COMP201 – Software Engineering I
Lecture 15 – Sofware Design

Lecturer: T. Carroll

Email: Thomas.Carroll2@Liverpool.ac.uk

Office: Ashton G.14

See Vital for all notes

Software Design Is…

Deriving a solution which satisfies software requirements

Recap…

Recap – Lecture 14

- Characteristics of good software
  - Simple
  - Understandable
  - Portable
  - Flexible
  - Re-usable
- Modular Design
- Characteristics of good modular design
  - Modular Decomposability
  - Modular Composability
  - Modular Continuity
  - Modular Protection
  - Modular Understandability

Today

Software Design

- In this lecture we shall again be looking at how to derive a software solution which satisfies the software requirements.
- We shall be examining ways in which architecture designs and individual components can be said to be “good” in practice.
- A good system could have the following properties:
  - Useful and usable;
  - Reliable;
  - Flexible;
  - Affordable;
  - Available.
- **Question**: How do modules allow us to meet some or all of the above goals of a good system?

Module Interfaces

- An interface to a module defines features of the module upon which other parts of the system may rely.
- An interface is an abstraction of the module.
- Encapsulation of the module means that users of the module cannot know more about the module than is given by the interface.
- Any **assumptions** should be documented in the interface.
  - Ie: changes to the *internals* of a module not causing a change to the interface should not necessitate a change to other parts of the system.

Example Interface (Vector.hpp)

```
// A C++ Vector class to perform mathematical calculations
// All Vector arguments should be normalized
class Vector {
public:
      Vector(float x, float y, float z);
      ~Vector();
      float dotProduct(Vector inVector);
      Vector crossProduct(Vector inVector);
      void setVector(float x, float y, float z);
private:
      void normalize();
      float x_value;
      float y_value;
      float z_value;
};
```

Public interface

Private interface

Example Interface (`Vector.hpp`)

- Does `Vector.hpp` tell us how to compute the cross product of two vectors?
  - **NO!**
  - Because of abstraction, we don't need to know…
  - That information is hidden in the module!
- We may choose to replace the code for computing the cross product to be faster or more numerically stable for example, but users of the module (or component) should not need to change their code since the interface has not changed.

Five Principles for Good Design

- From the discussion in the previous lecture, we can derive five principles that should be adhered to:
  - Linguistic modular units;
  - Few interfaces;
  - Small interfaces;
  - Explicit interfaces;
  - Information hiding.

Linguistic Modular Units

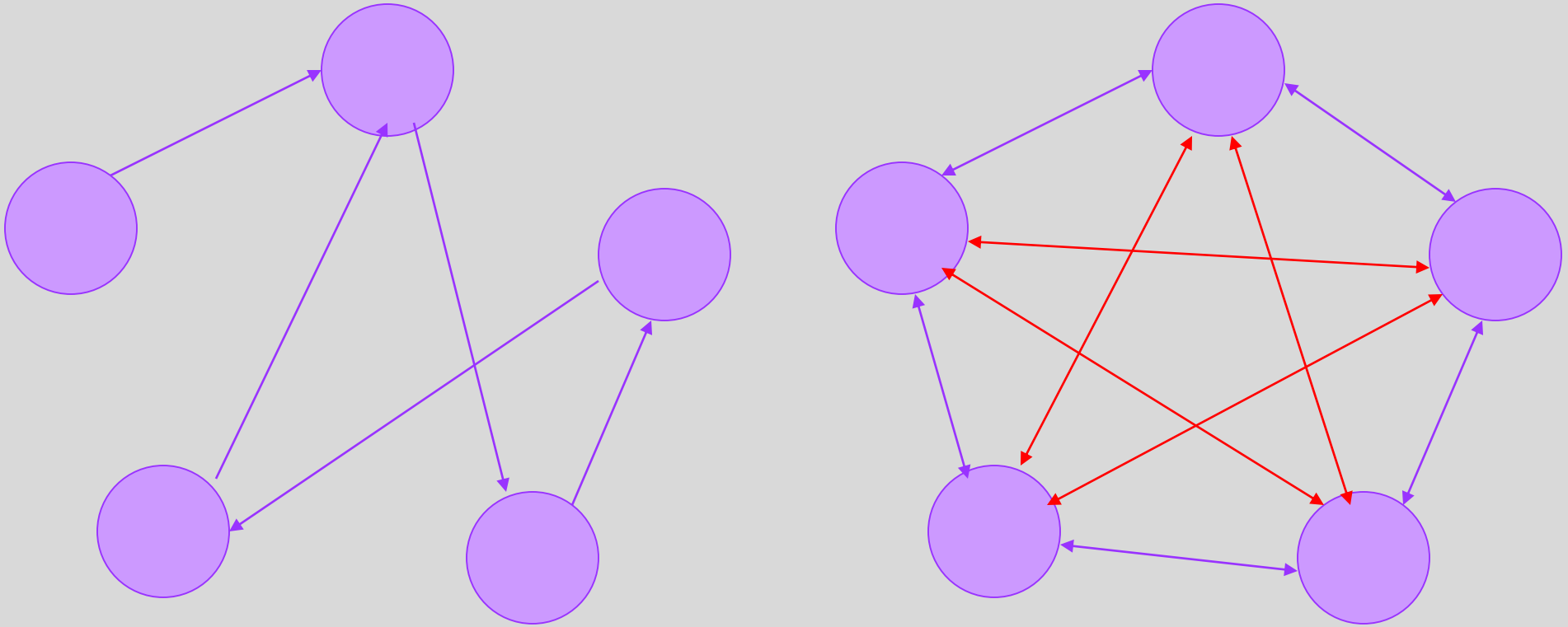
- Modules must correspond to linguistic units in the language used

