Brief look back at some general design issues, especially Interfaces

Brief look back at some general design issues, especially Interfaces 1
Purpose 2
1.3 What are good systems like? [see §1.3 of text] 2
1.3.1 Encapsulation: Low coupling 2
- 3 pieces of information pertinent when considering coupling: 2
B. (Provided Interface) 3
A, C (Context dependency) (p9 of text) 5
A Java Interface Example 6
Background: 2.4 Interfaces and Abstract Classes 6
2.4.1 Implementing Interfaces in Java, with an example 7
Recap of benefits of modularity with well-defined interfaces 9
1.3.2 Abstraction: Good (high) cohesion 9
1.3.3 & 1.3.4 Architecture & components & "Component-based design: pluggability" (*outline*) . 10

Purpose

For completeness, and before finishing presentation of UML class models, we review briefly some points made in Chapter 1 of the text book (Stevens & Pooley), and perhaps in earlier lectures, particularly regarding component interfaces.

We also present a concrete example of a Java interface.

The aim is to contextualise some of the detailed aspects already discussed particularly regarding "design by contract".

1.3 What are good systems like? [see §1.3 of text]

- A key point is the need to ***localise*** related material in design and code.

- Modularity is concerned with splitting of a system into manageable "chunks", with *interfaces* between them.

- *Dependency*: Module A depends on module B if it is possible for some change to module B to necessitate a change in module A also. [*The text speaks of module A as being a "client" of module B*]

- We must look at what is a "good" module.

1.3.1 Encapsulation: Low coupling

- It is desirable to "***Minimise the number of cases in which a change in one module necessitates a change to another module***". If we achieve this we will have ***low coupling***.

- **3 pieces of information pertinent when considering coupling**:

A. Which modules are "clients" of, that is depend on, a given module? *This information identifies the "client" modules that may need to be changed or re-tested if the given module is changed.*
B. What changes inside a module may affect its "clients" (and thereby the rest of the system)? *If we know the assumptions "clients" of a given module may make about it we should be able to tell which changes to it may impact outside itself.*
C. What does the given module itself depend on?

*B. (**Provided Interface**)*
Defines some features of the module on which a "client" may rely.

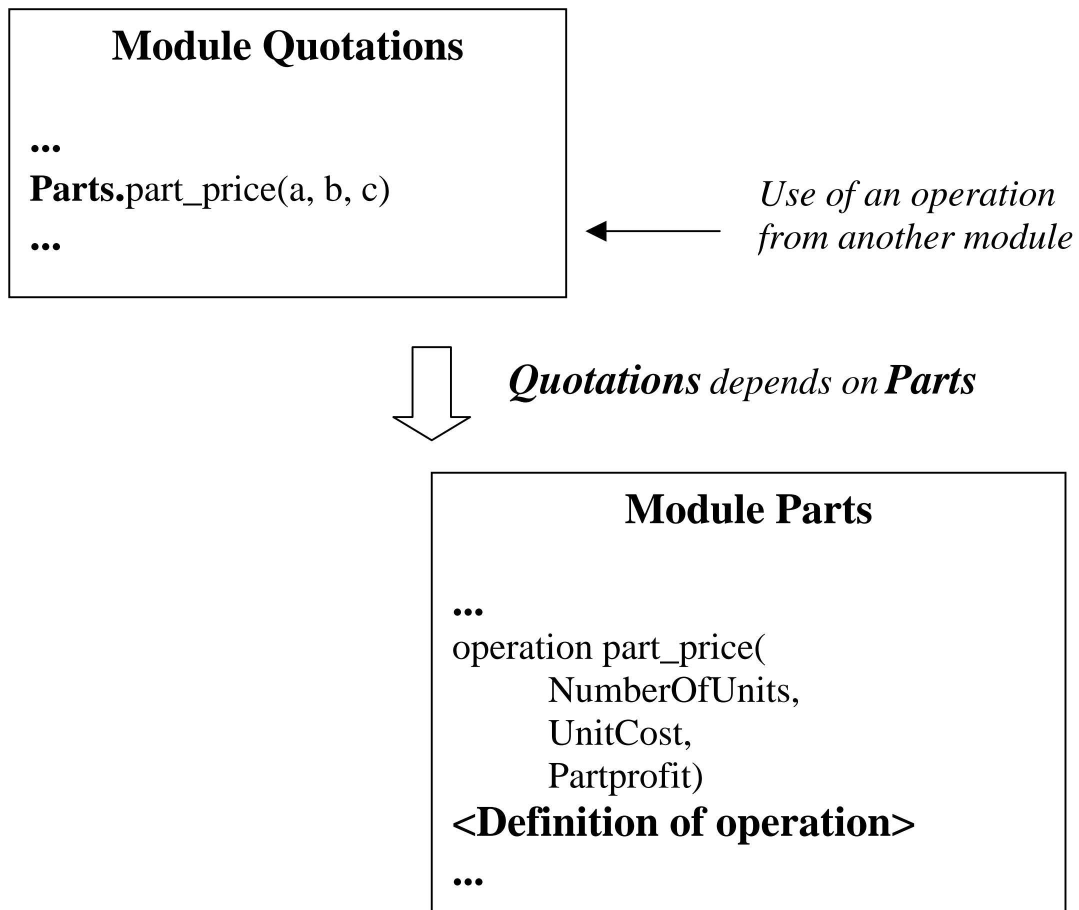
- Ideally, there should be automatic checks that no other module makes any assumption about this module that is not documented in its interface, and that the module always justifies the assumptions that are documented in the interface. However, in reality, there is variation between different programming languages and between design/analysis methods in how well they provide automatic support for such checks. [***This was mentioned in earlier notes***].

