Introduction to Objects

- The theory of modularity underwent steady progress during the 1970's and 1980's and objects resulted as a logical development within the theory of modularity.

Just in Case You Wanted to Know

Object-oriented concepts were introduced as early as 1966 in the simulation language Simula 67 [Dahl and Nygaard, 1966]. However, at that time the technology was too radical for practical use, so it lay dormant until the early 1980s when it was essentially reinvented within the context of the theory of modularity.

There are other examples in this chapter of the way that leading-edge technology lies dormant until the world is ready for it. For example, information hiding (Section 6.6) was first proposed by David Parnas in 1971 [Parnas, 1971], but the technology was not widely adopted until 10 years later when encapsulation and abstract data types had become part of software engineering.

It seems that we adopt new ideas only when we are ready to use them, not necessarily when they are first presented.

- Recall the Object–Oriented paradigm as explained on transparencies 8–10. Objects are the answer of (the responsible part of) the system's programmer's world to the fact that programs tend to become larger and more complex, in that objects provide a basis for the maintenance of such programs. They do so by adding *encapsulation* and *inheritance* to *abstract data* types, thereby increasing modularity and decreasing the incidence of regression errors during the maintenance phase.

- This chapter describes objects in the context of modularity as the next steps in the body of knowledge that begins with the concept of a module.

- The key definitions of this chapter — taken from chapter 6 of Schach's "Classical and O–O Software Engineering" — are briefly characterized in Figure 1 below.

Abstract data type a data type together with the actions performed on instantiations of that data type

Abstraction a means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details

Class an abstract data type that supports inheritance

Cohesion the degree of interaction within a module

Coupling the degree of interaction between two modules

Data encapsulation a data structure together with the actions performed on that data structure

Encapsulation the gathering together into one unit of all aspects of the real-world entity modeled by that unit

Information hiding structuring the design so that the resulting implementation details will be hidden from other modules

Object an instantiation of a class

Figure 1: Key definitions of this chapter

- And the progression of concepts in the theory of modularization beginning with modules and progressing to objects and beyond (as listed in chapter 6 of op.cit.) is listed in Figure 2 below.

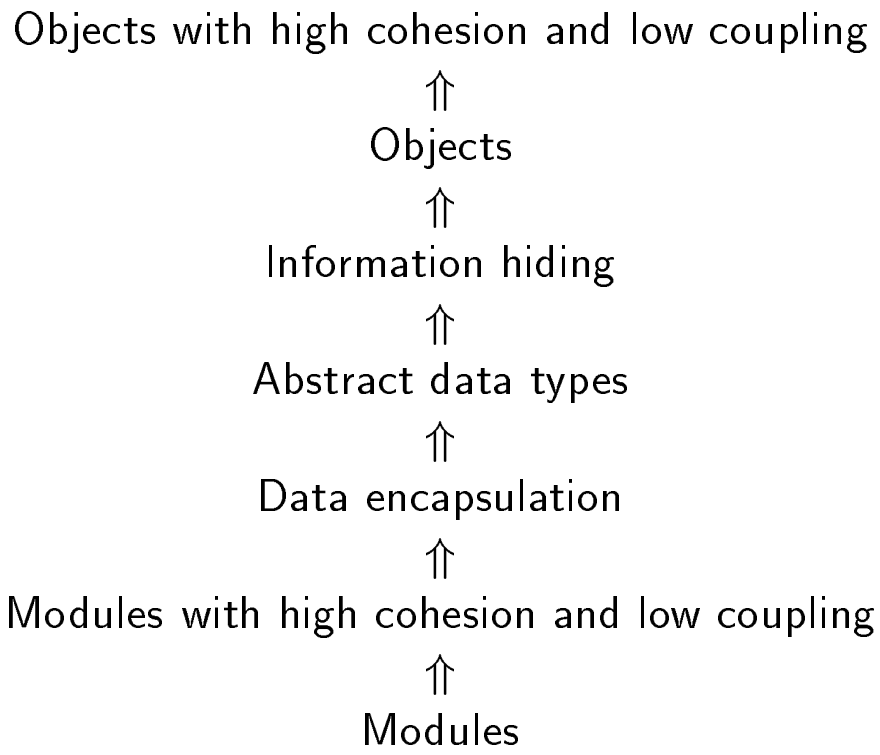

Figure 2: Major concepts of Chapter 6

- When a large product consists of a *simple monolithic* block of code, *maintenance* is a *nightmare*. Even for the author of such a monstrosity, attempting to debug the code is extremely difficult; for another programmer to understand it is virtually impossible. The solution is to break the product into smaller pieces, called *modules*.

- What is a module? Is the way a product is broken into modules important in itself, or is it important only to break a large product into smaller pieces of code?

- Yourdon and Constantine provide the following characterization of a module:

  > *"A module is a lexically continuous sequence of program statements, bounded by boundary elements, having an aggregate identifier."*

Examples of boundary elements are $begin \ldots end$ pairs in a block–structured language like PASCAL or ADA, or $\{ \ldots \}$ pairs in C or C^{++}.

- To understand the importance of modularization consider the following contrived example.

