The Design Phase

- The lifecycle phase that operates the transition:
  - ***from what*** is the software to be produced
  - ***to how*** that software has to be produced
- Takes as input the *software requirements specification document* and yields as output the *design document*, which drives the subsequent coding phase
- It is divided into two sub-phases:
  - **architectural** (or **preliminary**, or **high-level**) **design**, which operates the modular decomposition leading to the software architecture
  - **detailed design**, which provides the detailed design, in terms of data structures and algorithms, of each single module of the software architecture

Design principles

- stepwise refinement (also used in the *requirements specification phase*)
- abstraction (also used in the *requirements specification phase*)
- modular decomposition
- modularity
- information hiding
- reusability

Stepwise refinement

- Design strategy that makes use of a top-down approach, as proposed by *Wirth* (1971) in the context of structured programming
- The refinement process starts from the specification of a function (or data) that does not include the internal mechanisms of the function (or the internal details of the data structure)
- At each step of refinement the level of detail of the specification is increased by including additional elements, until the entire specification is reduced to base elements

Stepwise refinement (2)

- The importance of stepwise refinement directly comes from the **Miller Law (1956)**: *"at any one time a human being can concentrate on at most 7 ± 2 chunks (quanta of information)*"

- Refinement is a concept complementary to the abstraction concept

Abstraction

- Abstraction consists of focusing on the essential aspects of a given entity, by suppressing the more complex details
- The abstraction concept has been introduced by *Djikstra* (1968) in order to describe the layered architecture of operating systems
- In a conventional (i.e., waterfall) software process, each phase gives a more refined solution
- This means focusing on *what* an entity is and *what* it does before dealing with *how* it has to be realized

Abstraction (2)

- Abstraction can apply to control or to data:
  - **control (or procedural) abstraction** (e.g., *C functions*)
  - **data abstraction** (e.g., *C data structures*)
- A data structure along with the actions to be executed over it is referred to as providing ***data encapsulation*** (e.g., *a stack*, data structure with the actions that implement the LIFO access policy)
- Using *data encapsulation* at design time allows one to obtain significant advantages both during the coding phase and during maintenance

Abstract Data Types

- An *abstract data type*(ADT) identifies a data type whose instances provide data encapsulation
- An ADT combines control and data abstraction
- Using ADTs allows one to improve software quality, in terms of better levels of *reusability* and *maintainability* attributes
- A *C++ class* is an example of ADT, which also supports inheritance and polymorphism

Example *abstract data type* (C++ class)

```
class JobQueue
{
  // attributes
  public:
    int queueLength;                    // length of job queue
    int queue[25];                      // queue can contain up to 25 jobs

  // methods
  public:
    void initializeJobQueue ()
    /*
     * empty job queue has length 0
     */
    {
      queueLength = 0;
    }

    void addJobToQueue (int jobNumber)
    /*
     * add job to end of job queue
     */
    {
      queue[queueLength] = jobNumber;
      queueLength = queueLength + 1;
    }

    void removeJobFromQueue (int& jobNumber)
    /*
     * set jobNumber equal to the number of the job stored at the head of the queue,
     * remove the job at the head of the job queue, and move up the remaining jobs
     */
    {
      jobNumber = queue[0];
      queueLength = queueLength –  1;
      for (int k = 0; k •  queueLength; k++)
        queue[k] = queue[k + 1];
    }
} // class JobQueue
```

Modularity (1)

- Software products made of a single monolithic code block are difficult to:
  - maintain
  - fix
  - understand
  - reuse
- A valid solution is to decompose the product into a set of smaller segments, denoted as **modules**

Modularity (2)

- Definition (IEEE Std 610.12 - Standard Glossary of Software Engineering Technology):

  the extent to which software is composed of discrete components such that a change to one component has minimal impact on the other components

Modular decomposition

- A module is a software element that:
  - includes program statements, processing logic and data structures
  - can be compiled independently and stored in a software library
  - can be included in a program
  - can be used by invoking module segments identified by a name and a list of parameters
  - can use other modules
- Modular decomposition of a software product yields as output the so-called ***modular architecture*** (or structure chart)

Modular decomposition (2)