- Here is an outline of a possible *provided interface*:

• OPERATIONS (or methods or procedures ...)
- List of method names, with their arguments and argument types, that can be called by a "client".
• DATA (or attributes ...)
- List (if any) of data (directly) available to "clients"
• TYPES (or maybe classes ...)
- List (if any) of data types available to "clients"
• EXCEPTIONS
- List (if any) of exceptions "exported" to "clients"

- Stevens & Pooley (page 9) note there is still a lot of research on the notion of "interface".

- Following example illustrates some of the issues:



Text book (pages 8 and 9) discusses what information can be recorded in an interface and to what extent it can be automatically checked. For above example, many programming languages would make sure (probably at compile time) that the "client" (here **Quotations**) would use operation "part_price" with

(1) the right number of arguments and

(2) that the arguments were of the right type (e.g. integer, float, float or, perhaps, more precisely defined types).

These would be **syntactic checks**.

However, in some circumstances, it would be desirable to have **semantic checks** also. This would mean that "client" modules would be given (sufficient) information on how a module **behaves** rather than just on how to interact with it. For example, in (unit) testing a module it is often necessary to replace modules it depends on with **stubs**; A minimal stub for our "part_price" operation would be something that just returned a random "float" value (say). However, we might want more than that (and yet not the complete functionality) - for example, that the value returned be plausible in some sense.

A, C (Context dependency) (p9 of text)

When considering "coupling" in a system we need not only to look at **B** (***provided interfaces***) but also at **A** and **C**:

A. For a given module, which modules are actually its clients, that is, actually use its provided interface.

C. We also need to know what our given module itself depends on, that is, is a "client" of. This is the module's ***required interface***, a definition of what the module needs in order to work.

Note: Tools are available to help establish dependencies mentioned (*including UML, but there are verification tools that measure in some sense the degree of coupling of a system*). Also, some design methods (e.g. UML) support definition of both provided and required interfaces.

A Java Interface Example

- We are mainly concerned with design rather than programming. Still, it is interesting to look at an example of how provided interfaces may be implemented in Java, say. Example is taken from "*Data structures and algorithms in JAVA*" by Goodrich and Tamassia.

Background: 2.4 Interfaces and Abstract Classes

Following is an extract from Goodrich and Tamassia.

> In order for two objects to interact, they must "know" about the various messages that each will accept, that is, the methods each object supports. To enforce this "knowledge", the object-oriented paradigm asks that classes specify the ***application programming interface*** (API), or simply ***interface***, that their objects present to other objects. In the ***ADT-based*** approach (see Section 2.1.2) to data structures followed in this book, an interface defining an ADT is specified as a type definition and a collection of methods for this type, with the arguments for each method being of specified types. This specification is, in turn, enforced by the compiler or run-time system, which requires that the types of parameters that are actually passed to methods rigidly conform with the type specified in the interface. This requirement is known as ***strong typing***. Having to define interfaces and then having those definitions enforced by strong typing admittedly places a burden on the programmer, but this burden is offset by the rewards it provides, for it enforces the encapsulation principle and often catches programming errors that would otherwise go unnoticed.

ADT = Abstract Data Type

My underlining

Good Syntactic Checking

2.4.1 Implementing Interfaces in Java, with an example

Getting into Java specifics

Have gone on to some topics reviewed in Chapter 2 of our own text

The main structural element in Java that enforces an API is the ***interface***. An interface is a collection of method declarations with no data and no bodies. That is, the methods of an interface are always empty. When a class implements an interface, it must implement all the methods declared in the interface. In this way, interfaces enforce a kind of inheritance called ***specification***, where we require that each method inherited be specified in full.

Suppose, for example, that we want create an inventory of antiques we own, categorized as objects of various types and with various properties. We might, for instance, wish to identify some of our objects as sellable, in which case they could implement the `Sellable` interface shown in Code Fragment 2.7.

```
/** Interface for objects that can be sold. */
public interface Sellable{
  /** description of the object */
  public String description();

/** list price in cents */
  public int listPrice();

/** lowest price in cents we will accept */
  public int lowestPrice();}
```

Code Fragment 2.7: Interface `Sellable`

We can then define a concrete class, `Photograph`, shown in Code fragment 2.8, that implements the `Sellable` interface, indicating that we are willing to sell any of our `Photograph` objects: This class defines an object that implements each of the methods of the `Sellable` interface, as required. In addition, it adds a method, `isColor`, which is specialized for `Photograph` objects.