  A computer is characterized by:

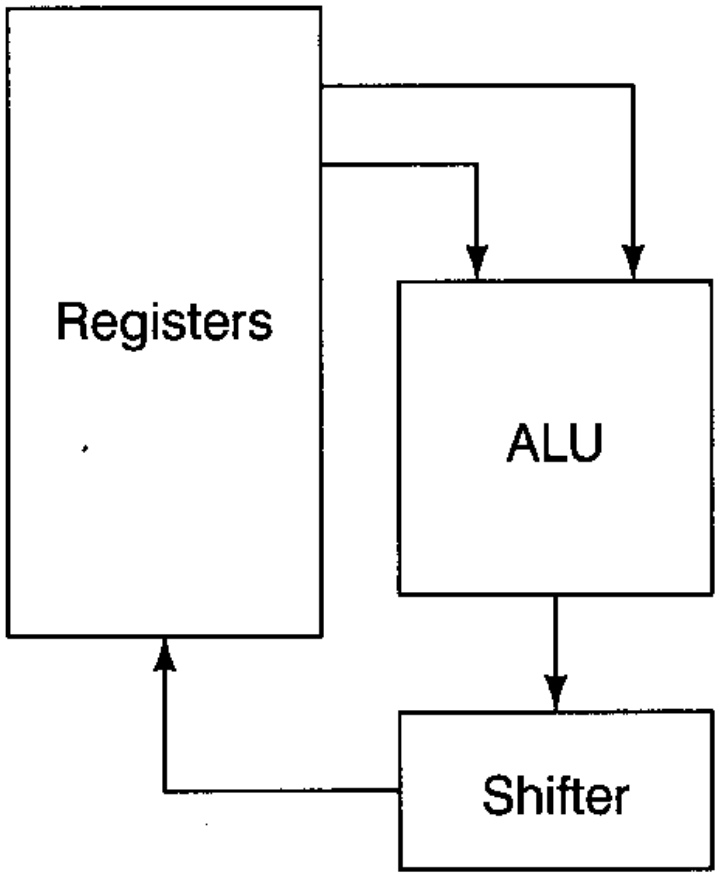

Figure 3: Design of computer

- To build such a computer first 3 chips are connected, of which Chip 1 contains the Registers, Chip 2 the Arithmetic Logical Unit (ALU) and Chip 3 a Shifter, as below in Figure 4.

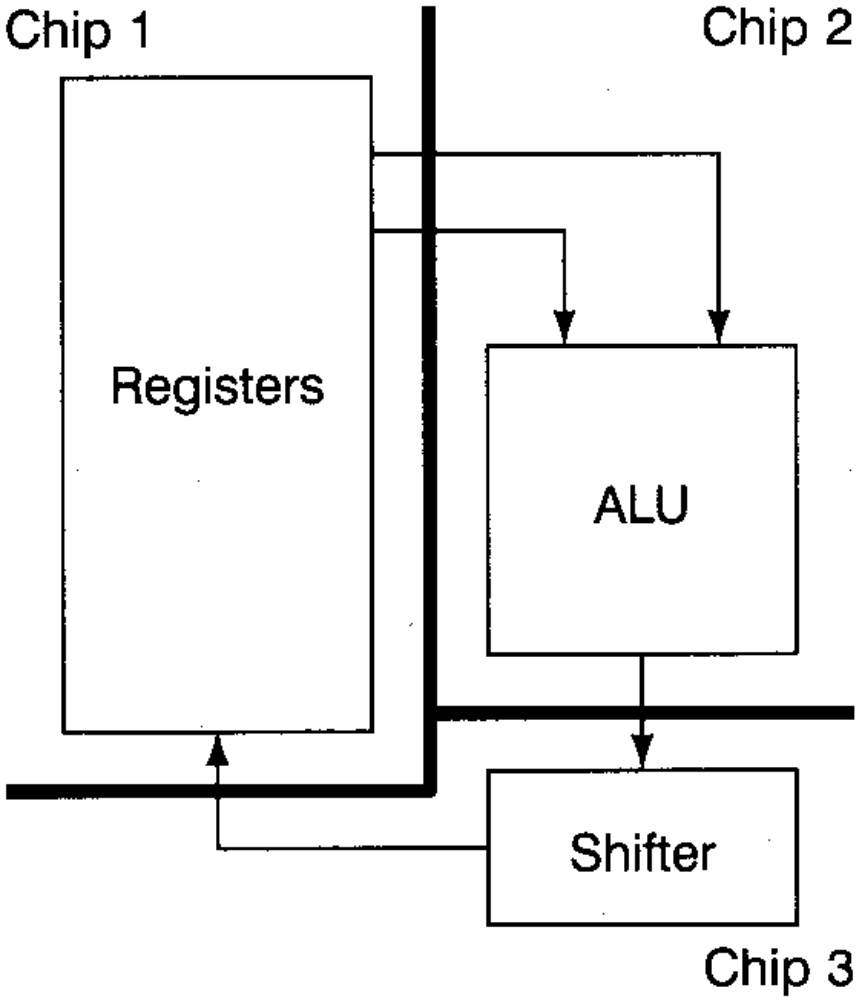

Figure 4: Computer of Figure 3 fabricated on 3 chips

- These chips utilize AND, OR and NOT gates (instead of NAND or NOR gates, as usual). Now these chips are redesigned as to contain all the AND gates on one chip, all OR gates on a second chip and all the NOT gates on a third one. The resulting "work of art" is shown in Figure 5 below:

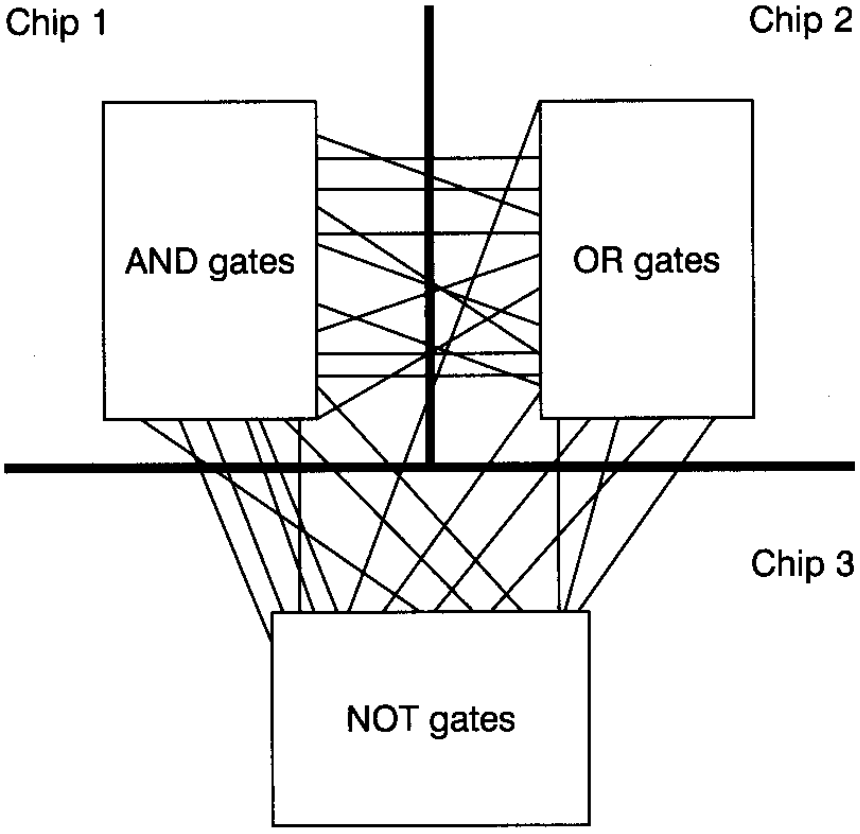

Figure 5: Computer of Figure 3 fabricated on 3 other chips

- Now it is in the intention of the designers that Figure 4 and 5 show functionally equivalent designs, i.e. they display hardware having the same overall behavior.

- What's their difference?

  1. Figure 5 is harder to understand than Figure 4, which is immediately recognized by anybody acquainted with digital logic.
  2. Corrective maintenance of the circuits shown in Figure 5 is *difficult*; it will be *difficult to locate the faults in Figure 5*, of which there will be many due to the unclarity of the design.
  3. A fault in the design of Figure 4 is *easier located* by determining whether it appears to be in the way the ALU works, the shifter works, or the registers work.
  4. If the computer of Figure 4 breaks down, it is *relatively easy to determine which chip to replace*; if the computer of Figure 5 breaks down, it is probably best to replace *all three chips*.

5. The computer of Figure 5 is difficult to *extend* or *enhance*; a new ALU or faster registers will necessitate a new design, while in case of the computer of Figure 4 it is easy to replace the appropriate chips.
6. Worst of all, the chips in Figure 5 cannot be *reused* in any new product.

- The point is that software products have to be designed to look like Figure 4, where there is "*a maximal relationship* ***within*** *each chip and a minimal relationship* ***between*** *chips*".

- The maintenance effort whether corrective (removing software errors), perfective (extending the functionality) or adaptive (adaptation to new hardware) is *reduced* when there is no such *maximal interaction within each module* and *minimal interaction between modules*

- This led to considering these two kinds of interaction separately:
  - that of $module\ cohesion$, the degree of interaction within a module, which should be $high$, and
  - that of $module\ coupling$, the degree of interaction between modules, which should be low.

Module Cohesion

- Meyers distinguishes at least *7 levels of cohesion* as in Figure 6 below.

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

Figure 6: Levels of cohesion

- We won't define all these concepts but give some examples.

- An example of a module with *coincidental cohesion*, i.e. low cohesion, is in the module:

  print next line; reverse the string of characters comprising the second argument; add 7 to the fifth argument; convert the fourth argument to floating point.

- Why is such a module bad?

  1. From the viewpoint of trying to understand this module (during maintenance) it is bad, because it lacks a *clear purpose*; errors *cannot* be easily traced *to one of its components*; it has little structure.
  2. It is unlikely such modules can be *reused* — the money spent on its design can never be recouped.

- A module has *temporal cohesion* when it performs *a series of actions related in time*, i.e. the module:

  open old_master_file, new_master_file, transaction_file, and print_file, initialize sales_district_table, read first_transaction_record and first old_master_file record.

The problem with this module is that its actions are weakly related to one another (almost independent), but more strongly to actions in other modules. Thus if the *sales_district_table* is *changed*, a number of such modules *will also have to be changed*. This increases the occurrence of regression faults.

- Consequently, *it is much better to have all operations on the sales_district_table inside one module* — this is called *informational cohesion*. A module has informational cohesion *if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure*, see, e.g., Figure 7 below.

