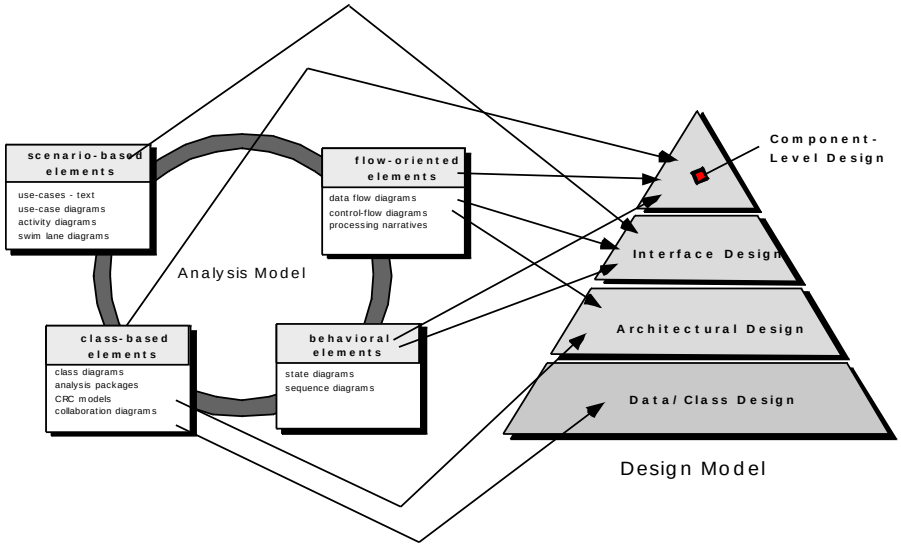
SOFTWARE ENGINEERING

Week 10
Design Engineering - I

Agenda

1. Layers of Software Design
   1. Data design
   2. Architectural design
   3. Interface design
   4. Procedural design (modules)
2. Design Principles
   1. Cohesion
   2. Coupling

Analysis Model ⟶ Design Model



Layers of Software Design (1)

- Software design is the process of applying various techniques and principles to define a system in sufficient detail to permit its physical implementation (coding).
- Design comes after analysis and it is basically based on analysis models.

- **Data design**
- **Architectural design**
- **Interface design**
- **Procedural design (modules)**

Layers of Software Design (2)

- Data Design
  - Transforms the information domain model created during the analysis into the data structures, file structures, database structures that will be required to implement software.
  - Data objects and relationships in ERD and the detailed data content depicted in the data dictionary provide the basis.

- Architectural Design
  - Defines the relationship among major structural elements of the program.
  - The modular framework of a program can be derived from the analysis model and interaction of subsystems depicted in DFD diagrams.

Layers of Software Design (3)

- Interface Design
  - Describes how software communicates
  - Interface implies a flow of information, therefore DFD/CFD diagrams provide the basis.
  - There are two types of Interface:
    - Application Programming Interface (API)
    - Graphical User Interface (GUI)

- Procedural Design (Modules)
  - Transforms structural elements of the program architecture into a procedural description of software components.
  - Information obtained from PDL (Algorithms / Flowcharts) serve as basis.

Design Principles

- Software modules should be in a hierarchical organization.
- Software should be modular, that is, the software should be logically partitioned into elements that perform specific functions.
- Should contain both data abstraction and procedural abstraction.
- Should lead to interfaces that reduce the complexity of connections between modules (low coupling).
- Must be an understandable guide for coders, testers and maintainers.
- Should exhibit uniformity and integration.

Design Concepts (1)

- **Component:**
  - Any piece of software or hardware that has a clear role.
  - A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
  - Many components are designed to be reusable.
  - Conversely, others perform special-purpose functions.

- **Module:**
  - A component that is defined at the programming language level.
  - For example, functions are modules in C.
  - For example, methods, classes and packages are modules in Java.