- The modular architecture describes the structure of modules, along with how such modules interact with each other and the data structures that are used
- Modular decomposition is based on the "*divide et impera*" principle
- Let's denote as **p1** and **p2** two *problems*, **C** their *complexity* and **E** the *effort* needed to solve them. Then:

$$\mathbf{C(p1) > C(p2) \Rightarrow E(p1) > E(p2)}$$

$$\mathbf{C(p1+p2) > C(p1) + C(p2)}$$

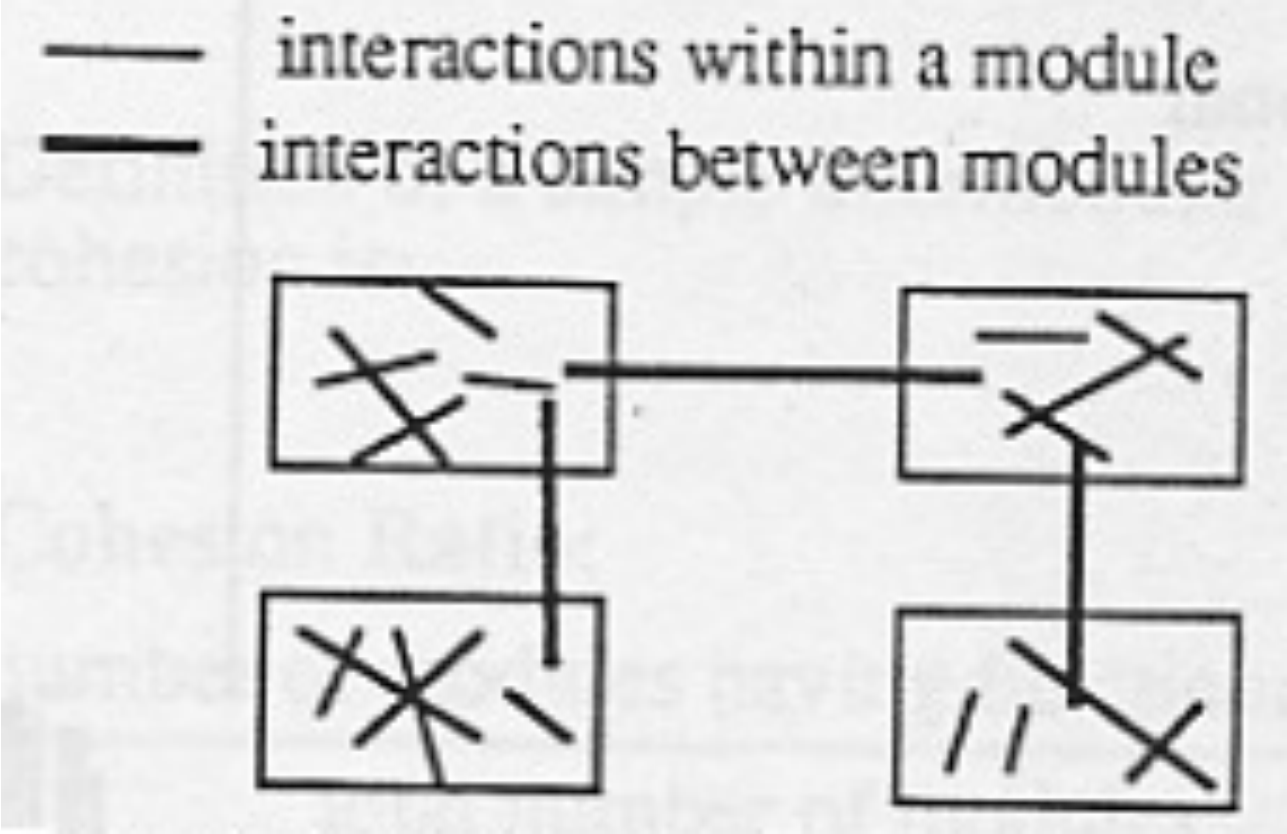
$$\Downarrow$$

$$\mathbf{E(p1+p2) > E(p1) + E(p2)}$$

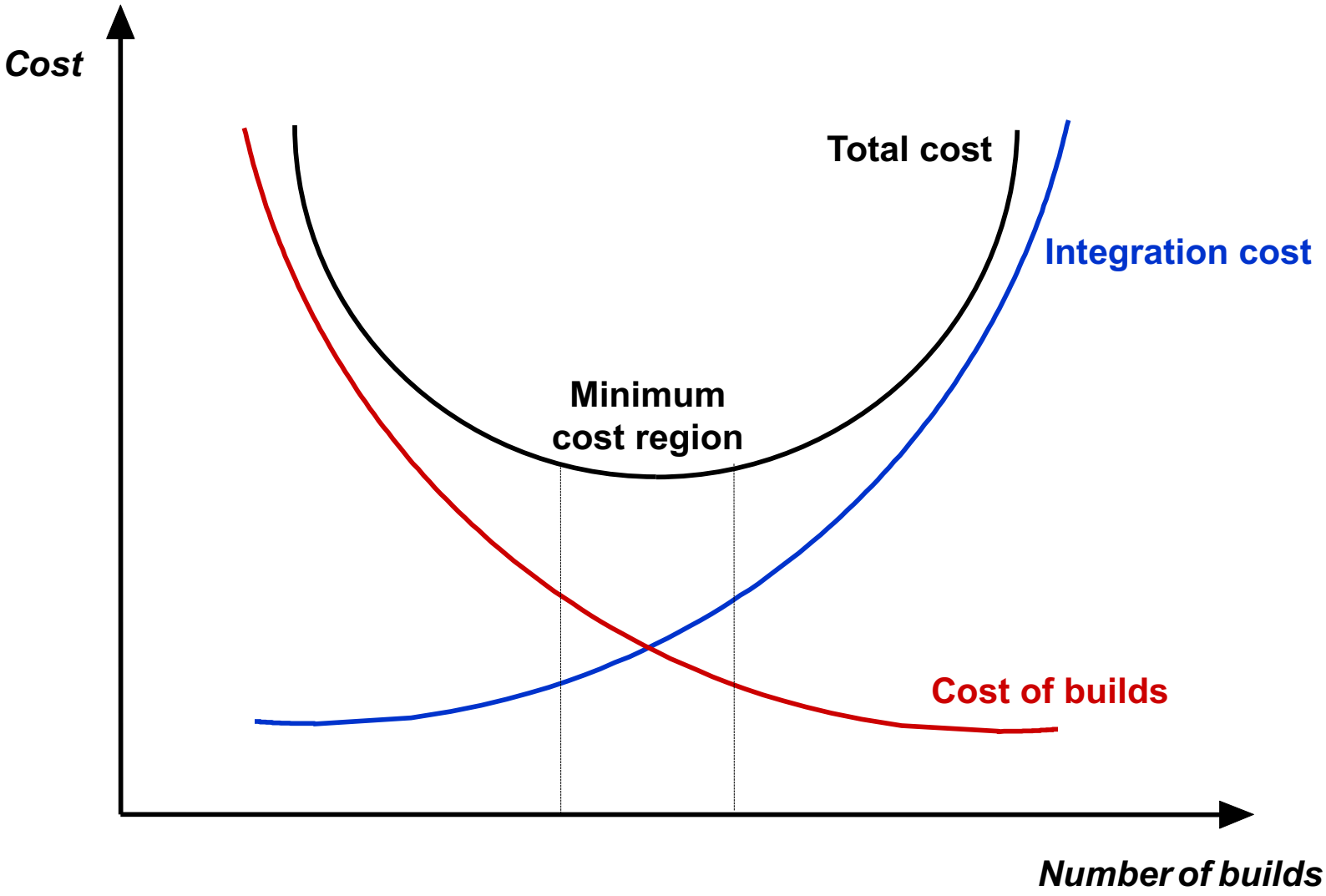
Modular decomposition (3)

- A modular decomposition is considered *good* if obtains:
  - *Maximum* **cohesion** *internal* to modules
  - *Minimum* **coupling** *between* modules
- Obtaining maximum cohesion and minimum coupling allows one to get increased levels of the following software product quality attributes:
  - Understandability
  - Maintainability
  - Extensibility
  - Reusability

Cohesion/Coupling with respect to Modularity



Modularity vs. Cost



Cohesion

- Various *actions* are usually required to execute a given *function*
- Such actions can be carried out by a single module, as well as be spread into different modules
- **Module cohesion**
  - a measure of the module capability to accomplish *internally* what required to execute a function (i.e., without interacting with other modules)
- Cohesion thus measures the strength of *internal relationship* between elements within a given module