- **EXAMPLE:** Java methods and classes
- **COUNTER EXAMPLE**: Subroutines in BASIC are called by giving a ***line number*** where execution is to proceed from; there is no way of telling, just by looking at a section of code, that it is a subroutine.

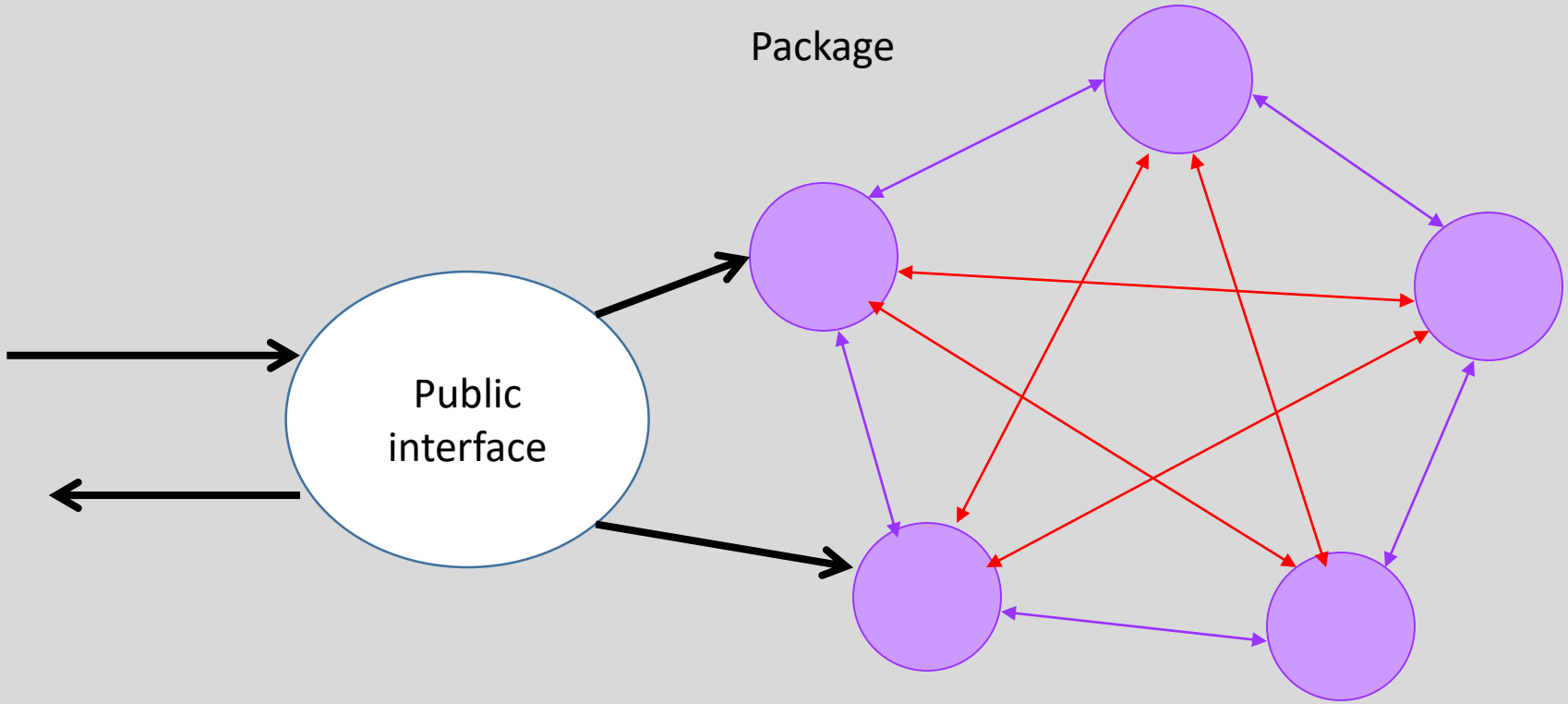
Few Interfaces

- This principle states that the overall number of communication channels between modules should be as small as possible:
  - Each module should communicate with as few others as possible.
