Chapter 9. Software Design

9.1 Introduction

- Design is “the process of applying various techniques and principles for the purpose of defining a device, a process, or a system in sufficient detail to permit its physical realization.” Design is also the most artistic or creative part of the software development process.
- The design process converts the “what” of the requirements to the “how” of the design. The results of the design phase should be a document that has sufficient detail to allow the system to be implemented without further interaction with either the specifier or the user.
- The design process also converts the terminology from the problem space of the requirements to the solution space of the implementation.

9.1 Introduction

- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are invariably inter-leaved.
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
  - Implementation is the process of realizing the design as a program.

An object-oriented design process

- Structured object-oriented design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design models are an important communication mechanism.

- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.

9.2 Phases of the Design Process

- The following are phases in design:

  Data design—This phase produces the data structures.

  Architectural design—This phase produces the structural units (classes).

  Interface design—This phase specifies the interfaces between the units.

  Procedural design—This phase specifies the algorithms of each method.

System context and interactions

- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

- A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.

Object class identification

- Identifying object classes is often a difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

- Approaches to identification
  - Use a grammatical approach based on a natural language description of the system.
  - Base the identification on tangible things in the application domain.
  - Use a behavioural approach and identify objects based on what participates in what behaviour.
  - Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

Design models

- Design models show the objects and object classes and relationships between these entities.
- Static models describe the static structure of the system in terms of object classes and relationships.
- Dynamic models describe the dynamic interactions between objects.

9.2 Phases of the Design Process

EXAMPLE 9.3

Design the library classes/data structures from the data items in the object model shown in Fig. 9-1 for the library problem (see Examples 8.1 and 2.6). The data design and the architectural phases have been combined in this example. The concentration in this example is on the loan and checkout functionality, with little regard for the other necessary tasks, such as administration, cataloging, assigning overdue fines, retiring books, and patron maintenance. The domain entity ‘‘book’’ is probably not going to continue into the design. It will be combined with ‘‘copy’’ into a class/data structure that stores all the information about a copy. It will probably use the ISBN and a copy number as the unique identifier. The patron information will be stored in a second data structure. Each record is probably identified by an unique patron ID number. The loan information may or may not be a separate data structure. If borrowing information needs to be saved beyond the return of the book, then it had better be a separate class/data structure. Otherwise, the patron ID can be part of the copy class/data structure along with the due date of the book. Note in Fig. 9-2 that many data items have been added that are more in the implementation/solution space than in the problem/domain space. It can be argued that ‘‘ISBN’’ is part of the problem space instead of the solution space, but many library systems do not allow normal users to search by ISBN.

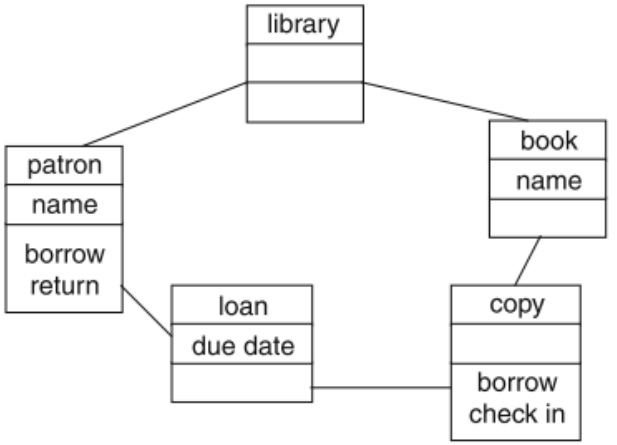


Fig. 9-1. Object model for library.

