide et Impera – Divide and Conquer
  - Only need to understand behaviour of modules and their interactions
  - Can focus on specific parts
- Easier to maintain
  - Can focus on one module, other modules are just black boxes
  - Modules can be reused
- Helps team-based development
  - Parallel development of modules
  - Clear responsibilities

Properties of Modular Designs

- Many modular designs possible

  → Which one is best?

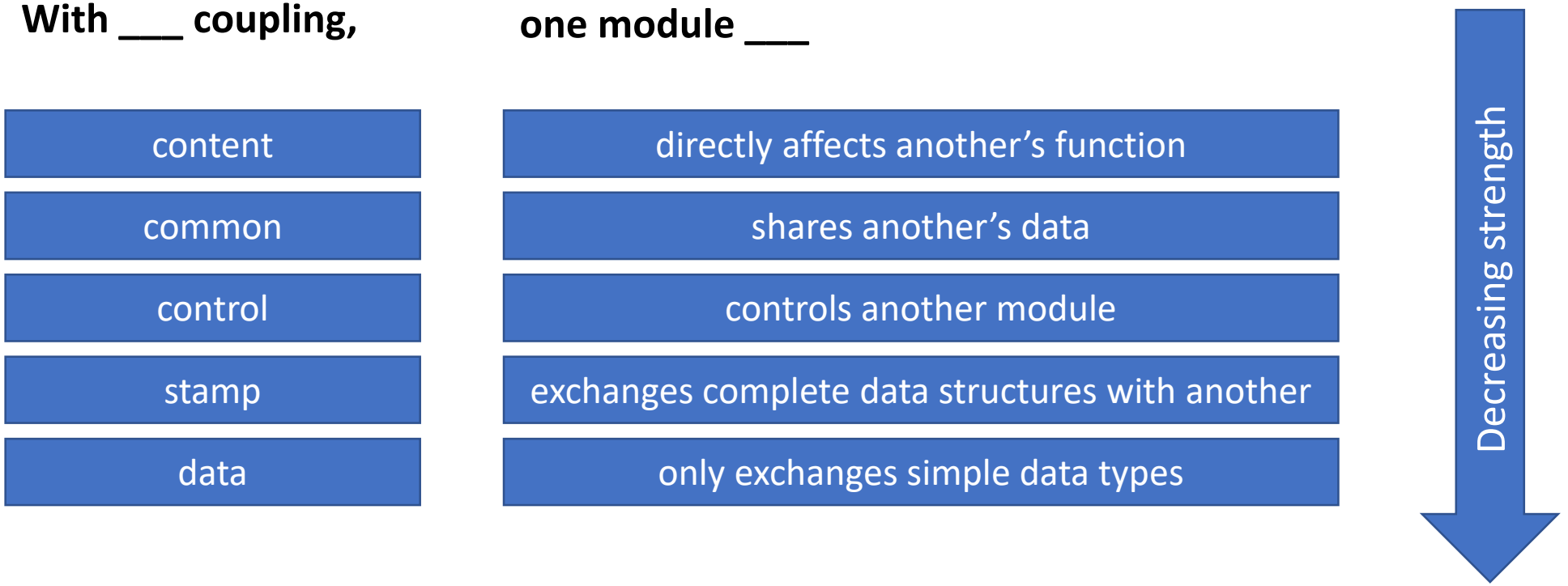
- Comparison criteria:
  - Cohesion
  - Coupling
- Cohesion
  - Mutual affinity of module's components
  - "Glue that keeps the module together"
- Coupling
  - Dependence on other modules
  - Strength of inter-module connections

General Aims: High Cohesion, Low Coupling

Types of Cohesion

With ___ cohesion,	components in module ___
coincidental	have no relation at all
logical	perform similar functions
temporal	are used at the same time
procedural	are run in given order
communicational	operate on same external data
sequential	operate on each other's output
functional	contribute to one single function of the module
data	encapsulate an abstract data type

Types of Coupling



Aim of Modular Design: Functional Independence

- Aims:
  - Maximise cohesion
  - Minimise coupling
- Maximise cohesion
  - Ensure module has single, well-defined purpose
  - Module should deal with a single function or entity
  - Module should not deal with unrelated functions or entities
- Minimise coupling
  - Minimise dependencies between modules
- The two aims may conflict

Advantages of strong cohesion and weak coupling

- Simpler communication between programmers – local decisions are possible
- Propagation of changes to other modules less likely – reduced maintenance costs
- Reusability is increased – less assumptions on specific environment increase likelihood of fitting another one
- Increased comprehensibility – understanding independent of environment
- Less error-prone
- Correctness proofs are easier

Techniques for Modularity

- Understanding cohesion
- Functions
- Object-Oriented Programming

Understanding Cohesion

- Simple exercise [Stevens74]:

  Write down and analyse a sentence describing the module's purpose

- Compound sentence, contains a comma or "and", or more than one verb?
  Yes → Probably more than one function, probably sequential or communicational cohesion
- Contains words relating to time (e.g. "first", "after")?
  Yes → Sequential or temporal cohesion likely
- Contains words like "initialise"?
  Yes → Temporal cohesion likely

Functions

- A *function* groups a set of statements so they can be run more than once in a program
  - If a set of statements is likely to be required more than once in a program then put them in a function
    - Aids Maintainability, Testability, Reusability and Adaptability
- Complex code will benefit from being broken into smaller functions for each subtask
  - Ensure each function carries out a single well-defined purpose (temporal/procedural cohesion to functional cohesion)
  - Aids readability
- Different developers can be assigned different functions to speed up development

Functions

- Aim: Decompose the task into purposeful functions (cohesion) and determine how the functions should communicate (coupling).