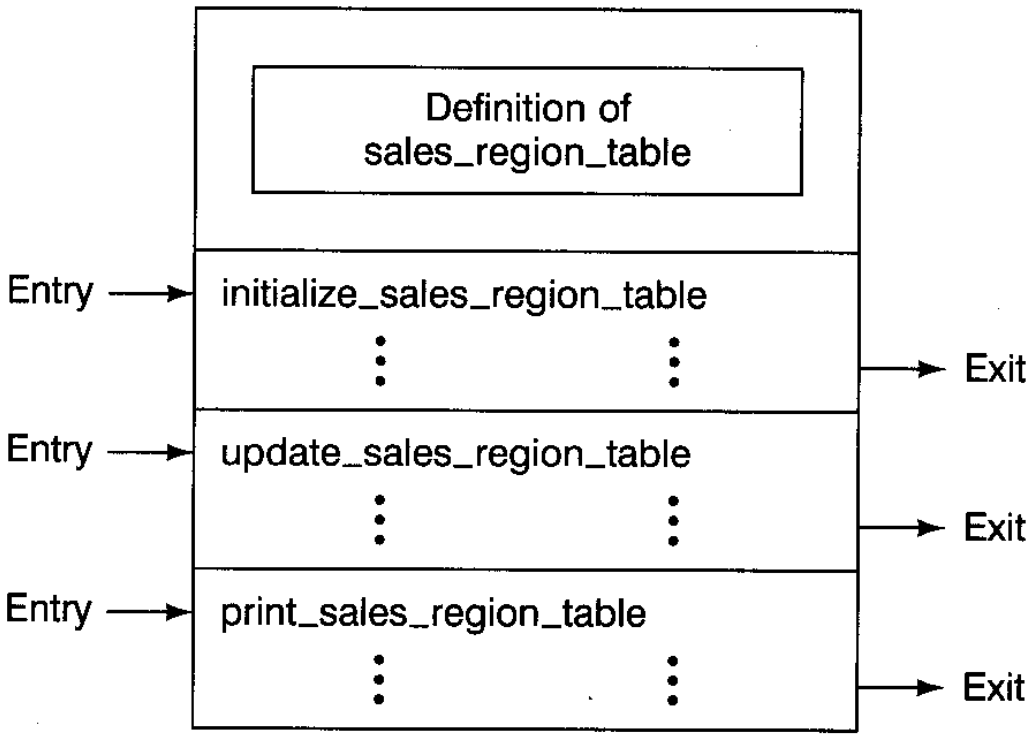

Figure 7: Module with information cohesion

- A module that performs exactly one action or achieves a single goal has *functional* cohesion. Such modules can be *reused* and lead to *fault isolation*. Examples of modules with functional cohesion are suggested below:
  - *get temperature of a furnace,*
  - *calculate sales mission,*
  - *write to diskette.*

Module Coupling

- Recall *coupling* in the degree of interaction between two modules — it should be low.

- Again coupling is split into levels, as in Figure 8 below:

  5. Data coupling (Good)
  4. Stamp coupling
  3. Control coupling
  2. Common coupling
  1. Content coupling (Bad)

Figure 8: Levels of coupling

- Examples of *context coupling* are:
  - *Module p branches to a logical label of module q.* It will be clear that *redesign of q leads to redesign of p* — i.e. p and q cannot be changed independently of each other.
  - *Module p refers to logical data of module q in terms of some numerical displacement within q.* The same criticism applies.

  Such modules are *inextricably intertwined.*

- An example of *control coupling* is:
  - *A module p calls a module q, and q passes back a flag to p saying "I am unable to complete my task; accordingly write error message ABC123."*
    The major difficulty that arises as a consequence of such coupling is that the two modules are not independent; module q, the called module, *has to be aware of the internal structure of module p, its caller.*

- *Stamp coupling* occurs if some data structure is passed as an argument, but the called module operates on only *some* of the individual components of that data structure.
  - Such coupling may lead to computer crime, and again necessitates to look for which components are updated upon, upon maintenance, in case of a need for changing the code of such modules, which may lead to regression faults if one such component is overloaded.

- Two modules are *data coupled* if all arguments are homogeneous data items, i.e., *every argument is either a simple argument or a data structure in which all elements are used by the called module.* Data coupling between two modules is the more desirable case.

Data Encapsulation

- Consider the following example:
  - Any job submitted to a particular computer is classified in having *high priority*, *medium priority* or *low priority*. And consider the restricted problem of batch jobs queuing up for *memory access*.
  - I.e., there are three queues for incoming batch jobs, one for each priority level.
  - When a job is *submitted* by a user it is *added* in the appropriate queue, when the operating system decides a job is *ready to run* it is *removed* from its queue and memory is allocated to it.

- There are a number of different ways to build this portion of the product, illustrated in Figure 9 and 10 below.