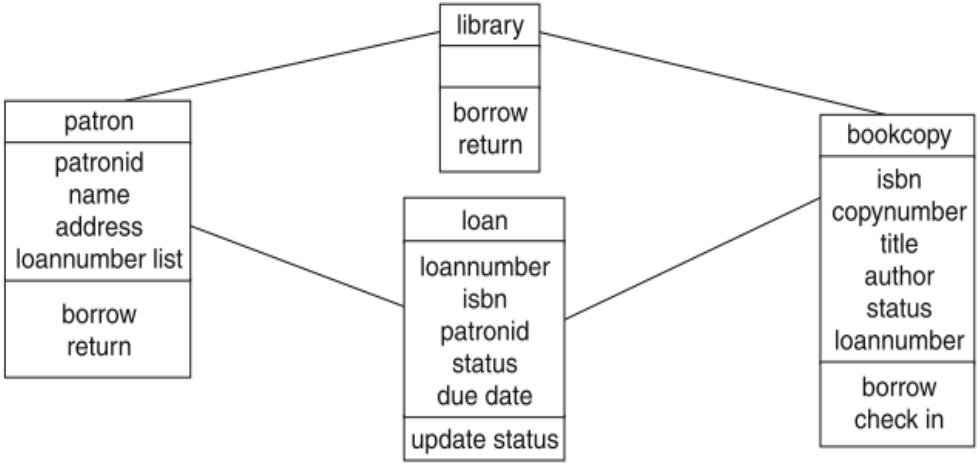


Fig. 9-2. Class diagram for library problem.

9.2 Phases of the Design Process

9.2.1 INTERFACES

- An interface specification is a specification of the external behavior of a module. It should be complete enough so that the calling module knows exactly what the called module will do under any circumstance. It should also be complete enough so that the implementer knows exactly what information must be provided.
- The interface specifications in an OO model are often the signatures of the public methods and the semantics associated with the methods. Interfaces can also be specified as part of a formal specification of the behavior of the whole system.
- Interfaces can also be the invariants, preconditions, and post-conditions for a method
- Objects may have several interfaces which are viewpoints on the methods provided.
- The UML uses class diagrams for interface specification but Java may also be used.

9.2 Phases of the Design Process

EXAMPLE 9.4

Design the interfaces for the borrow functions of the library problem using the class diagram produced in Example 9.3.

Both patron and bookcopy have ‘‘borrow’’ methods. Presumably, calling one or the other of these two methods creates the instance of loan. It is not clear from the class diagram which method creates the instance. However, it might be clear if the parameters and return type of each of these methods are specified.

```
Method patron::borrow
     Input parameters – isbn
     return type – int
          0 if book is not available
          1 if book is available and loan instance created successfully
          -1 if error condition
Method bookcopy::borrow
     input parameter – loannumber
     return type – int
          0 if book copy is not available
          1 if book copy updated successfully
```

9.3 Design Concepts

- Two approaches to design are known as refinement and modularity:

Refinement—This design approach develops the design by successively refining levels of detail. Sometimes this is called ‘‘top-down’’ design.

Modularity—This is a structuring approach that divides the software into smaller pieces. All the pieces can be integrated to achieve the problem requirements.

EXAMPLE 9.5

Refine the borrow book function from the library problem.

The top level starts with a function `borrow book` with two parameters, the title of the book and the name of the patron.

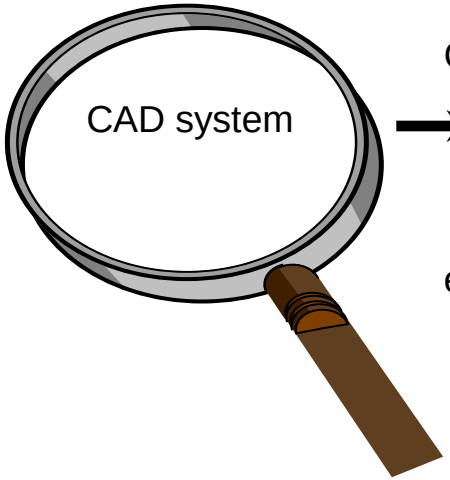
The next refinement adds the notion of the `loan` entity. It probably has the following parts: find book given book title, find the patron given patron name, and create loan instance given IDS of book and patron.

The next refinement expands each part. `Find book` returns ISBN if book is found and available, returns zero if book is not found, and returns –1 if book is in use. `Find patron` returns patron ID if patron is found and is in good standing, returns zero if patron not found, and returns –1 if patron is not eligible to borrow books. `Createloan` returns 1 if created successfully.

Stepwise Refinement

- A way of developing a computer program by first describing general functions, then breaking each function down into details which are refined in successive steps until the whole program is fully defined. Also called top-down design
- Starting from the requirements, at each step one constructs a more concrete description of the system and verifies it against the specification constructed in the previous step until one arrives at the implementation.