Design Concepts (2)

- **Modularity:**
  - A complex system may be divided into simpler pieces called *modules.*
  - A system that is composed of modules is called *modular.*
  - When dealing with a module we can ignore details of other modules.
  - Use divide and conquer method for modularity.
  - For two modules m1 and m2, the effort relation is as follows:

    E(m1) + E(m2) < E(m1+m2)

- **Cohesion:** The degree of dependencies within a module.
- **Coupling:** The degree of dependencies between two modules.

Design Concepts (3)

- **Abstraction:**
  - Abstraction is a means of achieving stepwise refinement by suppressing unnecessary details.
  - It is to conceptualize problem at a higher level.
  - Abstractions allow you to understand and concentrate on the essence of a subsystem without having to know unnecessary details.
  - The designer should keep the level of abstraction as high as possible.
- **Data abstraction (Abstract Data Type):**
  - A data type together with the operations performed on instantiations of that data type.
- **Procedural abstraction:**
  - The designer defines a procedure at a higher level.

Example: C definition without Data Abstraction


```
struct personel
{
  long int TCNum;
  char Ad[20], Soyad[20];
  int DTarihi_gun, DTarihi_ay, DTarihi_yil;
  int IGTarihi_gun, IGTarihi_ay, IGTarihi_yil;
} ;
```

11

Example: C definition with Data Abstraction


```
typedef struct
{
  int gun, ay, yil;
} tarih;

struct personel
{
  long int TCNum;
  char Ad[20], Soyad[20];
  tarih DogumTarihi;     //abstraction
  tarih IseGirisTarihi; //abstraction
} ;
```

12

Example:C program without Procedural Abstraction


```
#include <stdio.h>
#include <stdlib.h>
#define N 5
int main()
{
  int a[N] = {10,20,30,40,50};
  int b[N] = {15,25,35,45,55};
  int i;
  for (i=0; i < N; i++)  printf("%d \t", a[i]);
  printf("\n");
  for (i=0; i < N; i++)  printf("%d \t", b[i]);
  return 0;
}
```

13

Example:C program with Procedural Abstraction


```
#include <stdio.h>
#include <stdlib.h>
#define N 5

void yaz(int dizi[], int M) {
  int i;
  for (i=0; i < M; i++)
    printf("%d \t", dizi[i]);
  printf("\n");
}
int main() {
  int a[N] = {10,20,30,40,50};
  int b[N] = {15,25,35,45,55};
  yaz(a, N); //abstraction
  yaz(b, N); //abstraction
  return 0;
}
```

14

Design Strategies

- **Stepwise refinement:**
  - It is a top-down design strategy.
  - A higher level abstraction is refined to a lower level abstraction with more details in each step.
  - Several steps are taken.

- **Top-down design:**
  - First design the very high level structure of the system.
  - Then gradually work down to detailed decisions about low-level constructs.
  - Finally arrive at detailed individual algorithms that will be used.

- **Divide and conquer:**
  - Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things .
  - Separate people can work on each part.
  - Each individual component is smaller, and therefore easier to understand.

15

Cohesion

- **Cohesion** is a measure of dependencies ***within*** a module.
- If a module contains many closely related functions its cohesion is high.
- The designer should aim **high cohesion.**
  - Module is understandable as a meaningful unit
  - Functions of a module are closely related to one another
  - This makes the system as a whole easier to understand and change

1. Functional cohesion	(best)
2. Informational cohesion	(desirable)
3. Communicational cohesion	
4. Procedural cohesion	
5. Temporal cohesion	
6. Logical cohesion	(should be avoided)
7. Coincidental cohesion	(worst)

16

1.Functional Cohesion

- A module with functional cohesion performs exactly one action.
- This is achieved when *all the code that computes a particular result* is kept together - and everything else is kept out.