- So, in the system with *n* modules, there may be a minimum of *n-1* and a maximum of $\frac{n(n-1)}{2}$ links;
- Your system should stay closer to the minimum.

Few Interfaces



Facade structure



Small Interfaces (Loose Coupling)

- **Loose coupling** implies that :
  - If any two modules communicate, they exchange as little information as possible.
- A module's interface defines the features on which a client may rely
- The rest of the system can only use the module as permitted by the interface.

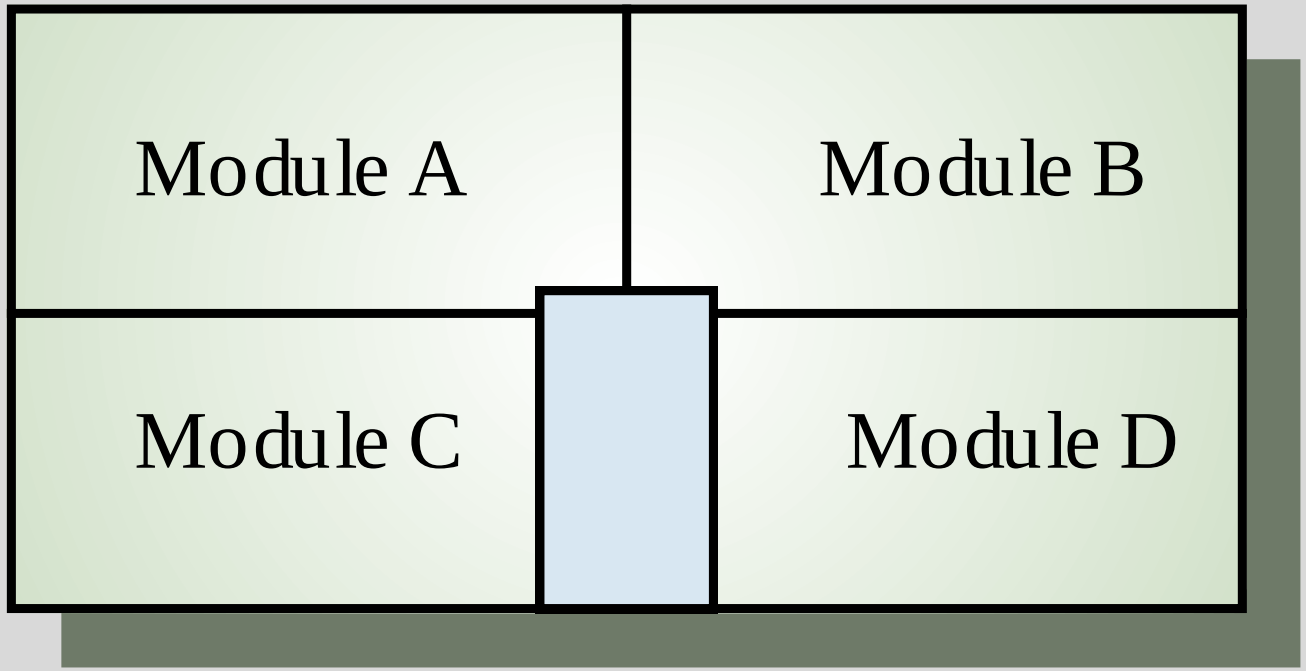
Loose Coupling – Invoking method of class only