- Guidelines:
  - Coupling – use arguments for inputs and return for outputs. Try to make the function independent of things outside it.
  - Coupling – use global variables only when truly necessary
  - Coupling – avoid changing variables in another module directly
  - Avoid unnecessary side-effects

Functions

- The more self-contained a function is, the easier it will be to
  - Understand
  - Reuse
  - Modify

  → good for maintainability and testability!

- Guidelines (Continued):
  - Cohesion – each function should have a single unified purpose.
  - Size – each function should be relatively small and ideally be visible on one screen.

Object-Oriented Design

- System is decomposed structurally into:
  - Objects, their attributes and services(methods) and the relationships between objects
  - System functionality is provided by sets of objects that interact by passing messages
- Not dependent on implementation language
  - Java, C++, Python etc. support OO

Classes

- Classes provide a template for generating objects
  - Objects are specific instances of a class that store values
  - Classes define a range of attributes storing information about an object
  - Functions inside a class are called *methods.*
  - Methods are used to change the values stored in the attributes
    - This changes the state of the object
- Each class should have a single, *cohesive purpose*
- *Coupling is minimised* by passing data using arguments when methods are called.

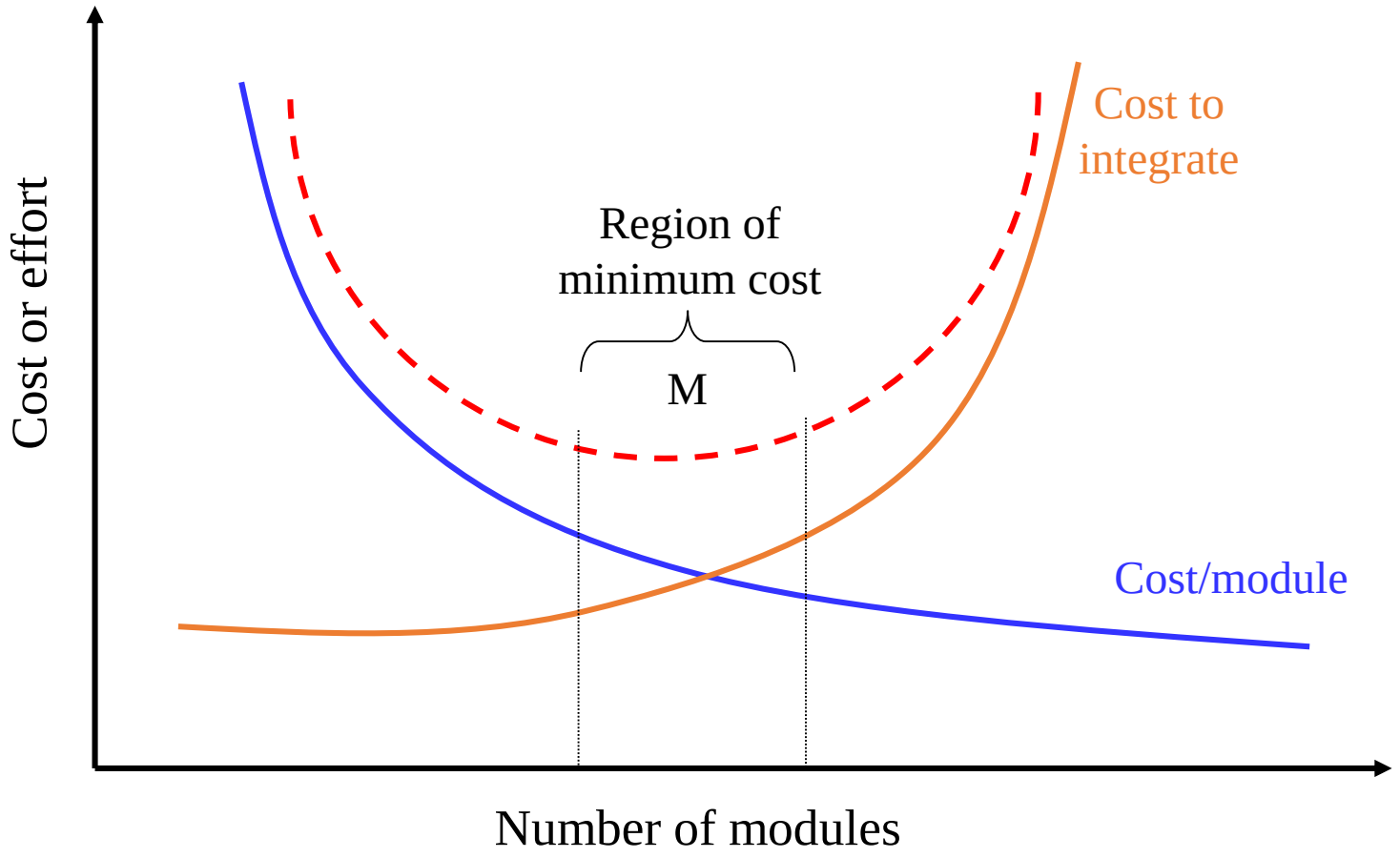
Classes

- Methods also define class behavior
  - Methods should be part of the class their behavior belongs to
- Methods can be called when needed from this or other classes
- Encapsulation promotes adaptability
  - Code in the method can change without breaking code in the calling classes if the interface to the method is unchanged

Effective Modular Design

- Effective modular design is achieved by:
  - Refinement: Create successive representations of a system where each new representation is more detailed than the last.
  - Abstraction: Concentrate on essential issues and ignore details that are irrelevant at a particular level of representation
  - Information hiding:
    - Hide internal details of processing, data, & control activities
    - Communicate only through well-defined interfaces

Cost of Modularity (Pressman pg. 339)



Conclusion

- Break problem up into manageable parts
- Break solution up into manageable parts
- Maximise cohesion
- Minimise coupling
- Hide information
- Do not overdo it