- Example:

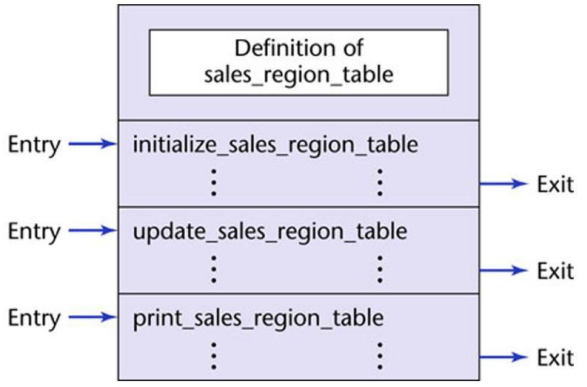
  calculate_sales_commission;

- Advantages:
  - Easier to understand
  - More reusable
  - Corrective maintenance is easier due to fault isolation
  - Easier to extend

17

2.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.
- Example:



18

3.Communicational Cohesion

- A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data.
- All the *actions that access or manipulate certain data* are kept together (e.g. in the same class) - and everything else is kept out.
- Example:

  update_record_in_database_AND_write_it_to_audit_trail;

  calculate_new_coordinates_and_send_them_to_terminal;

19

4.Procedural Cohesion

- A module has procedural cohesion if it performs a series of actions related by the procedure to be followed in the product.
- Procedures that are used one after another are kept together.
  - Even if one does not necessarily provide input to the next.
- Example:

  read_part_number_AND_update_repair_record_on_master_file;
- Disadvantages
  - Actions still weakly connected, so not reusable.

20

5.Temporal Cohesion

- A module has temporal cohesion when it performs a series of actions related in time.
- Operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out.
- Example:

```
initialize_sales_district_table,
read_first_transaction_record,
read_first_old_master_record;
```

(a.k.a. perform_initialization)

21

6.Logical Cohesion

- A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module.
- Example:

```
int  op_code = 7;
do_operation (op_code,
                dummy_1,
                dummy_2,
                dummy_3);
```

- Disadvantage: The interface is difficult to understand.
- (It is not clear which of the dummy variables will be used when op_code is equal to 7.)

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

22

7.Coincidental Cohesion

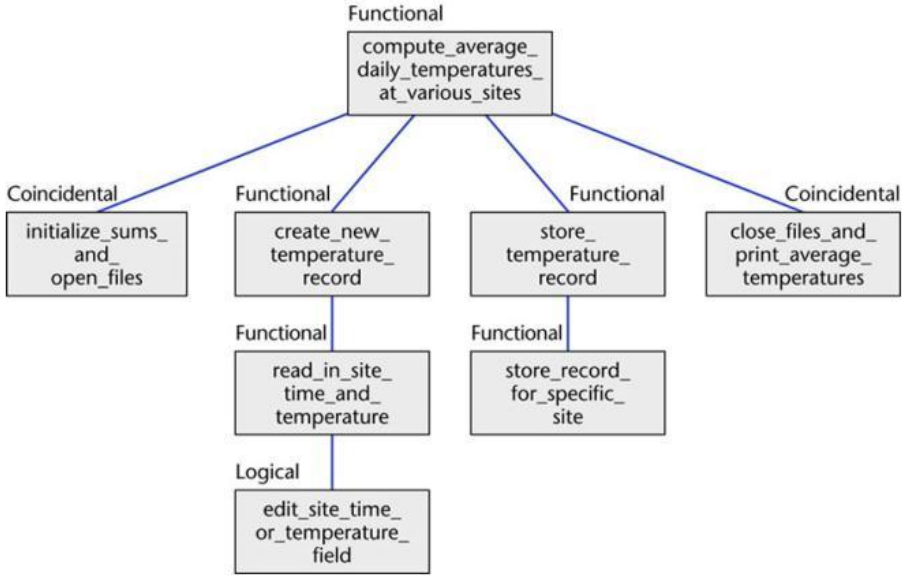
- A module has coincidental cohesion if it performs multiple, completely unrelated actions.
- Example:

  print_next_line,
  reverse_string_of_second_parameter,
  add_7_to_fifth_parameter,
  convert_fourth_parameter_to_ floating_point;