Tighter coupling – Changing data within another class directly

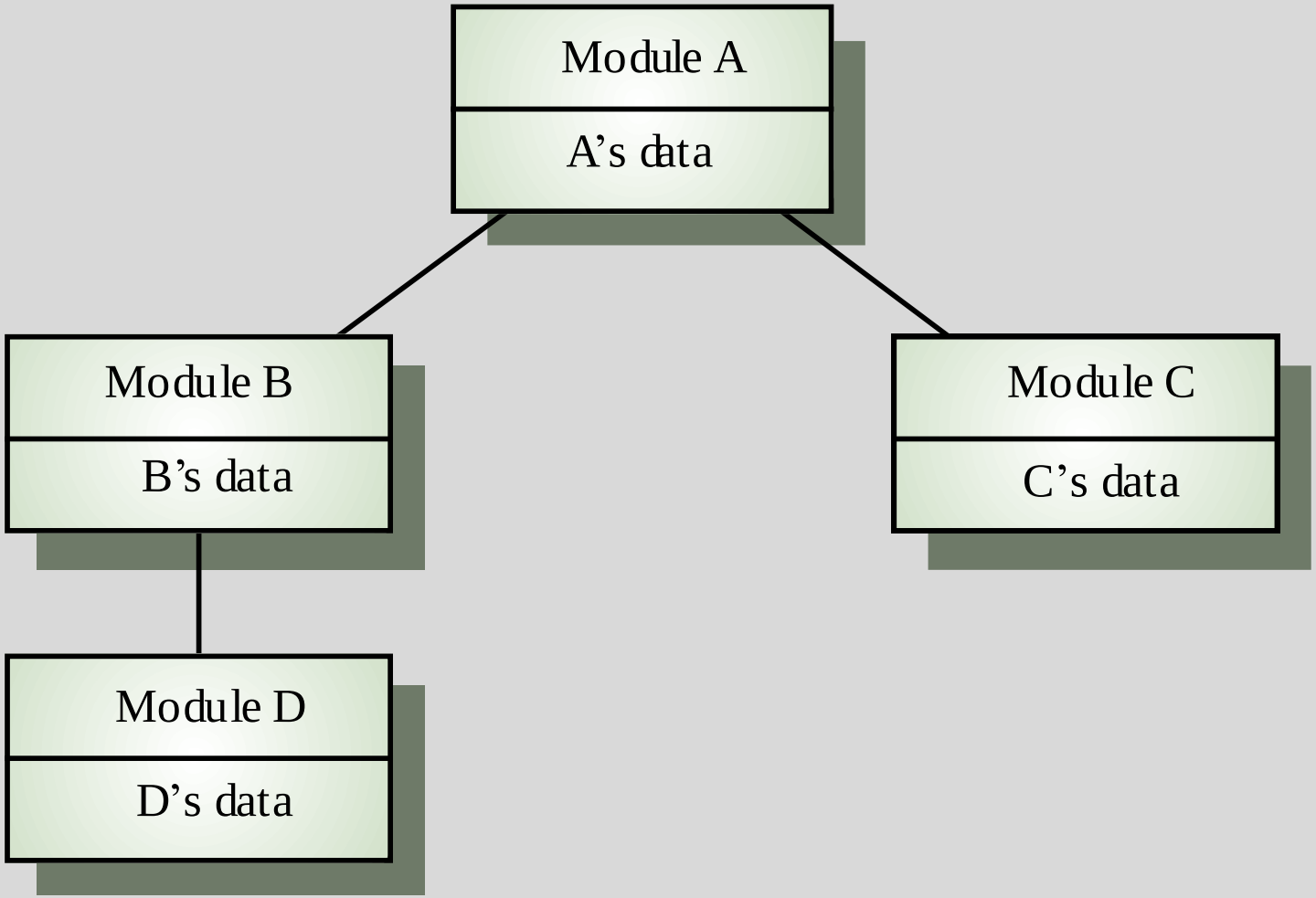
Coupling

- **Coupling** is a measure of the strength of the inter-connections between system components.
- **Loose coupling** means component changes are unlikely to affect other components.
  - Shared variables or control information exchange lead to **tight coupling** (usually bad).
  - Loose coupling can be achieved by:
    - state decentralization (as in objects)
    - component communication via parameters or message passing.

Tight Coupling



Loose Coupling – OO Example

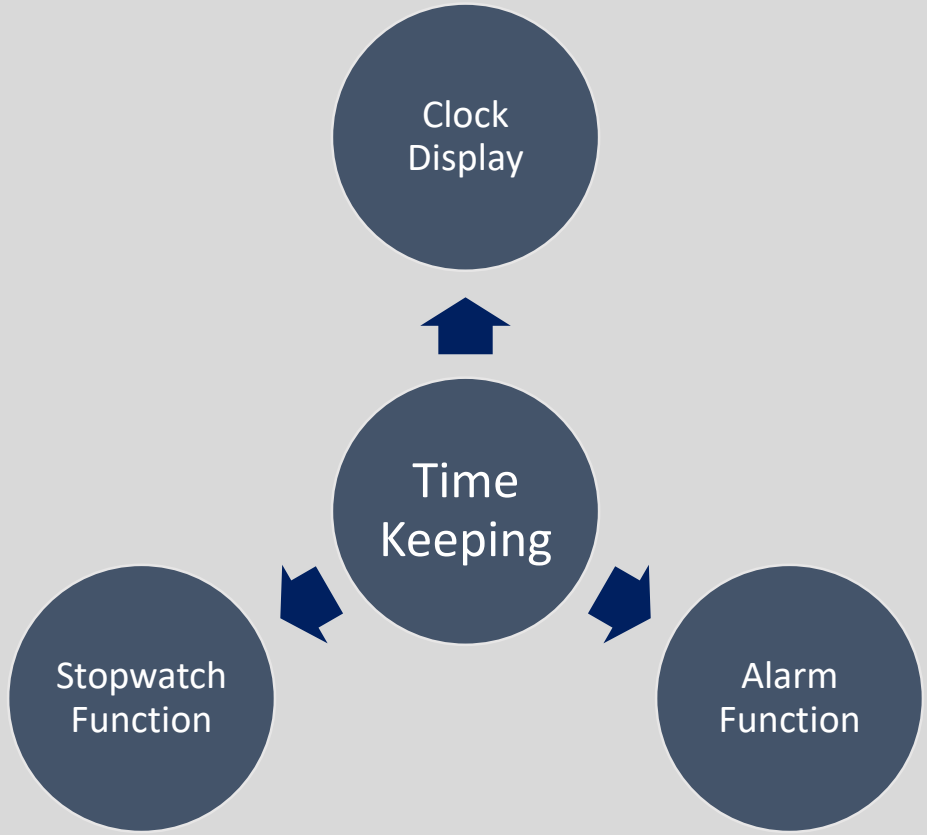


Coupling and Inheritance

- Object-oriented systems are loosely coupled because there is no shared state and objects communicate using message passing.
- **However, an object sub-class is coupled to its super-classes.**
- Changes made to the attributes or operations in a super-class propagate to all sub-classes.

Repository Models

- Sub systems making up a system must exchange and share data so they can work together effectively.
- Two main approaches:
  - **Repository model** – All shared data held in a central database, accessed by all sub-systems
  - **Local Database** on each subsystem. Data exchanged between sub-systems via message passing.



Repository Model

✓ Advantages:

- ✓ Databases are an efficient way to share large amounts of data
  - ✓ Subsystems use a single data representation
- ✓ Sub-systems producing data need not be concerned with how that data is used by other sub-systems.
- ✓ Many standard operations such as backup, security, access control, recovery and data integrity are centralised and can be controlled by a single repository manager.
- ✓ The data model is visible through the repository schema.

