
Chapter 13 of text (Implementation diagrams)
&
Chapter 14 of text (Packages, subsystems, models)

IMPLEMENTATION DIAGRAMS

Component Model

-
-

[*classifiers*]

Development View

(Dependencies between parts of code)

Deployment Model

-
-

[*instances*]

Physical & Process Views

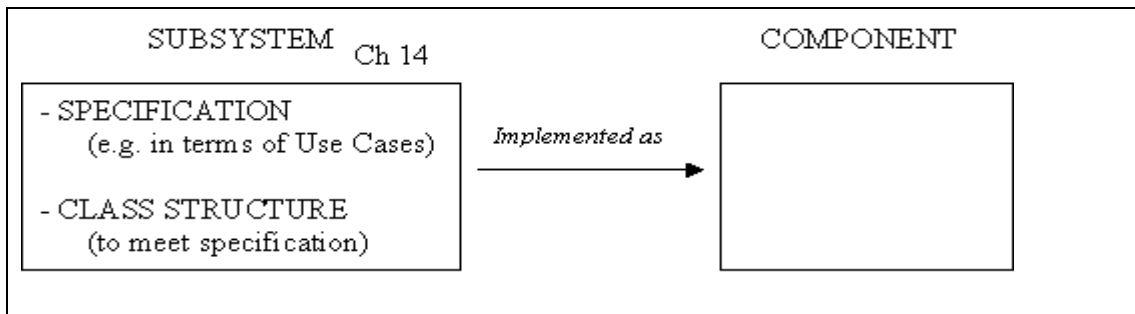
(Structure of run-time system, including hardware)

• Definition of a component (*for purpose of chapter 13 specifically*):

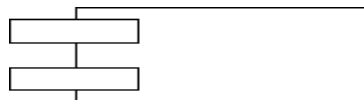
"A distributable piece of implementation of a system, including software code (source, binary, executable) but also including business documents etc., in a human system" [*from "UML notation guide version 1.4" - see text*]

Note: In rest of text we think of a component, in a less specific way, as a "plug-in" part of a system design"

- In sense of Chapter 13, we may think of a "component" as an implementation of a "subsystem". Thus,