Abstraction I



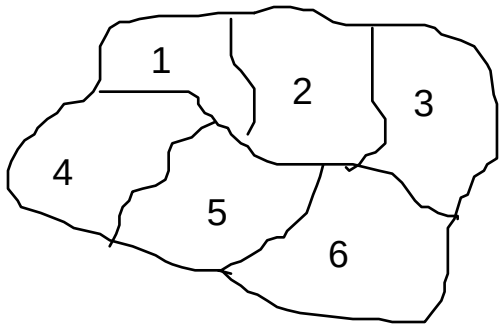
Abstraction II

```
CAD software tasks:
    user interaction task;
    2-D drawing creation task;
    graphics display task;
    drawing file management task;
end.
```

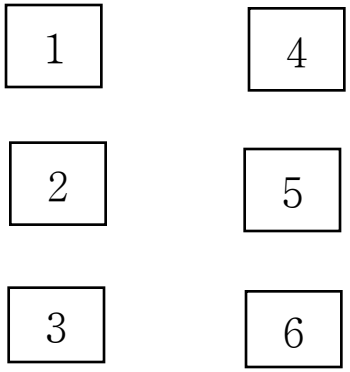
Abstraction III

```
procedure: 2-D drawing creation;
    repeat util <drawing creation task terminates>
        do while <digitizer interaction occurs>
            digitizer interface task;
            determine drawing request;
                line: line drawing task;
                circle: circle drawing task;
                .
                .
                .
```

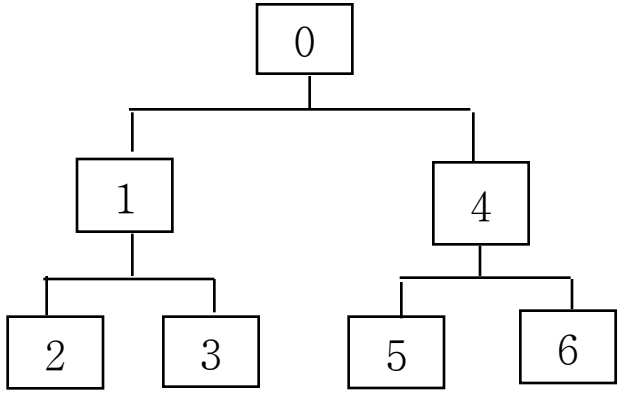
Modularization



Problem Domain



System Decomposition



System Structure

- How to decompose system?
- Size of a module
- Portability
- High degree of module cohesion
- Low coupling between modules

9.3.1 ATTRIBUTES OF DESIGN

- Three design attributes are as follows:

Abstraction—An object is abstract if unnecessary details are removed. Similarly, abstraction in art tries to convey an image with just a few details. Abstraction in software design tries to let the designer focus on the essential issues without regard to unnecessary low-level details. Good abstraction hides the unnecessary details.

Cohesion—A material is cohesive if it sticks together. A procedure is cohesive if all the statements in the procedure are related to every output. A class is cohesive if all the attributes in the class are used by every method. That is, cohesion in a module is achieved if everything is related. High cohesion is generally considered desirable.

Originally, cohesion was defined in terms of types of cohesion. The types included coincidental, logical, temporal, procedural, communicational, sequential, and functional. Temporal cohesion was when all functions were grouped together, since they had to be performed at the same time. Logical cohesion was when the functions logically belonged together.

Coupling—Coupling is a measure of how interconnected modules are. Two modules are coupled if a change to a variable in one module may require changes in the other module. Usually the lowest coupling is desirable.

EXAMPLE 9.6

Evaluate the abstraction in the borrow functionality in the library problem.

The `borrow` function appears in three classes: `library`, `patron`, and `bookcopy`. The best abstraction is if the `borrow` function in library knows as few details about the `patron` and `bookcopy` functions as possible. For example, does the `borrow` function need to know about the `loan` class?

As shown in Fig. 9-3, if the `borrow` function in `library` just calls the `borrow` function in one of the lower levels, then it has good abstraction. That lower class will be handle the details of creating the `loan` instance and passing the pointer to the other lower-level class.