Cohesion levels

(*1* is the worst, *7* the best)

1. **Coincidental** (no relationship between module elements)
2. **Logical** (logically related elements, characterized by the fact that only one is executed by the calling module)
3. **Temporal** (temporally related elements, i.e., the elements are processed at a particular time in program execution)
4. **Procedural** (elements related by the fact that are executed according to a predefined sequence).
5. **Communicational** (elements related by the fact that are executed according to a predefined sequence and on a single data structure)
6. **Informational** (each element has a separate portion of code with associated input/output ports; all elements operate on the same data structure)
7. **Functional** (all the elements are related by the fact that execute a single function)

Cohesion levels (2)

optimal for the structured paradigm

optimal for the OO paradigm

7.	Functional cohesion	(Good)
	Informational cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

Coincidental Cohesion: example

- Module functions:
  - print next line
  - invert characters of the second string parameter
  - add 7 to the fifth parameter
  - perform *int-double* conversion to the fourth parameter

Logical Cohesion: example

module performing all input and output
1. Code for all input and output
2. Code for input only
3. Code for output only
4. Code for disk and tape I/O
5. Code for disk I/O
6. Code for tape I/O
7. Code for disk input
8. Code for disk output
9. Code for tape input
10. Code for tape output
⋮ ⋮ ⋮
37. Code for keyboard input

Temporal Cohesion: example

- Module functions:
  - Open *old_master_file*
  - Open *new_master_file*
  - Open *transaction_file*
  - Open *print_file*
  - Initialize *sales_region_table*
  - Read first *transaction_file* records
  - Read first *old_master_file* record