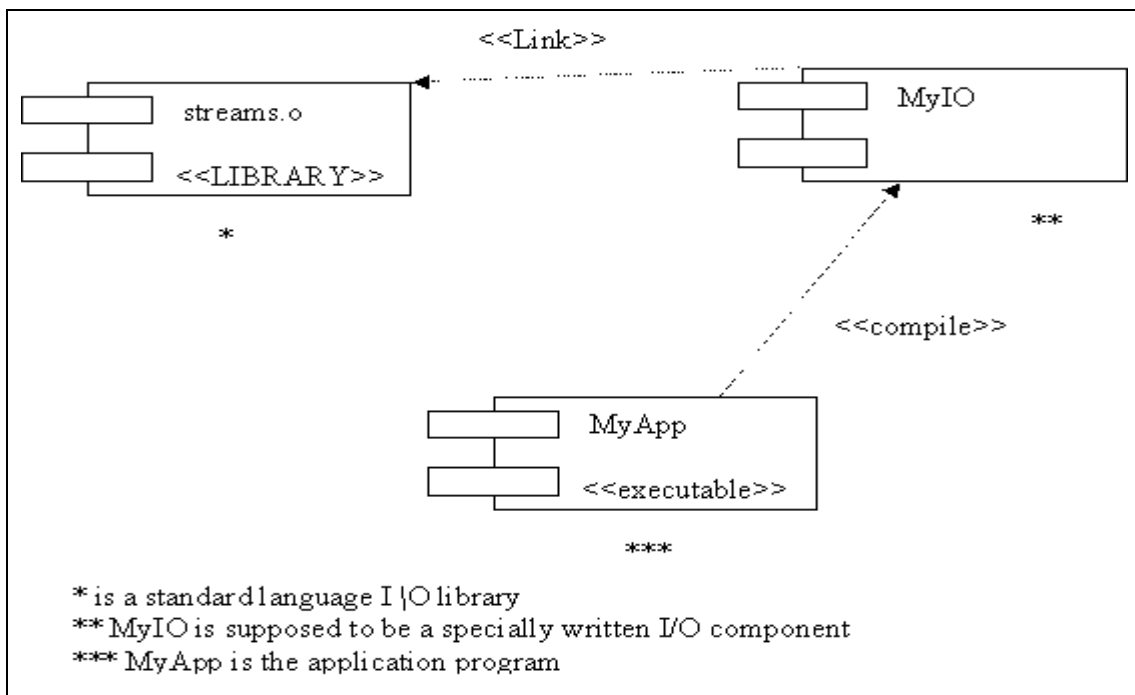


- In UML, components in the above sense are depicted by



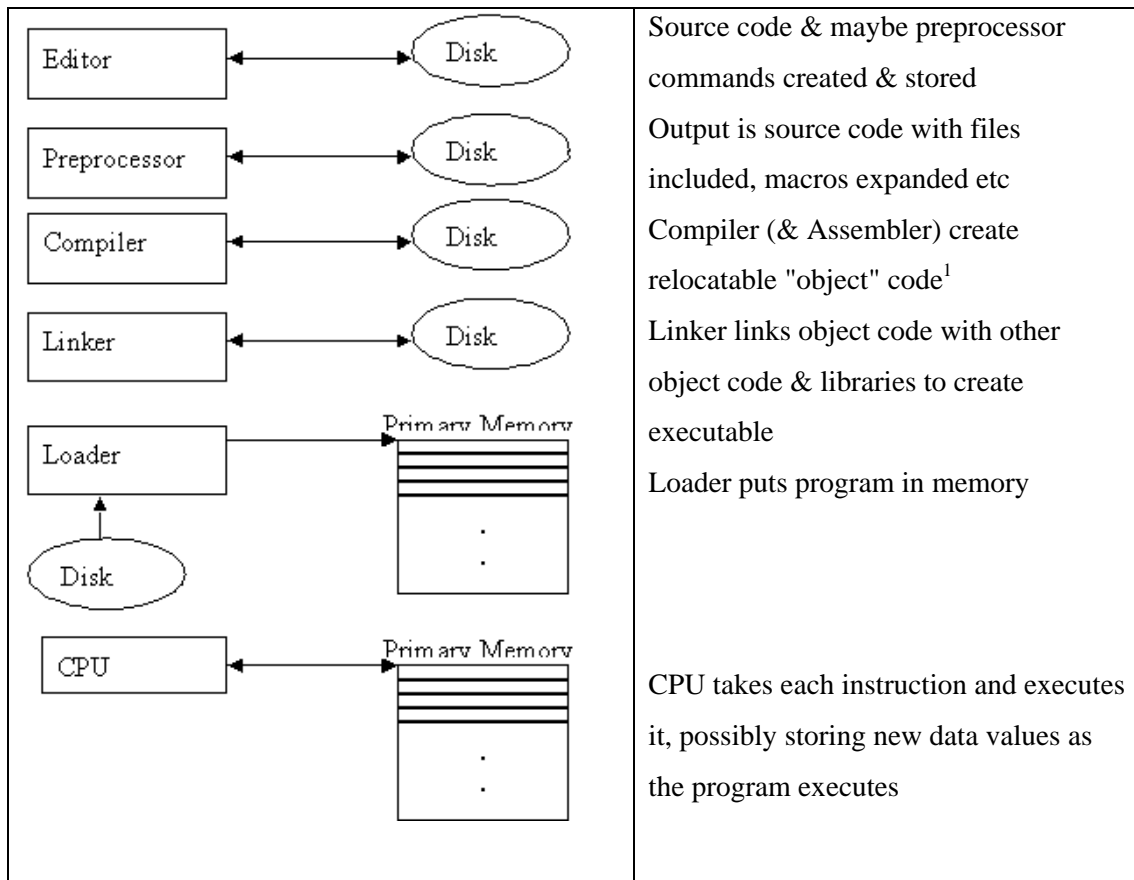
"----->" shows dependency between components

- Example: Compile-time dependencies for a C++ program - Fig 13.1 of text



[ASIDE to illustrate some typical component dependencies:

Includes outline of typical "phases" in development of a piece of software.