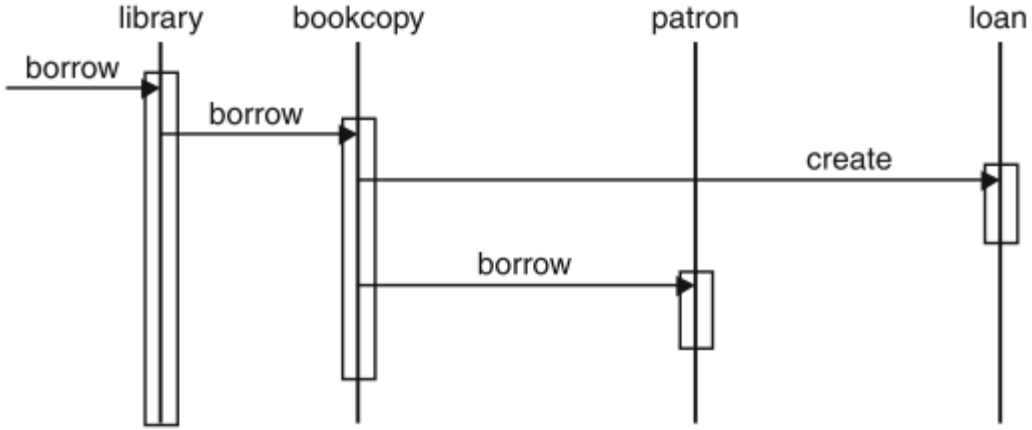


Fig. 9-3

9.3 Design Concepts

If, however, the `borrow` function in `library` knows about the `loan` class, it can check the availability of the `book`, create the `loan` instance, and call both lower-level `borrow` functions to set the values of the `loan` instance. (See Fig. 9-4.)

The version in Fig. 9.5 does not have good abstraction. That is, the details of the lower-level classes have not been hidden from the borrow function in library.