- Disadvantage:
  - Module not reusable.
    - Solution: Break the module into separate modules, each performing one task.
  - Degrades maintainability
  - No reuse

23

Cohesion Examples



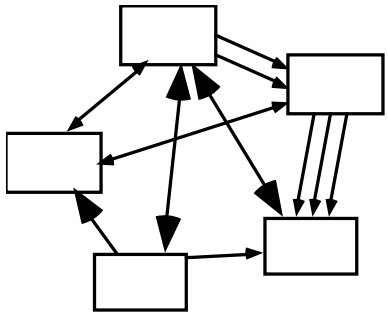
24

Coupling

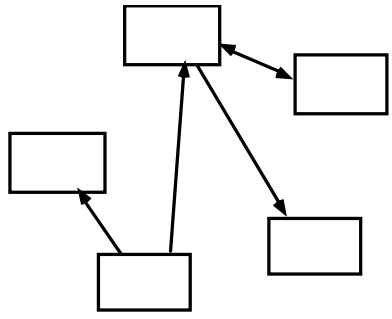
- **Coupling** is a measure of the dependencies ***between*** two modules.
- If two modules are strongly coupled, it is hard to modify one without modifying the other.
- The designer should aim **low coupling.**
  - Modules have low interactions with others
  - Understandable separately

1. Data coupling
2. Stamp coupling
3. Control coupling
4. Common coupling
5. Content coupling

Coupling



High coupling



Low coupling

1.Data Coupling

- Two modules are data coupled if all parameters are the same data types (simple parameters or data structures)
- The more arguments a module has, the higher the coupling
  - All modules that use the called module must pass all the arguments
  - Coupling should be reduced by not defining modules with unnecessary arguments
- Example:

```
compute_product (first_num, second_num);
```


2.Stamp Coupling

- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure.
- Examples:

```
calculate_withholding (employee_record);
print_inventory_record (warehouse_record);
```

- Disadvantages:
  - It is not clear, without reading the entire module, which fields of a record are accessed or changed.
  - More data than necessary is passed

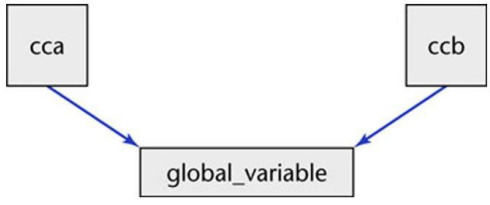
3.Control Coupling

- Two modules are control coupled if one passes an element of control to the other.
- Occurs when one procedure calls another using *a* ***'flag'*** *or* ***'command'*** that explicitly controls what the second procedure does.
- Example:
  - An operation code is passed to a module with logical cohesion
- Disadvantages:
  - The modules are not independent
  - To make a change you have to change both the calling and called modules

29

4.Common Coupling

- Two modules are common coupled if they have write access to **global data.**
- All the modules using the global variable become coupled to each other.
- Example:
  - Modules cca and ccb can access *and change* the value of global_variable
- Disadvantages:
  - A change during maintenance to the declaration of a global variable necessitates corresponding changes in all modules
  - Common-coupled modules are difficult to reuse



30

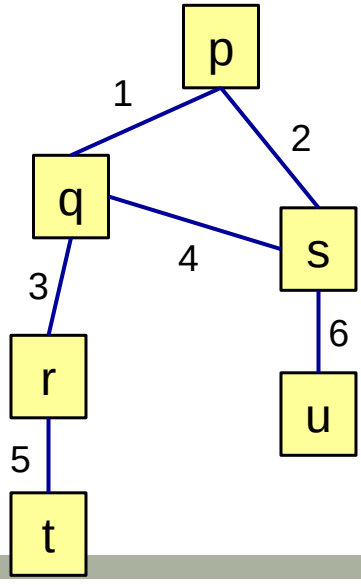
5.Content Coupling

- Two modules are content coupled if one directly references contents of the other.
- Occurs when one module modifies data that is ***internal*** to another module.
- Example:
  - Module p refers to a local data of module q and numerically changes it.
- Disadvantage: Almost any change to module q, requires a change to module p.
- To reduce content coupling you should therefore *encapsulate* all instance variables (declare variables as `private` in Java)
- Avoid call-by-reference function calling in C.