Some notes:

(1) Usually, some of these steps are combined to facilitate programmers.

(2) Here is a simple example of a "macro" for the C language, where prefix "#" denotes a preprocessor directive or command:

```

$cat test.c                                Display of source program (Unix cmd)
#define square(n)(n*n)
main()                                      The source program including definition of a
{                                           macro through the preprocessor
    printf("%f\n",27.0/square(3.0));
}
$cc test.c                                  Invoke preprocessor, compiler, assembler, linker (Unix)
$a.out                                       Run executable program (Unix)
3.000000
$

```

¹ Nothing to do with object oriented programming!

(3) Note that the situation may be more complex than depicted above. For example, Java programs are typically (always?) executed on a Java Virtual Machine which in turn is executed on the actual machine.

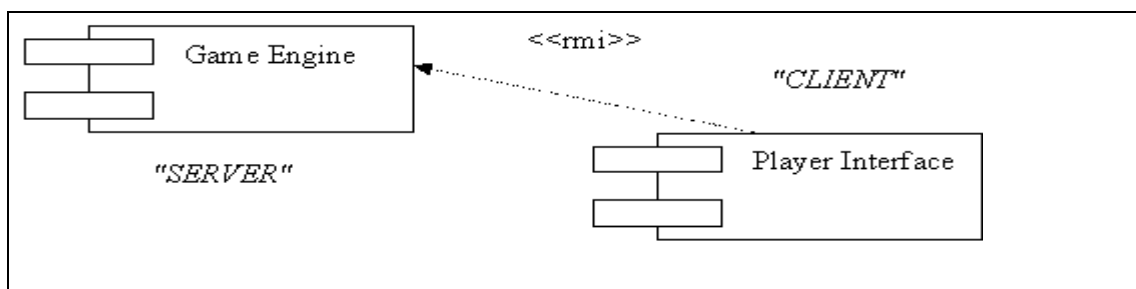
(4) If one is using an object oriented approach involving inheritance then some dependencies can be noted, as follows (case of C++ but analogous for other languages?). Suppose we have a base class from which we want a derived (or sub-) class to inherit attributes and behaviour. In C++ this is done without the derived class having access to the base class's source code, but the derived class does need to be able to link to the base class's object code (so there is a <<link>> dependency). However, the derived class does need access to header file(s) (related to the base class) at compile time.

End of ASIDE]

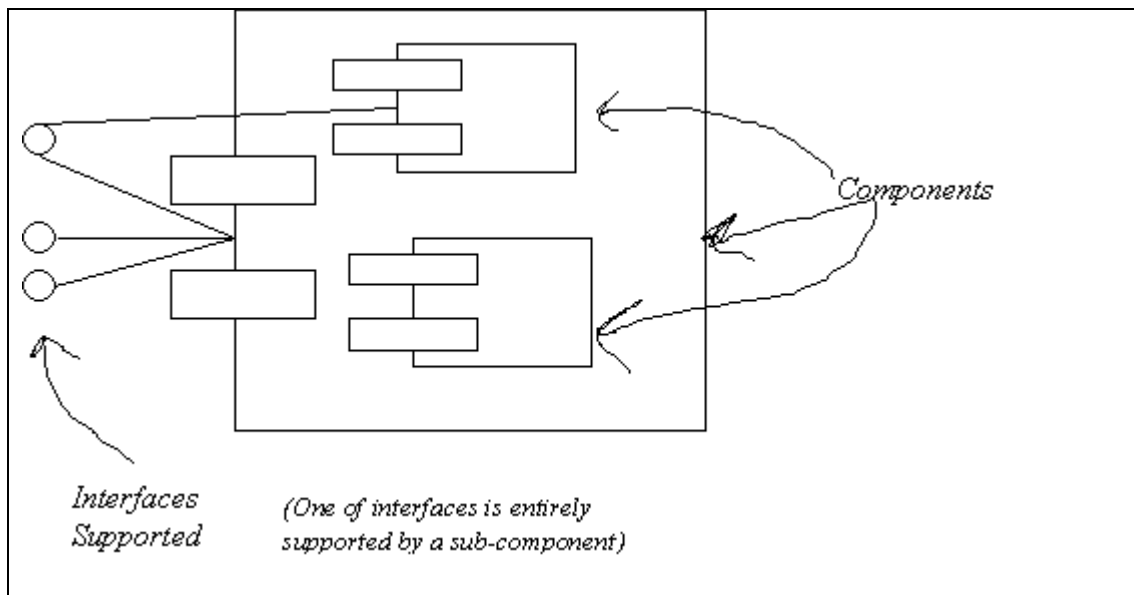
- There are various common kinds of components:

Kind of Component	Dependency [names are suggestions only]
<u>Source code</u> (e.g. file containing code of a class)	<<compile>> Any components which have to be available when it is compiled
<u>Binary object code</u> (e.g. a class library)	<<link>> Any object code with which it must be linked to form an executable
<u>Executable application</u> (e.g. a bought-in spreadsheet, data base manager, etc)	<<rmi>> Any other executable programs it needs to interact with it at run-time

- Example (Figure 13.2 of text) showing run-time dependency:



- General UML notation for components (including interfaces):



- "Classifiers and Instances", Panel 13.1 of text (some already mentioned):

CLASSIFIER	INSTANCE
Class	Object
Use Case	Scenario
Actor	Actor
Component	Component
Subsystem	Subsystem

-- E.G. a component type might be an executable file while corresponding instances would be instances of the file that are running at the same time.

-- *Notation* to distinguish classifier and instance in UML:

Same icon used for both	Underline icon when it represents an instance: (e.g. <u>instanceName</u> : <u>classifierName</u> where <u>instanceName</u> is optional)
-------------------------	---

- Remark: In component models we are concerned with *classifiers* but in deployment models concern is with *instances*.

- DEPLOYMENT MODEL

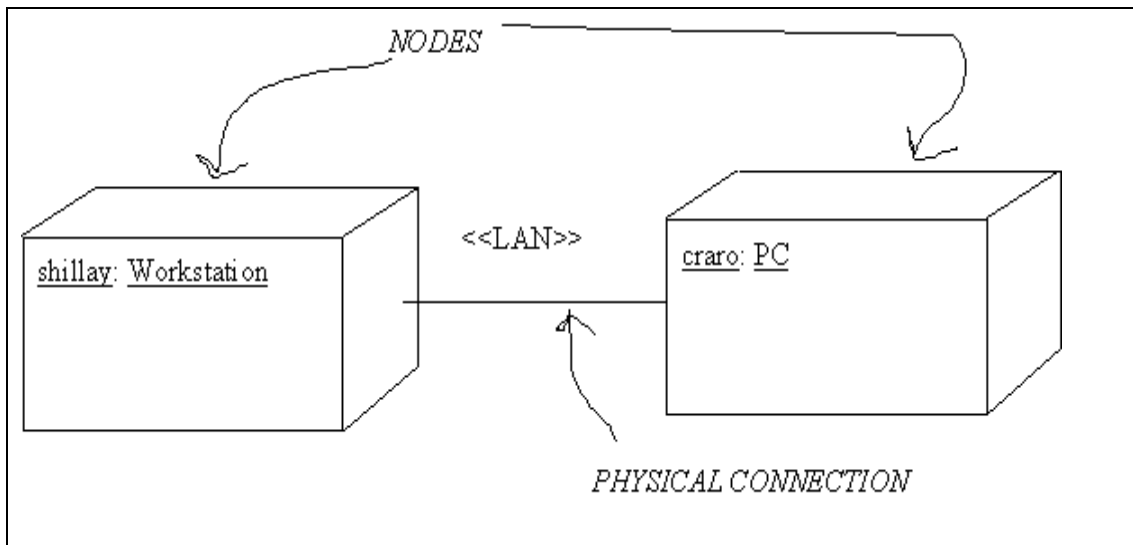


Figure 13.3 (of text): A deployment model *without* the software

[Shows how to depict physical connections between HW items]

