DESIGN PHASE, III: Information Hiding Aspects

This part of the lecture is based on chapter 8 of Schach's "Practical Software Engineering". Its objectives are listed below:

- Appreciate the importance of information hiding
- Learn how to design a product so that aspects that are likely to change are isolated in a single function or in a single compilation unit consisting of related functions
- Discover how to endure that details as to how data structure is implemented within a compilation unit are not known outside of that compilation unit.

Information Hiding Principle

The term **information hiding** is due to D.L. Parnas:

> **If one part of the product is likely to be changed, then everything related to that part should be isolated in one function or compilation unit.**

Information hiding is supported best as a method and as a language "feature" in the OO paradigm:

> **OO paradigm = abstract data types + encapsulation + inheritance**

However, principles of information hiding are applicable outside OO world, which is our main focus of interest in this lecture.

Q: What are the advantages of language support concerning information hiding?

Information Hiding: Motivation

Indeed, we appreciate the IH principle and OO paradigm, so the question is not whether we should apply these principles or not, but:

> **what is a good design in the sense of information hiding?**

A lot of things can go wrong:

- Information hiding concerns design of module interfaces. Changes in interfaces lead to changes in related modules: *global rather than local changes.*

- Maldesigned modules may lead to redundant code: different functions offering same or similar features. This, in turn, *complicates the maintenance.*

C-CC case study

Suggested modularization is as follows:

- Module *shop.c* defines internal (*static*) data type *struct shop*

- Module *region.c* defines internal (*static*) data type *struct region*

Hence, details are hidden inside modules *shop.c* and *region.c*.

Problem: how to implement mutual dependencies: features of region module must have information about shops and vice versa?

Modules offer access to the information they maintain via **functions**. To preserve information hiding, function interfaces (their arguments and return parameters) are different from internal data type definitions.

Exception handling and information hiding

Inter-module information exchange has to take care of exception handling, too.

Here the question is: what kind of communication is reasonable between calling module and called module, when an exceptional situation is encountered in the calling module?

Example. Function *main()* calls *input_new_shop_record()* (251) and the latter calls *write_shop_record()*, where an exceptional situation occurs (e.g. disc full).

- *write_shop_record()* terminates the program with *exit()*.
- *write_shop_record()* passes the exception to *input_new_shop_record()*, and the latter decides, what to do.

Q: What are the differences in module interfaces?
How to preserve information hiding?

C-CC case study: reports

We have decided early in the development phase that REPORT does not lead to an object. Refer to 224-227 for reasons for that.

Assume that in the maintenance phase, the following new feature should be implemented:

> REPORTS should be collected and PERFORMANCE DIAGRAMS should be computed.

Assume, we would like to add a new object REPORT now.

Q: How to modify the existing design in order to allow for the new object REPORT?

Recall the design so far (fig. 6.8, transparency 227):

Object	Data Structure	Operations
REGION	region_record	*read region_record* *write region_record* *delete region_record* *generate region_report* *generate vice-president_report*
SHOP	shop_record	*read shop_record* *write shop_record* *delete shop_record* *generate shop_report*

Figure 6.8 Object-operations table: third refinement.

Describe the following components of the new module reports.c:

- Internal data
- Functions implementing new features: report collection and performance diagrams
- Changes on existing modules *shop.c* and *region.c*