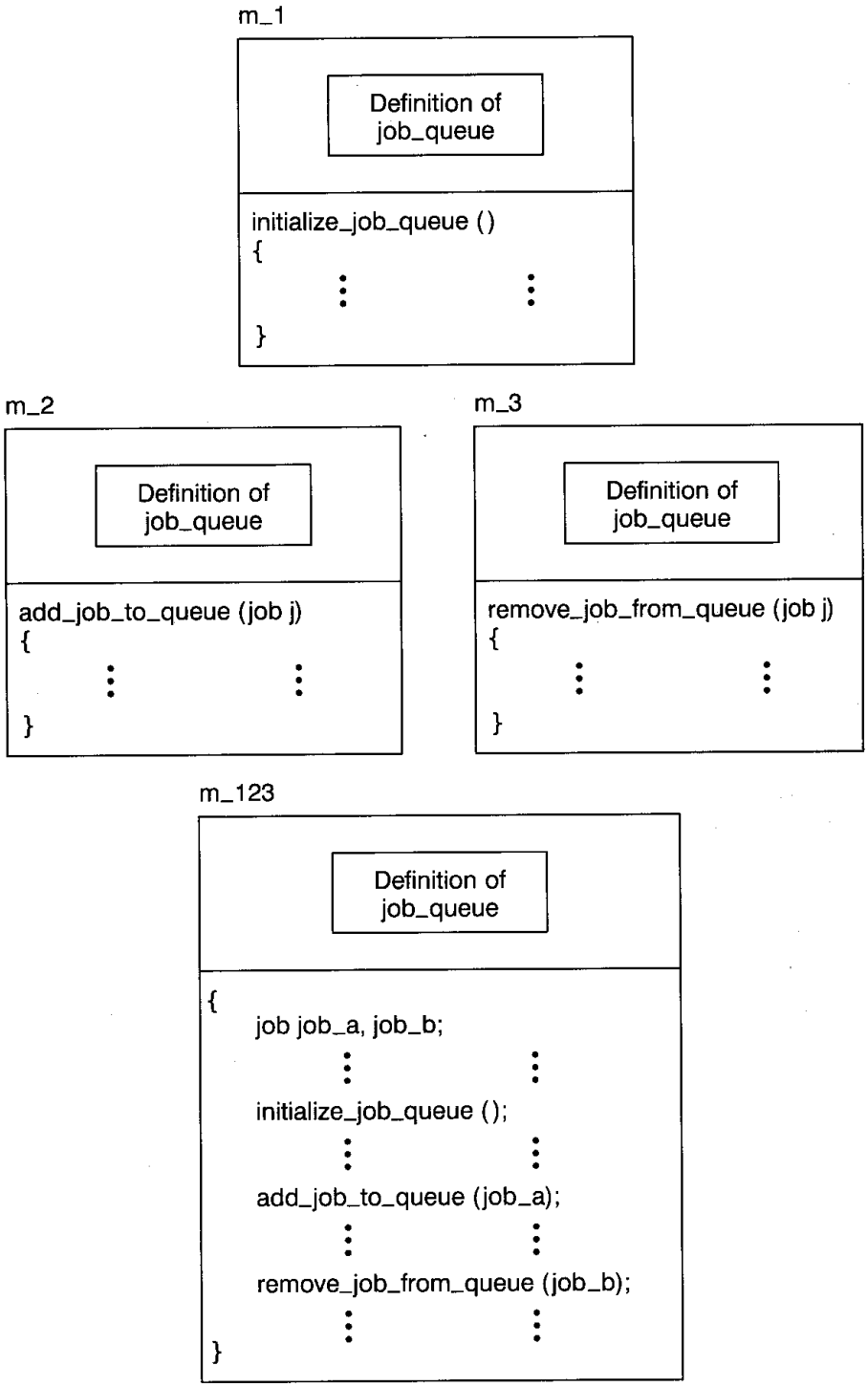

Figure 9: One possible design of job queue portion of operating system

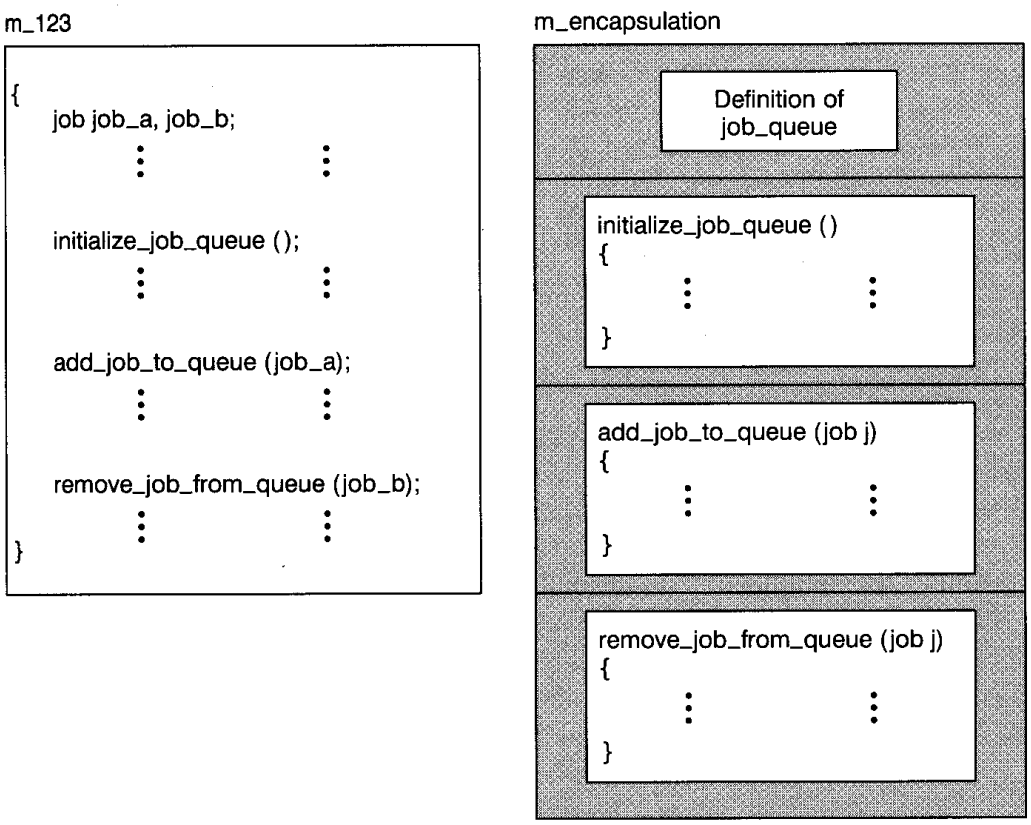

Figure 10: Design of job queue portion of operating system using data encapsulation

- The modules of the design of Figure 9 have low cohesion, because their actions on the job queue are spread all over the product. If the way a job queue is implemented is changed, i.e. to a linked list of records instead of as a linear list, modules *m_1*, *m_2* and *m_3* as well as *m_123* have all to be changed, too.

- The point of Figure 10 is that *this is not the case*, Figure 10 is an implementation of *data encapsulation*, i.e. *a data structure* (i.e. job queue), *together with all the actions to be performed on that data structure.*

What is the Advantage of Data Encapsulation?

- From the viewpoint of development, the developer is able *to conceptualize the problem as a higher level, that of jobs and job queues, rather than at the lower level of records and arrays.*

- I.e., at the higher level stage it is *entirely irrelevant* how the job queue is implemented. Once the complete higher–level design has been obtained, the *second step* is to design the lower level components in terms of data structures they will be implemented in. At this level *the design totally ignores the intended use of jobs, job queues and actions.*

- Thus, during the first step merely the *existence* of the lower level is *assured*, while at the second step *the existence of the higher level is ignored*. Consequently, *a change of implementation at the lower level does not affect the higher level at all*.

- From the viewpoint of maintenance, data encapsulation of data structures is advantageous because *data structures as such are unlikely to change*. Only the specific way they are implemented changes. Data encapsulation *offers a way of coping with that change by factoring it out at the higher level of use of data structure operations and coping with it at the lower level of their implementation*.