Note: Textual specifications of links and/or nodes may be included to augment the diagram

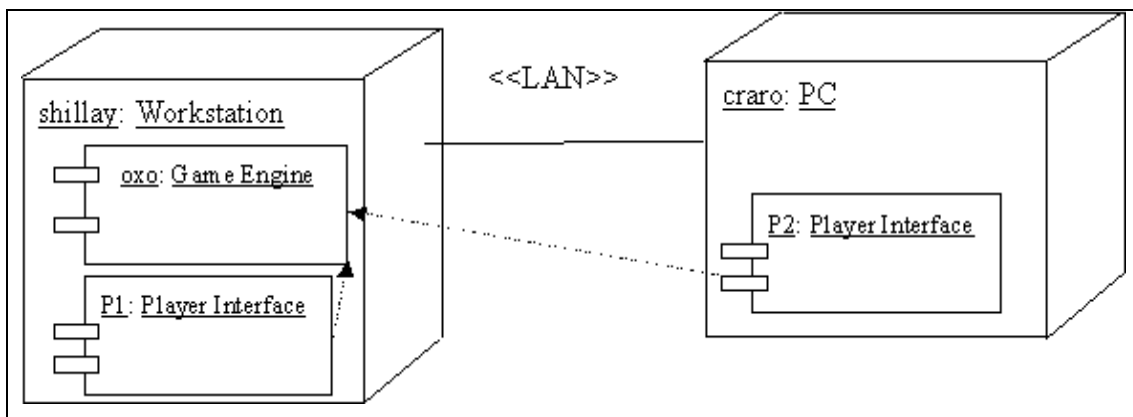
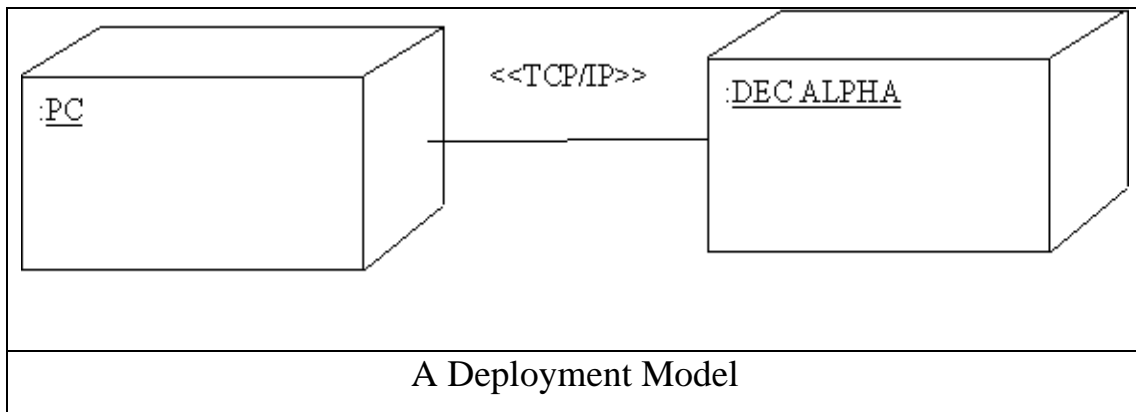
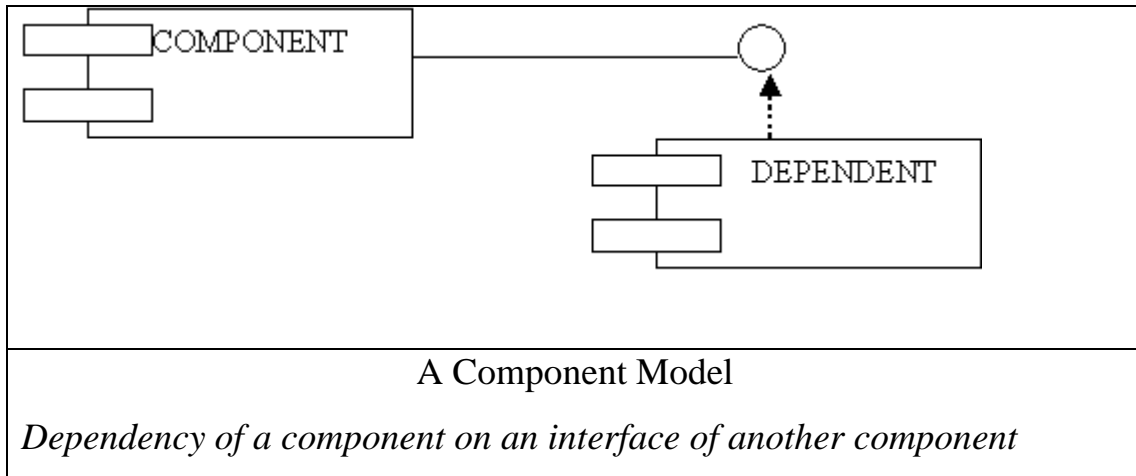