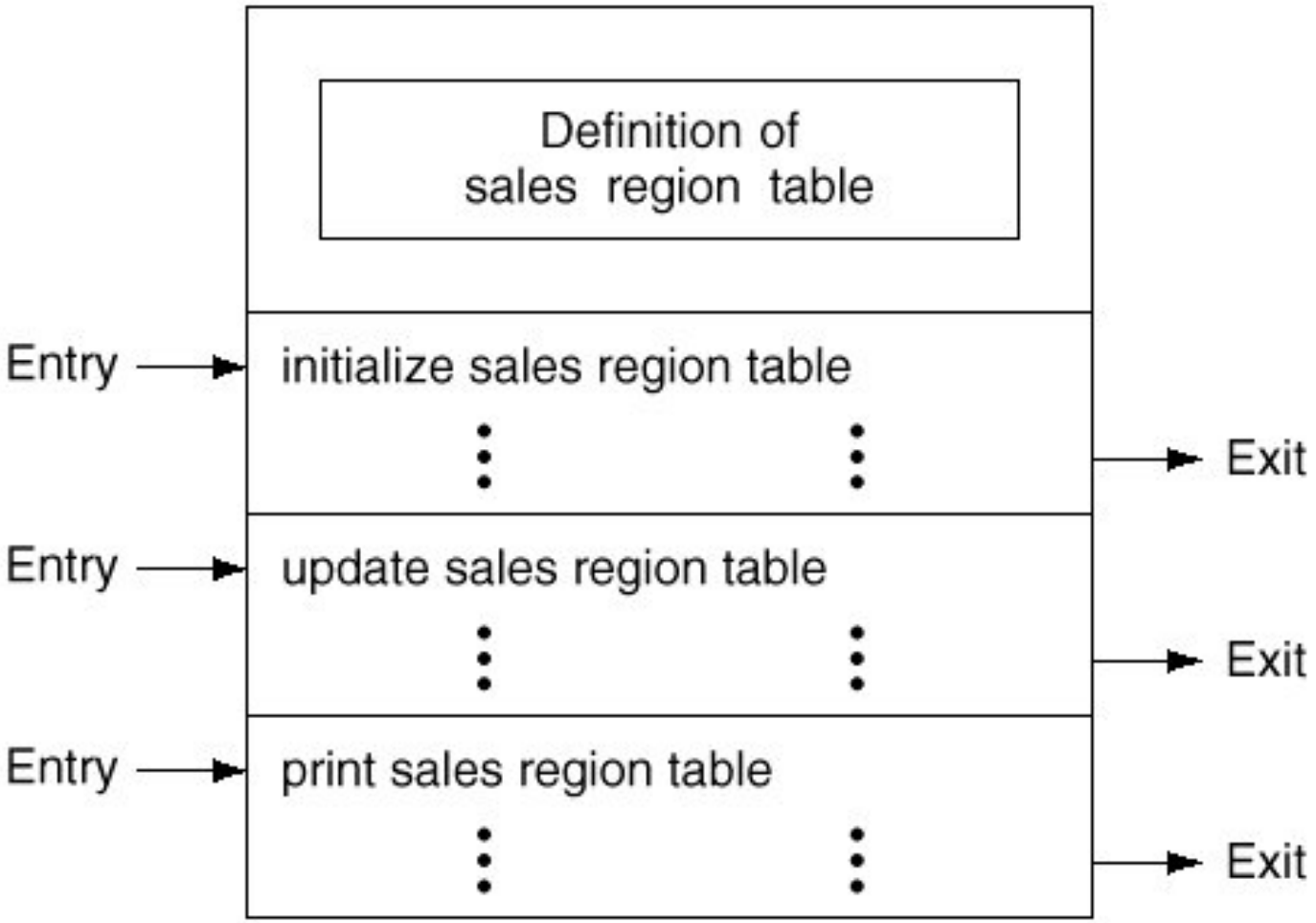
Procedural Cohesion: example

- Module functions:
  - Read *part_number* from database
  - Use *part_number* to update *repair_record* on *maintenance_file*