31

Coupling Examples

• Modules p, t, u access the same database in update mode.



32

Interface descriptions

Interface Number	Inputs	Outputs
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 between modules

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

User Interface Design

Aspects of usability

- Usability: The system should allow the user to learn and to use the basic capabilities easily.
- Usability can be divided into separate aspects:
  - *Learnability*
    - The speed with which a new user can become proficient with the system.
  - *Efficiency of use*
    - How fast an expert user can do their work.
  - *Error handling*
    - The extent to which it prevents the user from making errors, detects errors, and helps to correct errors.
  - *Acceptability*
    - The extent to which users like the system.

Terminology of Graphical User Interface (GUI)

- **Dialog**: A specific window with which a user can interact, but which is not the main UI window.
- **Control** or **Widget:** Specific components of a user interface.
- **Affordance**: The set of operations that the user can do at any given point in time.
- **State**: At any stage in the dialog, the system is displaying certain information in certain widgets, and has a certain affordance.
- **Mode**: A situation in which the UI restricts what the user can do.
- **Modal dialog**: A dialog in which the system is in a very restrictive mode.
- **Feedback**: The *response from the system* whenever the user does something, is called feedback.
- **Encoding techniques**. Ways of encoding information so as to communicate it to the user.

37

User Interface Design Principles

Principle	Description
User familiarity	Use terms and concepts which are drawn from the experienced users.
Consistency	Be consistent in that, similar operations should be activated in the same way.
Recoverability	Include mechanisms to allow users to recover from errors.
User guidance	Provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	Provide appropriate interaction facilities for different types of users (such as clerk or manager).

38

Usability Principles (1)

1. Base the User Interface designs on users' *tasks*.
   - Perform use case analysis to structure the UI.

2. Ensure that the sequences of actions to achieve a task are as *simple* as possible.
   - Reduce the amount of manipulation the user has to do.
   - Ensure the user does not have to navigate anywhere to do subsequent steps of a task.

3. Ensure that the user always knows what he should do next.
   - Ensure that the user can see *what commands are available*.
   - Make the *most important commands stand out*.

Usability Principles (2)

4. Provide good feedback including effective error messages.
   - Inform users of the *progress* of operations and of their *location* as they navigate.
   - When something goes wrong, explain the situation in adequate detail and *help the user to resolve the problem*.

5. Ensure that the user can always get out, go back or undo an action.
   - Ensure that all operations can be *undone*.
   - Ensure it is easy to *navigate back* to where the user came from.

6. Ensure that response time is adequate.
   - Keep response time less than a second for most operations.
   - Warn users of longer delays and inform them of progress.

Usability Principles (3)

7. Use *understandable encoding techniques*.
   - Choose encoding techniques with care.
   - Use labels to ensure all encoding techniques are fully understood by users.