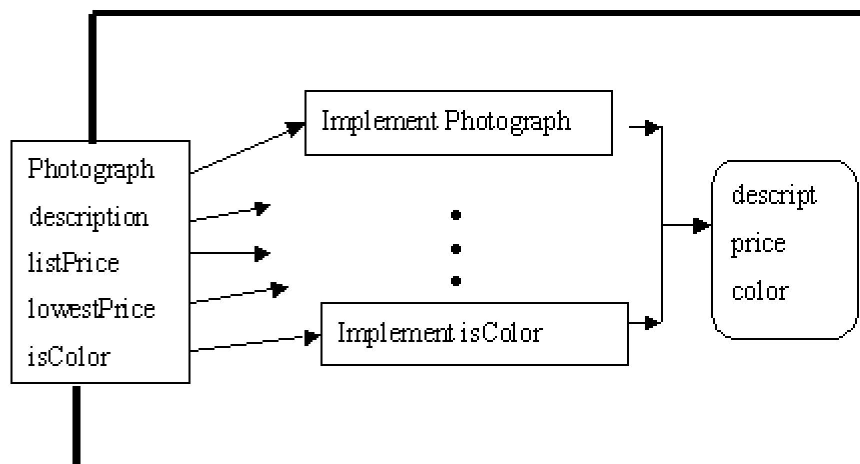
```
/** Class for photographs that can be sold. */
public class Photograph implements Sellable{
  private String descript;    // description of this photo
  private int price;          // the price we are setting
  private boolean color;      // true if photo is in colour

public Photograph(String desc, int p, boolean c){//constructor
  descript = desc;
  price = p;
  color = c;
}

  public String description(){ return descript;}
  public int listPrice(){ return price;}
  public int lowestPrice(){ return price/2;}
  public boolean isColor(){ return color;}
}
```

Attributes

Special method which when called "makes" a photograph object

Code Fragment 2.8: Class `Photograph` implementing the `Sellable` interface

A simple diagram (not UML) of the above class, showing public interface and private implementation, could be



Recap of benefits of modulari**ty with well-defined interfaces**

- Makes multi-person teams more productive
- Allows developers to focus on a particular part of the system, knowing the extent to which other parts can be ignored
- Should make "bugs" easier to find
- Helps in testing (especially integration and re-testing)
- Makes re-use more feasible

- Need for interfaces to be clearly defined is an essential starting point. However, a more interesting question perhaps is how to set about choosing modules with the "right" content and then ensuring that they have the "right things" in their interfaces. This brings us on to *cohesion* and is, of course, something we already looked at in designing classes.

1.3.2 Abstraction: Good (high) cohesion

- "Cohesive" means we want a module to represent some intuitively related things rather than a collection of unconnected material.

For example, a module that

Provides a user interface/Does financial accounts/Calculates volumes and areas

is not cohesive! Moreover, such a module is bad for coupling also as one is likely to get more disparate modules depending on it.

On the other hand, a module whose sole responsibility is calculation of areas and perimeters of plane figures would be more cohesive [See CRC cards earlier].

- Given that a module has "cohesive responsibilities", the next thing is to provide it with an interface that "abstracts away from" implementation details. These are details that a "client" module neither needs to nor should know about.

Example (text): Giving a point in either rectangular or polar coordinates to a "client". As long as "client" gets the right result it shouldn't know about the calculation details.

- Terms "**abstraction**", "**information hiding**" and "**encapsulation**" are all used, sometimes with slightly different meanings. Text uses:

Abstraction: when a "client" does not need to know more than is in the interface

Encapsulation: when a "client" is not able to know more than is in the interface.

1.3.3 & 1.3.4 Architecture & components & “Component-based design: pluggability” (*outline*)

If a module has high cohesion and low coupling (*at least in terms of its required interface*) it should be a good candidate for **re-use** and/or **replacement** (*by a more advanced version of itself, say*). Ideally, it should be a "pluggable component", for re-use in similar software products or for replacement in later versions of current software.

How practical or feasible this use as a component is will depend also on the architecture of which the module is a part. Briefly, and roughly,
-- Often, in the past, the architecture of a piece of software was considered in isolation from other pieces of software.
-- But, it is more fruitful in many cases to consider a common architecture for a range of products or, at the extreme, throughout a software development organisation.
A key benefit, one hopes, of such an architectural approach would be greater use of modules as pluggable components.
-- We speak of “**frameworks**” and will present an example shortly [Chapter 16 of text].

Notes:
1) We will touch on “pluggable components” again when presenting **UML support** for documenting such. However, be aware that the whole area of component-based software engineering is very wide.
2) Another aspect of reuse in SW development is in identifying and making use of **design patterns**. We will touch on this briefly later.