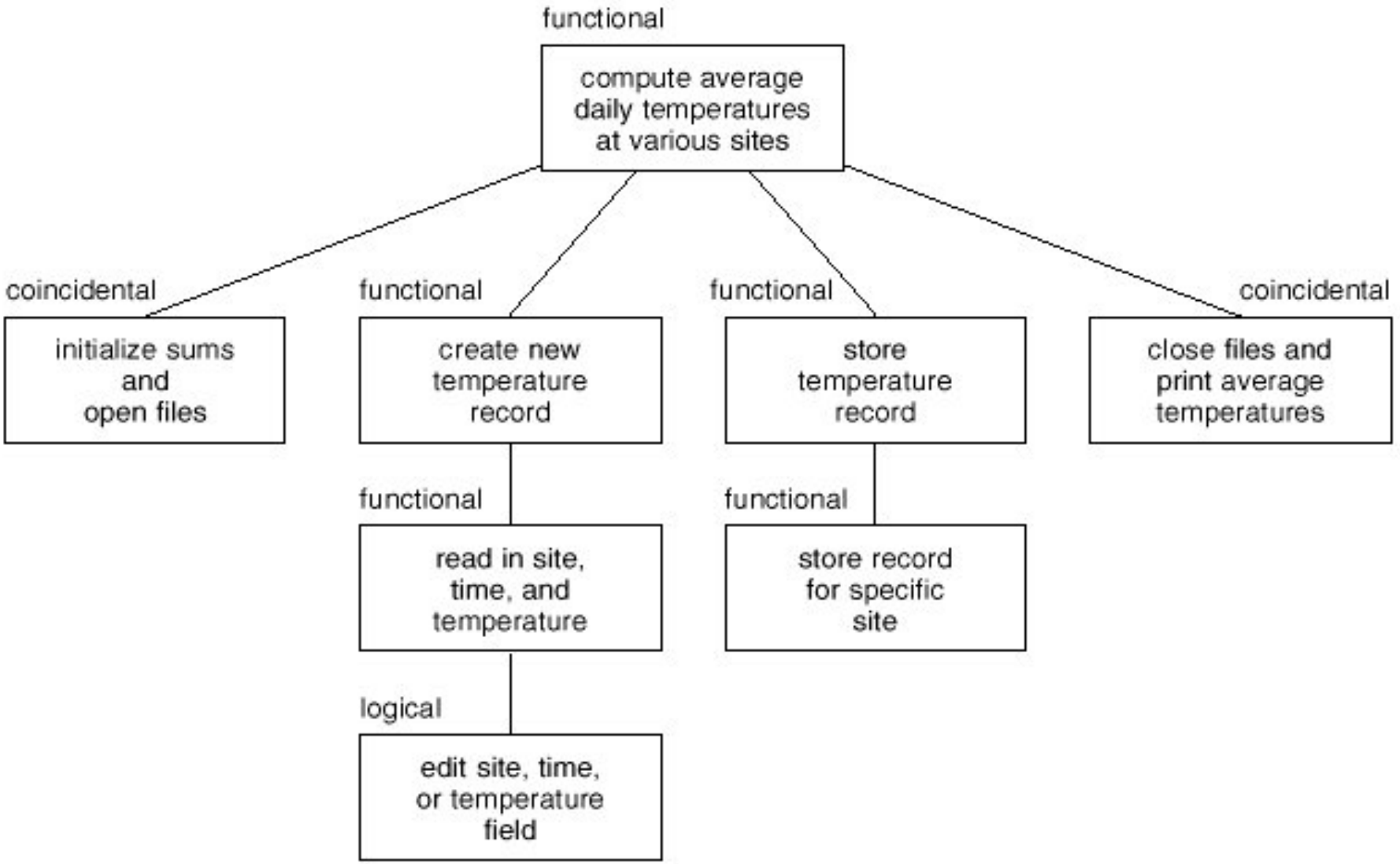
Communicational Cohesion: example

- Ex. 1: module functions
  - Update *record_a* in database
  - Write *record_a* to the *trajectory_file*
- Ex. 2: module functions
  - Calculate *new_trajectory*
  - Send *new_trajectory* to the printer

Informational Cohesion: example



Example Structure Chart & Modules Cohesion



Coupling

- A measure of the *degree of interaction between modules*
- *Coupling levels* (*1* is the worst, *5* the best):
  1. **Content** (a module explicitly refers to the content of another module)
  2. **Common** (two modules that have complete access to the same data structure)
  3. **Control** (a module that explicitly controls the execution of another module)
  4. **Stamp** (a module that passes a data structure to another module, which uses some elements of the data structure only)
  5. **Data** (a module that passes an argument of simple type to another module, or a data structure for which all elements are used).

Factors affecting Coupling

Strength of coupling depends on:

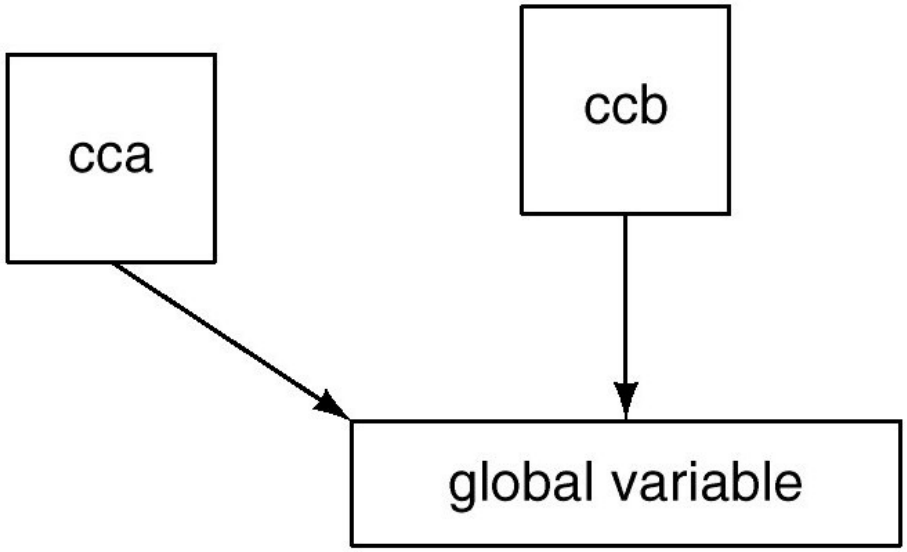
- the number of references of one module by another
- the amount of data passed/shared between modules
- the complexity of the interface between modules
- the amount of control exercised by one module over another

Content Coupling: example

$p \rightarrow q$

- Ex. 1: module *p* modifies a statement of module *q*
- Ex. 2: *p* refers to local data of module *q* in terms of some numerical displacement within *q*
- Ex. 3: *p* branches to a local label of *q*