8. Ensure that the UI's appearance is *uncluttered*.
   - Avoid displaying too much information.
   - Organize the information effectively.
   - Use consistent language and meaningful keywords
   - Avoid abbreviations
   - Make text readable, use both upper and lower case
   - Use colors and graphics effectively

Usability Principles (4)

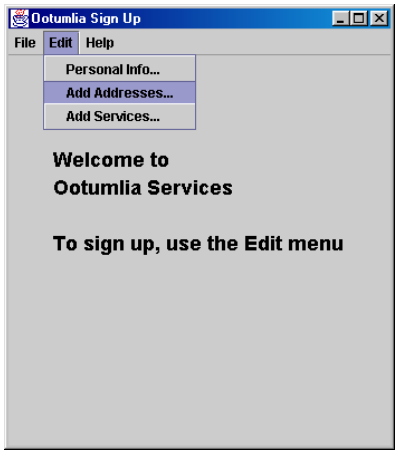
9. Robustness.
   - Minimize keystroke and mouse travel distance
   - Provide defaults for missing data (e.g. current date)
   - Automatically correct the obvious errors

10. Provide all necessary *help*.
    - Integrate help with the application.
    - Ensure that the help is accurate.

11. Be *consistent and uniform*.
    - Use similar layouts and graphic designs throughout your application.
    - Follow look-and-feel standards.

Example: Welcome screen

Wrong

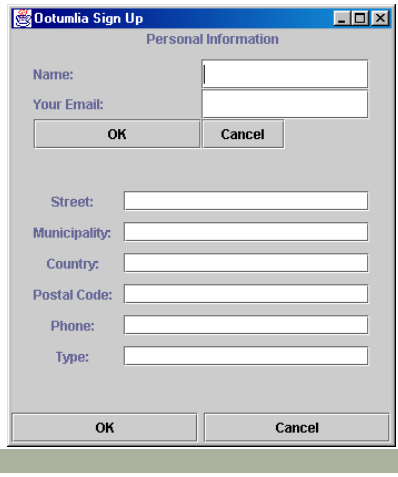


Correct

43

Example: Personal information screen

Wrong

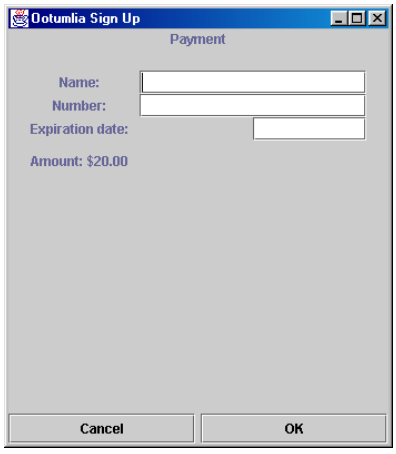


Correct

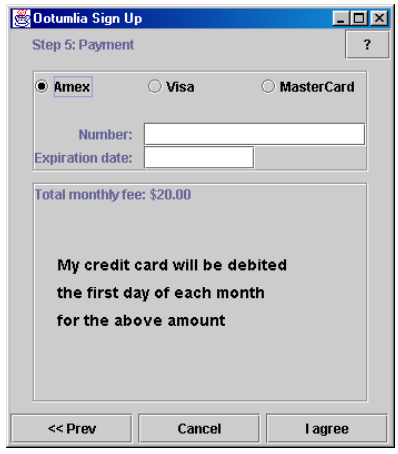
44

Example: Payment screen

Wrong



Correct

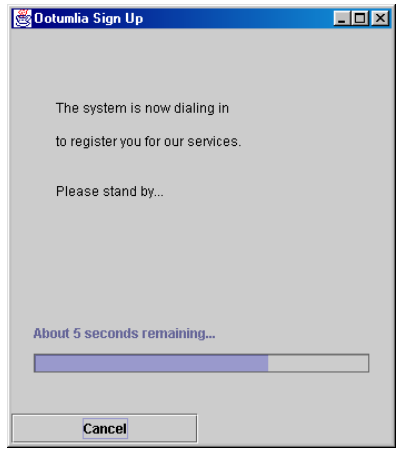


45

Example: Sign up screen

Wrong

Correct



46

User Interface Patterns

- Many standard patterns are available for
  - Page layout
  - Page elements
  - Forms and input
  - Tables
  - Direct data manipulation
  - Navigation
  - Searching

Example: Web page layout