Figure 13.4 (of text): A deployment model *with* the software

[The software components are running *instances*]

Note: Should read Panel 13.2 on when deployment issues are addressed in a project - should often be very early in a project's life-time.

- A FEW MORE EXAMPLES



- **Packages, Subsystems, Models** (*see Ch. 14 of text*)

- We discuss & define what is meant by a package and what its uses are. Then, we define what is meant by a subsystem, a special type of package. Finally, we recapitulate the different kinds of models provided in UML.

- Packages overview:

- Why?

- Diagram (including hierarchy & "tree" option)

- Namespace control (including importing & accessing)

- Note: Text points out (p. 162) that some aspects of "packages" in UML are controversial (or at least not finally agreed). We will not go into too much detail for this reason.

- 14.1 Packages

- Reasons for:

- Convenience (e.g. to hide irrelevant detail)

- Allocate work among team members

- Specify and design a component

| *Probably use a*

| *"subsystem" package*

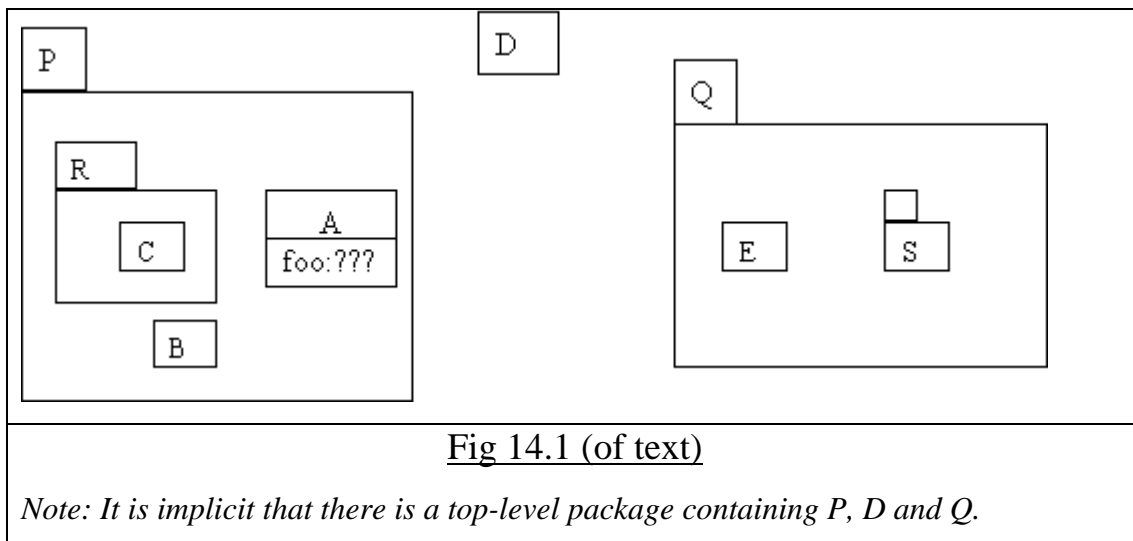
- Diagrams for packages:

Fig 14.1 (of text)

Note: It is implicit that there is a top-level package containing P, D and Q.