× Disadvantages:

- × Sub-systems must agree on the data model
  - × compromises must be made
- × Evolution may be difficult since a large amount of data is generated
  - × translation may be difficult and expensive.
- × Different systems have different security/recovery/backup requirements
  - × may be difficult to enforce in a single database.
- × It may be difficult to distribute the repository over a number of different machines.

Reusability

- A major obstacle to the production of cheap quality software is the intractability of the reusability issue.
- Why isn’t writing software more like producing hardware?
- Why do we start from scratch every time, coding similar problems time after time after time?
- Obstacles:
  - Economic
  - Organizational
  - Psychological

Stepwise Refinement

- The simplest realistic design method, widely used in practice.
- Not appropriate for large-scale, distributed systems: mainly applicable to the design of methods.
- Basic idea is:
  - Start with a high-level spec of what a method is to achieve;
  - Break this down into a small number of problems (usually no more than 10)
  - For each of these problems do the same;
  - Repeat until the sub-problems may be solved immediately.

Explicit Interfaces

- If two modules ***must*** communicate, they should do it so that we can see it:
  - If modules A and B communicate, this must be obvious from the documentation of A or B or both.
- **Why?** If we change a module, we need to see what other modules may be affected by these changes.
- A **traceability matrix** can be used for this purpose.

Information Hiding (Encapsulation)

- This principle states:
  - All information about a module, (and particularly *how* the module does what it does) should be private to the module unless it is specifically declared otherwise.
- Thus each module should have an *interface*, which is how the world sees it
- Anything beyond that interface should be hidden.
- There is a limit to how much humans can understand at any one time.
- The default Java rule:
  - Make everything private

Cohesion

- A measure of how well a component “fits together”
- A component should implement a single logical entity or function.
- Cohesion is a desirable design component attribute
  - when a change has to be made, it is localized in a single cohesive component.

Cohesion Levels

- Coincidental cohesion (weak)
  - Parts of a component are simply bundled together.
- Logical association (weak)
  - Components which perform similar functions are grouped.
- Temporal cohesion (weak)
    - Components which are activated at the same time are grouped.

Cohesion Levels

- Communicational cohesion (medium)
  - All the elements of a component operate on the same input or produce the same output.
- Sequential cohesion (medium)
  - The output for one part of a component is the input to another part.
- Functional cohesion (strong)
  - Each part of a component is necessary for the execution of a single function.
- Object cohesion (strong)
  - Each operation provides functionality which allows object attributes to be modified or inspected.

Cohesion as a Design Attribute

- The concept of cohesion is not well-defined and is often difficult to classify.
- Inheriting attributes from super-classes weakens cohesion.
  - To understand a component, the super-classes as well as the component class must be examined.
  - Object class browsers assist with this process.

Cohesion versus Encapsulation

- **Cohesion** is a measure of abstraction that means developers do not need concern themselves with the internal working of a module.
- **Encapsulation** means that developers are unable to use hidden information within a module, ensuring that subtle errors cannot be introduced when using connected modules.
- This is a subtle but important difference between the two concepts.

High Cohesion, Low Coupling Modules

- If a module has high cohesion, low coupling and a well defined interface, then it may be feasible to reuse that module in other systems
  - I.e., it may be a *pluggable module*.
  - This will also depend upon the architecture in which the module was developed, remember the example of the Ariane 5 space launcher from the last lecture?
- Architectural decisions are usually important to the entire design of the module and thus should be taken early. For example, perhaps the module would be made less efficient but more general since we may wish to reuse the module later..

Again..what makes a Good System?

- We said that a good system has the following properties:
  - Useful and usable;
  - Reliable; (low coupling)
  - Flexible; (low coupling, high cohesion)
  - Affordable; (software reuse)
  - Available. (decreased development time through reuse)
- **Question**: Can we now answer why properly designed modules allow us to meet some or all of the above goals of a good system?

Lecture Key Points

- We have seen the definition of **coupling**, **cohesion** and **interfaces** to modules and how they can inform us about the reusability of a component
- Decisions about modules and their interfaces are important design considerations that should be made early in the design phase
- Object-oriented development often encourages good modular design