- This is illustrated in Figures 11 and 12 below.

```
// See the Just in Case You Wanted to Know box for details

static int queue_length;
static int job_queue[25];

void initialize_job_queue(void)
/* empty job queue has length 0 */
{ queue_length = 0; }

void add_job_to_queue(int job_number)
/* add job to end of job queue */
{
job_queue[queue_length] = job_number;
queue_length = queue_length + 1
}

void remove_job_from_queue(int& job_number)
{
job_number = job_queue[0];
queue_length = queue_length - 1
for(int k=0; k<queue_length; k++)
   job_queue[k] = job_queue[k+1];
}
```

Figure 11: C++ implementation of compilation unit m_encapsulation.c

```
struct job_record
{
  int                job_no;        // number of the job (integer)
  struct job_record  *in_front;     // pointer to job record in front
  struct job_record  *in_rear;      // pointer to job record behind
};

struct job_record    *front_of_queue, *rear_of_queue;
                                    // pointers to front, rear of queue

void initialize_job_queue (void)
{
  /*
   * initialize the job queue by setting front_of_queue and rear_of_queue to NULL
   */
}

void add_job_to_queue (int job_number)
{
  /*
   * Create a new job record,
   * place job_number in its job_no field,
   * set in_front field to point to the current rear_of_queue
   * (thereby linking the new record to the rear of the queue),
   * and set in_rear field to NULL.
   * Set in_rear field of record pointed to by current rear_of_queue
   * to point to the new record (thereby setting up a two-way link), and
   * finally, set pointer rear_of_queue to point to this new record
   */
}

void remove_job_from_queue (int& job_number)
{
  /*
   * set job_number equal to the job_no field of the record at the front of the queue,
   * update the front_of_queue pointer to point to the next item in the queue,
   * set the in_front field of the record that is now the head of the queue to NULL
   */
}
```

Figure 12: C++ implementation of compilation unit m_encapsulation.C using a two–way linked list

- Figure 11 illustrates an C^{++} implementation of compilation unit $m\_encapsulation.C$ in terms of an array of up to 25 job numbers. The reserved word **static** ensures that $queue\ length$ and job queue are not visible outside $m\_encapsulation.C$, and can be modified only inside that compilation unit. Each job number is represented as an integer. Also, the ampersand & is necessary in the declaration of function $remove\ job\ from\ queue$ because its argument is passed by reference.

- Figure 12 features an implementation of that module using a two–way linked list, i.e. using the two–way linked job_record data structure illustrated in Figure 13 below.

```
struct job_record
{
  int job_no;                       //number of the job
  struct job_record *in_front; //pointer to job record in front
  struct job_record *in_rear;  //pointer to job record behind
};
```

Figure 13: C++ specification of two–way linked job_record

- Next consider Figure 14 below, featuring an c^{++} implementation of *m_123*.

```
void m_123(void)
{
   extern void initialize_job_queue (void);
   extern void add_job_to_queue (int job_number);
   extern void remove_job_from_queue (int&  job_number);
   int job_a, job_b;

    // various statements

   initialize_ job_queue();

   // more statements

   add_job_to_queue (job_a);

    // still more statements

   remove_job_from_queue (job_b);

    // further statements

}
```

Figure 14: C++ implementation of *m_123*

- The **extern** statements in m_123 above indicate that the functions *initialize_job_queue*, *add_job_to_queue*, *remove_job_from_queue* have been defined elsewhere (the linker will locate their definitions in *m_encapsulation*: *the interface information regarding these three functions*.

- Since this interface is the same for the implementation of *m_encapsulation.C* of Figure 11 and of Figure 12, *it does not matter from the viewpoint of* ***m_123*** *which implementation of this compilation unit is taken*.

- *Thus data encapsulation supports the implementation of data abstraction in a way that simplifies product maintenance and reduces the chance of regression faults*.

Abstract Data Types

- A major difficulty with both implementations of *m_encapsulation* is that *they apply only to one queue.*

- Rather than specifying a data structure, together with relevant actions, it would be more useful to have a *data type*, together with the actions to be performed on *instantiations* of that data type. Such a construct is called an *abstract data type* and is illustrated in Figure 15 below.

```
struct job_queue_type
{
  // data elements
  int                    queue_length;     // length of job queue
  int                    job_queue[25];    // queue can contain up to 25 jobs

  // operations on the data elements
  void initialize_job_queue (void)
  {
    // body of function unchanged from Figure 6.18
  }

  void add_job_to_queue (int job_number)
  {
    // body of function unchanged from Figure 6.18
  }

  void remove_job_from_queue (int& job_number)
  {
    // body of function unchanged from Figure 6.18
  }
};
```

Figure 15: Job queue implemented in C++ as abstract data type. The code is stored in header file job_queue.h

- This C++ implementation of the job queue abstract data type uses a *struct* construct. Compare Figure 15 with Figure 11.

  - One difference is that the declarations of $queue\_length$ and job_queue are encapsulated in **struct** job_queue_type in Figure 15, whereas there are independent declarations of $queue\_length$ and job_queue as **static** variables in Figure 11.
  - The second difference is that the three functions, namely $initialize\_job\_queue$, $add\_job\_to\_queue$ and $remove\_job\_from\_queue$ are now members of the structure.

- The abstract data type of Figure 15 is stored in header file $job\_queue.h$.
  Figure 16 below shows how this abstract data type may be utilized. Two job queues are instantiated, $queue\_1$ and $queue\_2$. E.g., the statement $queue\_1.initialize\_job\_queue()$ means: apply function $initialize\_job\_queue$ to structure $queue\_1$.