Points to note about package diagrams (as illustrated by Fig 14.1):

- A package is identified by "tab"
- May or may not (as desired) show what's inside a package
- Convenient convention is to write package name on its tab if its contents are shown, and within the "main" package rectangle otherwise
- Can use package symbols in all kinds of diagrams BUT an ordinary package (i.e. not a subsystem) cannot take part in an interaction as it is not a classifier

[For detail see end of p. 163 of text]

- One can also use a "tree" diagram for situations where packages contain sub-packages. For example, corresponding to the above diagram we have:

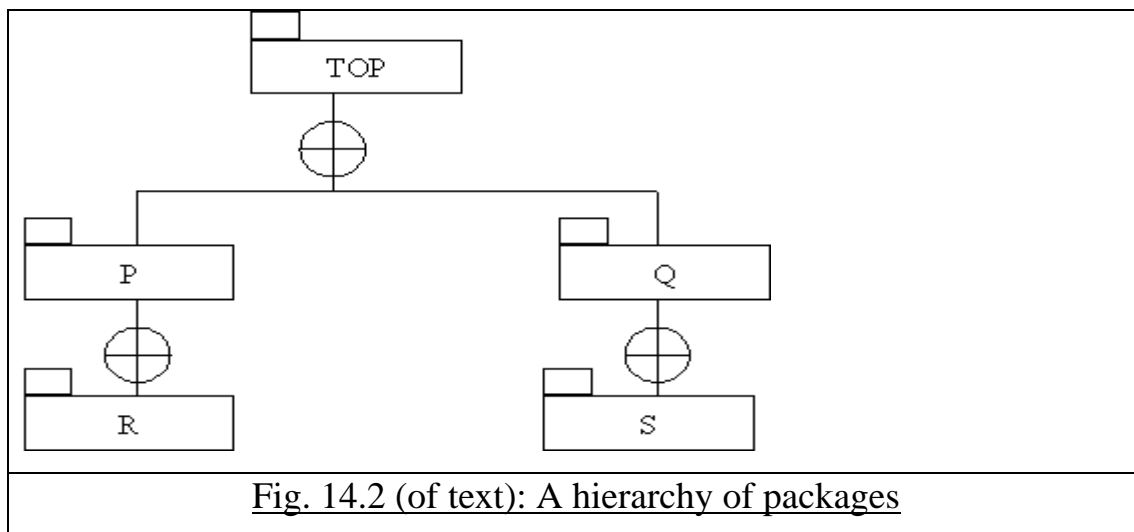


Fig. 14.2 (of text): A hierarchy of packages

• 14.1.1 Name Space Control

- The only thing an ordinary package (i.e. not a subsystem) can do is to define the namespace of the elements in it. (This is similar to concepts of scope and visibility in programming languages).

- Notation "P::A" (refer to Figure 14.1) is a precise name for the class A contained in package P.

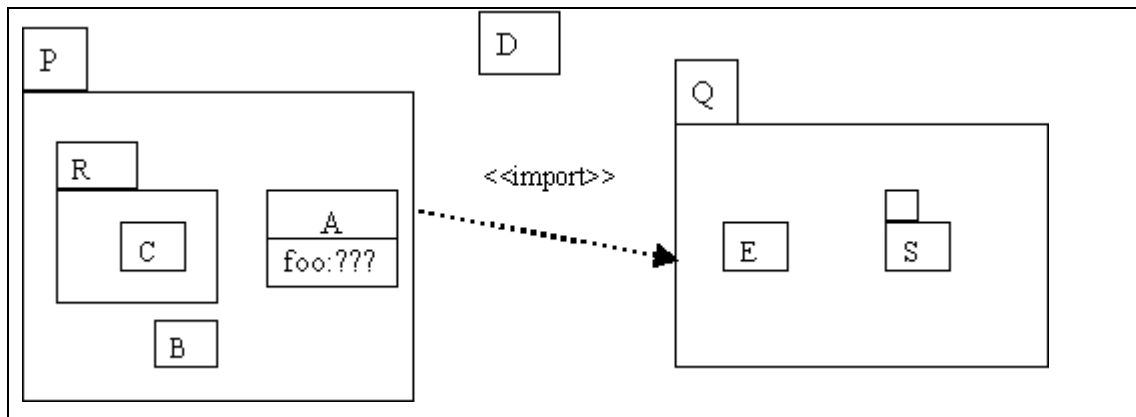
- Essentially, as for classes, you can have the same name for different things in different packages. No confusion results as full names are prefixed by the package name.