Common Coupling: example

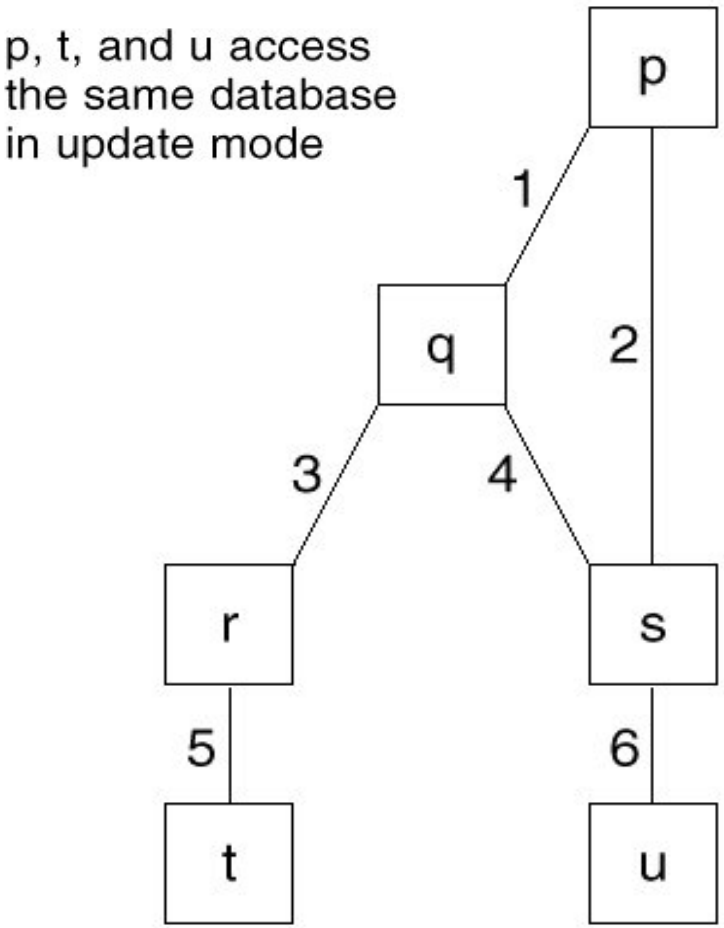



```
while (global variable == 0)
{
   if (argument xyz > 25)
      module 3 ();
   else
      module 4 ();
}
```

Control Coupling: example

- Module *p* calls module *q*
  - and asks *q* to perform an action,
  - *q* passes back a flag (e.g. “task not completed”)
  - and also asks *p* to perform an action (e.g. “print an error message”)

Example Structure Chart & Modules Coupling



parameters

number	In	Out
1	aircraft type	status flag
2	—	list of aircraft parts
3	function code	—
4	—	list of aircraft parts
5	part number	part manufacturer
6	part number	part name

coupling

	q	r	s	t	u
p	Data	—	Data or Stamp	Common	Common
q		Control	Data or Stamp	—	—
r			—	Data	—
s					Data

Information hiding

- Control and data abstraction are derived from a more general concept, denoted as *information hiding* and introduced by Parnas (1971)
- Information hiding consists in defining and designing a module so that implementation details (both data and functions) that are not required to use that module are hidden, and thus not visible to other modules
- The advantages of *information hiding* are evident when it comes to apply changes to software products (*testing* and *maintenance* activities)