- The **#include** statement causes the entire file *job_queue.h* to be copied into module *m_123* below.

```
#include "job_queue.h"


void m_123(void)
{
  job_queue_type     queue_1, queue_2;
  int                job_a, job_b;
 // various statements
 queue_1.initialize_job_queue();
 // more statements
 queue_2.add_job_to_queue(job_a);
 // still more statements
  queue_1.remove_job_from_queue(job_b);
 // further statements
}
```

Figure 16: C++ module m_123 using abstract data type of figure15

- Abstract data types support both *data abstraction* (i.e., the product is produced in terms of high–level concepts such as *jobs*, *job queues*, and the operations that are performed on job queues) and *procedural abstraction. The latter extends the programming language by supplying the developer with other user–defined functions that are part of the language as originally defined.* These functions are defined in terms of lower level actions of the language, and so on, until the lower level is reached. Important are two points:
  - The designer can *ignore* the level below, which will be handled at the next level of abstraction, that is, the next refinement step.
  - The designers can also ignore the level above, a level that is irrelevant from the viewpoint of designing the current level.

Information Hiding

- The two types of abstraction discussed, *data abstraction* and *procedural abstraction*, are instances of a more general design concept put forward by David Parnas, namely *information hiding*.

- Parnas' ideas are directed towards future maintenance:

  > Before a product is designed, a list should be made of *implementation decisions likely to change in the future*. Modules should be designed so that these implementation details of the resulting design *are hidden from other modules*. Thus any future change is *localized* to one specific module. Because the details of the original implementation decision *are not visible to other modules*, changing the design *cannot affect any other module*.

- Let's see how these ideas can be used in practice:
  - First, consider the data encapsulation implementation of Figures 11 and 14. The only implementation details of module $m\_encapsulation$ used in module m_123 are the names and the interfaces of the functions that operate on a job queue; **static** variables $queue\_length$ and job_queue are not visible outside $m\_encapsulation$. Consequently, no change needs to be made to module m_123, when changing $m\_encapsulation$ to that of Figure 12.
  - Now consider the abstract data type implementation of Figures 15 and 16. The *primary reason* for using the abstract data type is to *ensure that the contents of a job queue can be changed exclusively* by invoking one of the three functions of Figure 15. *However this goal has not been achieved as yet!*

- As a consequence of the statement **#include** "*job_queue.h*", all of the header file *job_queue.h* is visible to module *m_123*. In particular, the precise details of how *job_queue* is implemented are accessible. As a result, it is legal in C^{++} to use an assignment statement such as

  queue_1.job_queue[7] = -5678

  anywhere in *m_123* to change *queue_1*.
- Consequently, it is possible to change the contents of a job queue *without using any of the three actions of the data type*. In addition to lowering cohesion and increasing coupling *the product may now be vulnerable to computer fraud*.
- Fortunately, there is a way out. The design of C^{++} provided for information hiding *even within a structure specification*. This is shown in Figure 17, using the concepts of **private** and **public** declarations. Apart from these concepts, Figure 17 is identical to Figure 15. But now the exact way that job queues are implemented is **private**, and invisible to the outside.

```
struct job_queue_type
{
  // data elements
  private:
    int                 queue_length;     // length of job queue
    int                 job_queue[25];    // queue can contain up to 25 jobs

  // operations on the data elements
  public:
    void initialize_job_queue (void)
    {
      // body of function unchanged from Figure 6.18
    }

    void add_job_to_queue (int job_number)
    {
      // body of function unchanged from Figure 6.18
    }

    void remove_job_from_queue (int& job_number)
    {
      // body of function unchanged from Figure 6.18
    }
};
```

Figure 17: Abstract data type implementation with information hiding, correcting the problem of Figure 15

– The diagram in Figure 18 shows how a structure with private data enables a C++ user to implement an abstract data type with full information hiding.

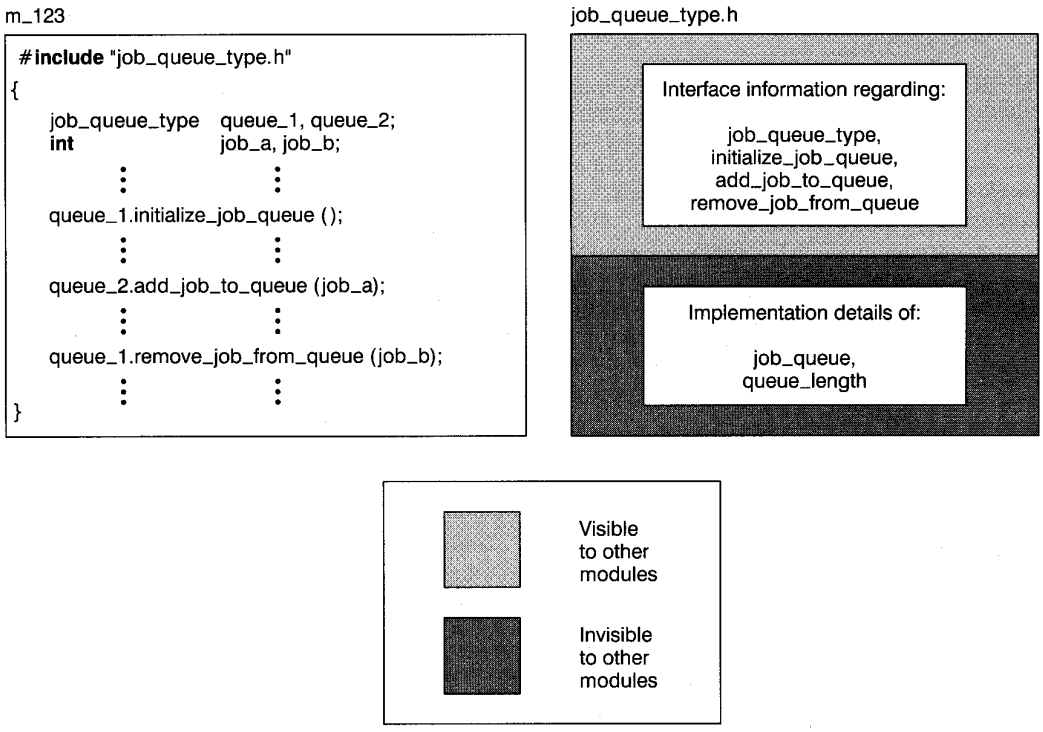

Figure 18: Representation of abstract data type with information hiding achieved via structure with private data (Figures 16 and 17)

Objects

- Consult again Figure 19 below.