- "*Things outside a package cannot see in*"² - In Fig. 14.1, the class of attribute "foo" could be "B" or "D" but not "C" or "E" (as the latter two are not in namespace of "A").

² But things inside a package can "see out"

- <<import>> or <<access>>

- Both of these stereotypes allow some element inside "P" to name something in another package "Q" that does not contain "P". For example, with reference to Figure 14.1, if we draw



then the effect would be to make Q's elements visible to those of P. Hence, "foo" could now have (*i.e. be of*) class "E".

<<import>> - In the above case, where <<import>> is used, elements of "P" can refer to elements of "Q" just as if they were in "P" (e.g. "foo: E").

<<access>> - On the other hand, if we had used <<access>> rather than <<import>>, elements of "P" would have to include a prefix in references to elements of "Q" (e.g. "foo: Q::E").

Note: Packages can designate some elements as "public" and some as "private". We will not cover this aspect.

- 14.2 Subsystems

- A subsystem is a package that has a specification and a realization part.
- Subsystem diagram: May use stereotype <<subsystem>> or a special symbol on the "tab" (as below) to identify a package as a subsystem:

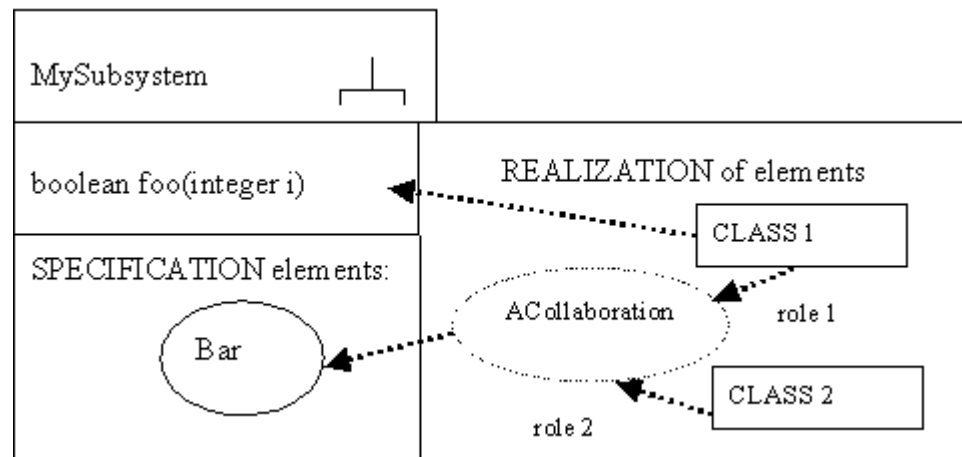


Fig. 14.3

Note: Here "dotted ellipse" is shorthand for a collaboration which is assumed to be defined in more detail elsewhere..

- Specification part:

- describes operations that can be done without revealing internal structure (could include Use Cases)
- can match interfaces (see earlier, Chapter 13)

- Realization part:

- May contains classes and other subsystems

- A subsystem may or may not be instantiable (depending on how it is defined)

• 14.3 Models (recap)

View	Models & Diagrams
Use Case view	Use Case model
Logical view	Class model (+ interaction and state diagrams as necessary)
Process view (threads of control)	Interaction & activity diagrams (as necessary) + Deployment diagrams
Development view	Component diagrams (+ (maybe) packages and subsystems)
Physical view	Deployment diagrams