Information hiding example

Example of abstract data type (*C++ class*) with *information hiding*

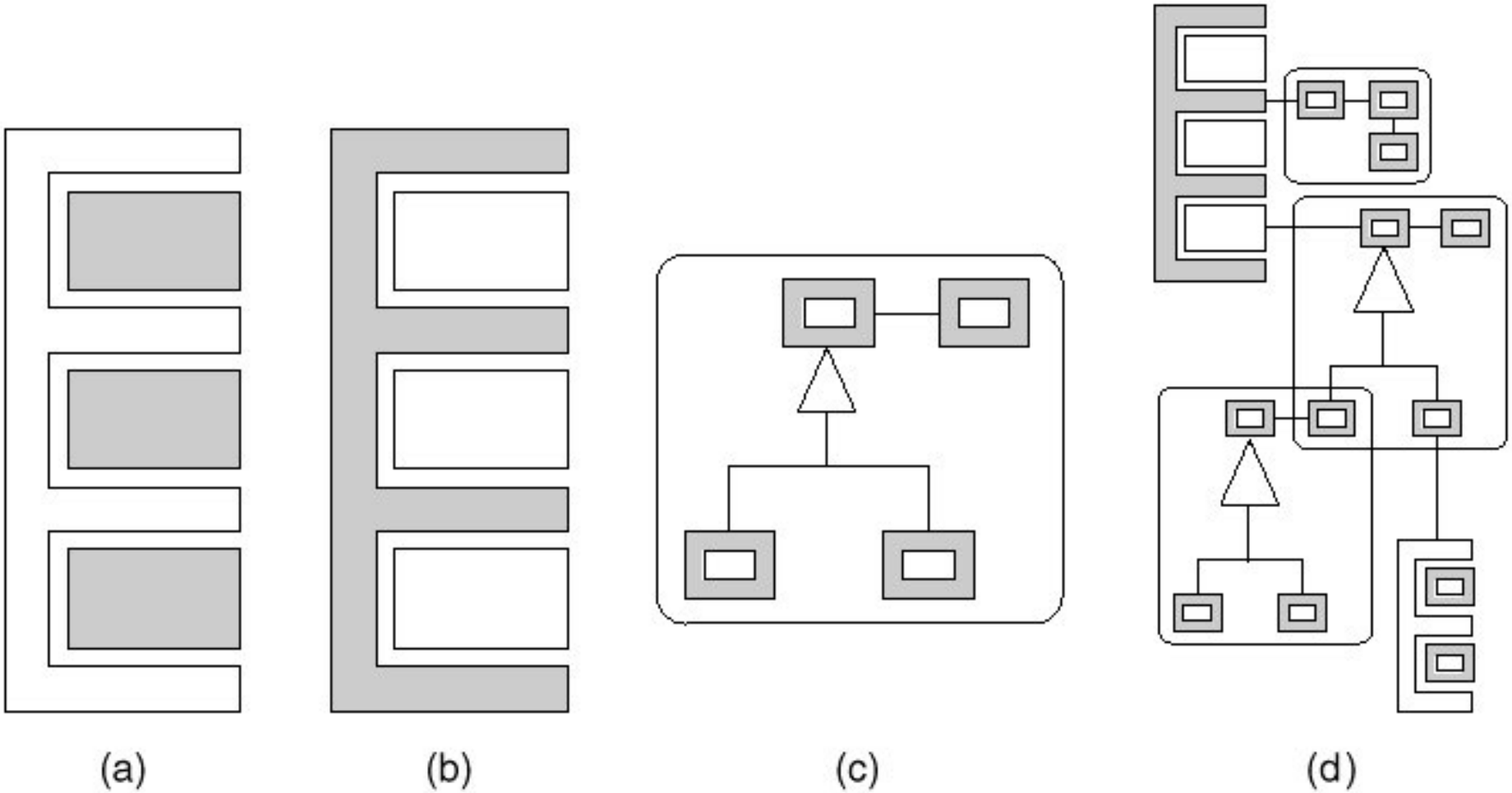
```
class JobQueue
{
   // attributes
   private:
      int      queueLength;      // length of job queue
      int      queue[25];        // queue can contain up to 25 jobs

   // methods
   public:
      void initializeJobQueue()
      {
         ....
      }
      void addJobToQueue (int jobNumber)
      {
         ....
      }
      void removeJobFromQueue (int& jobNumber)
      {
         ....
      }
} // class JobQueue
```

Reusability

- Reusability refers to the ability of using already available components into different products
- A reusable component does not merely identify a module or code fragment, but also design solutions, document sections, sets of test data and/or estimations of development cost and time
- Advantages:
  - significant reductions in terms of software development cost and time
  - reliability growth, due to the use of already validated components
- Reusability at design time applies to:
  (a) *software modules*
  (b) *application frameworks*, which incorporate the application logic of a design solution
  (c) *design patterns*, which identify design solutions to recurrent design problems
  (d) *software architectures*, which include (a), (b) and (c)

Reusability (2)



software module

application framework

design pattern

software architecture