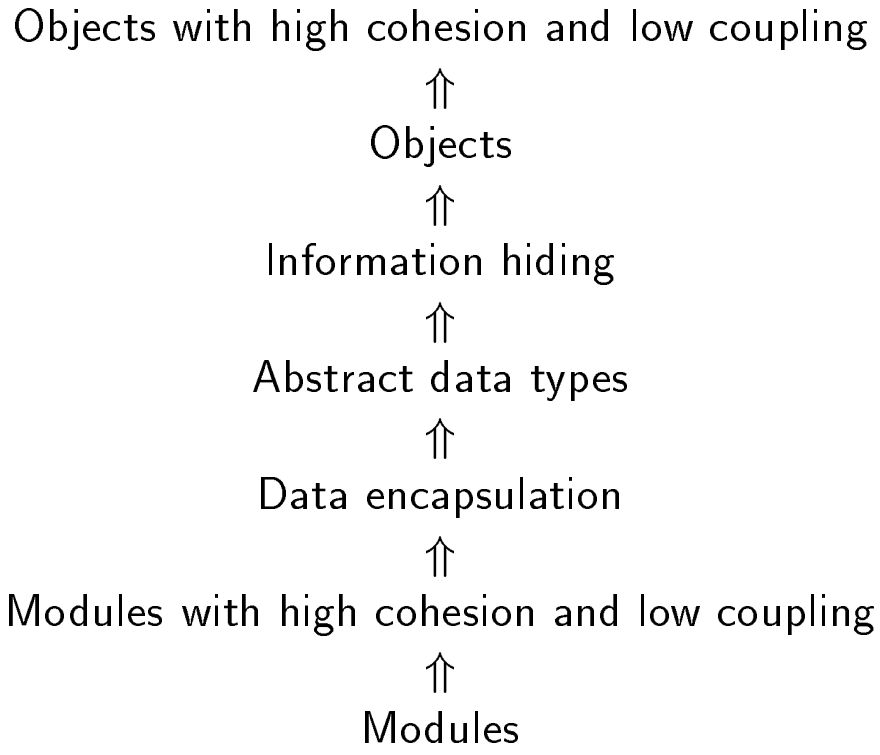

Figure 19: Major concepts of Chapter 6

- Objects are simply the next step in the evolution of modularization concepts after abstract data types and information hiding.
  I.e. they are abstract data types or modules with information hiding.

- In more detail,

  an incomplete definition of an object is an instance of an abstract data type.

- However, objects also feature *inheritance*. The basic idea is that new data types can be defined as *extensions* of previously defined types, rather than defining them from scratch, using the concept of *class*.

- A class is an abstract data type with information hiding that supports *inheritance*. An object is an instantiation of a class. See Figure 20 below:

```
class Human_Being
{
  private:
    int                        age;
    float                      height;
    enum {male, female}        gender;

  public:
    // declarations of operations on Human_Being
};

class Parent: public Human_Being
{
  private:
    char                       name_of_spouse[20];
    int                        number_of_children;

  public:
    // declarations of operations on Parent
};
```

Figure 20: C++ implementation of Figure 21

- In Figure 20 a class, i.e. an abstract data type with information hiding with operations age, $height$ and $gender$ defined, called $Human\_Being$. However, it also features an extension of that class by the operations $name\_of\_spouse[20]$ and $number\_of\_children$, called $Parent$.

- The relationship between classes $Parent$ and $Human\_Being$ is depicted in Figure 21.

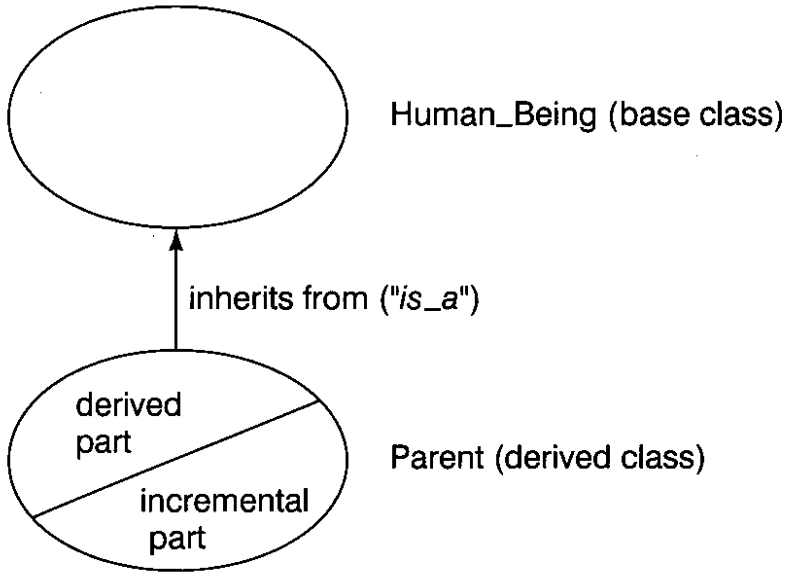

Figure 21: Derived types and inheritance

- Parent is a $derived$ class, and inherits all the attributes from, i.e., "is_a", $Human\_Being$, which is the $base\ class$ of this example.

- The last development in the theory of modularization is that of $polymorphism$ and $dynamic\ binding$, for which we refer to Section 6.8 of Schach's op.cit.