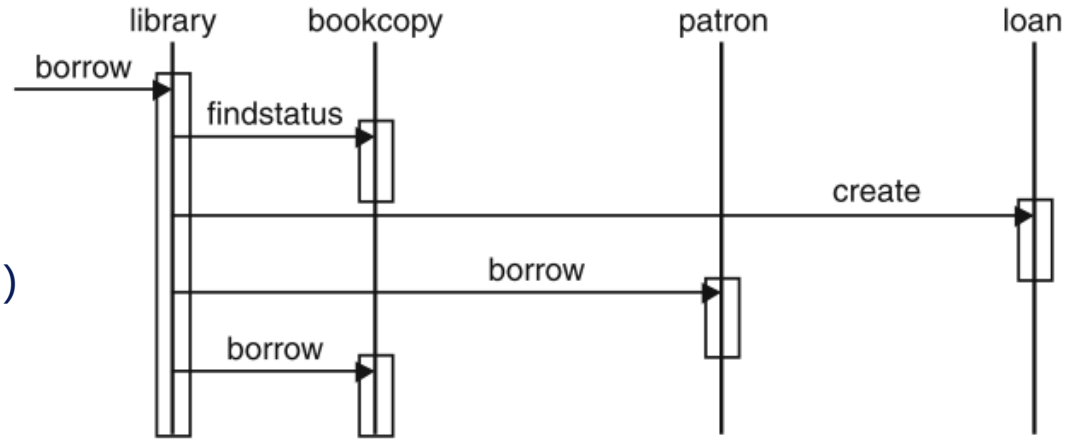


Fig. 9-4

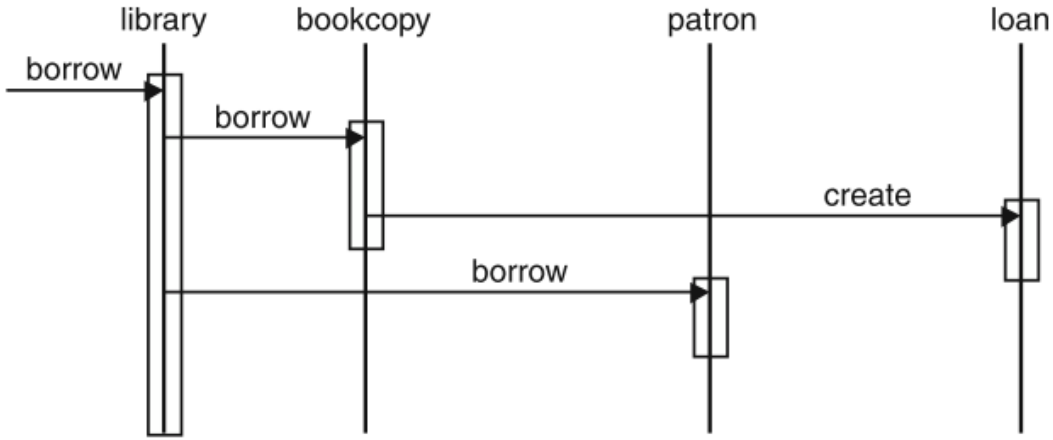
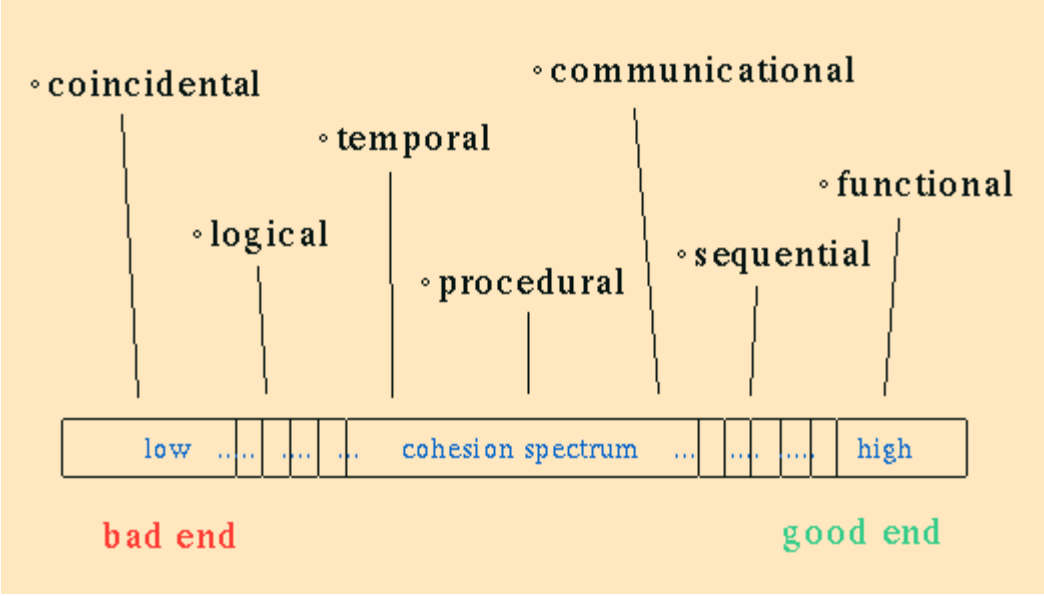


Fig. 9-5. Borrow interaction diagram—version 1.

Module Cohesion

- Cohesion is the measure of strength of the association of elements within a module. Modules whose elements are strongly and genuinely related to each other are desired.
- A module should be highly cohesive.



Module Cohesion

- Coincidental cohesion
  - A module has *coincidental cohesion* if its elements have no meaningful relationship to one another.
  - Performs multiple, completely unrelated actions
  - No reusability
  - Poor correct maintenance and enhancement

- Logical cohesion
  - A *logically cohesive* module is one whose elements perform similar activities and in which the activities to be executed are chosen from outside the module.
  - Module performs a series of related actions, one of which is selected by the calling module
  - Parts of the module are related in a logical way, but not the primary logical association

Module Cohesion

- Temporal cohesion
  - A temporally cohesive module is one whose elements are functions that are related in time.
  - Modules performs a series of actions that are related by time
  - Often happens in initialization or shutdown
  - Degrades to temporal cohesion if time of action changes
  - (Example)
    - Consider a module called "On_Really_Bad_Failure" that is invoked when a Really_Bad_Failure happens. The module performs several tasks that are not functionally similar or logically related, but all tasks need to happen at the moment when the failure occurs. The module might
      - cancel all outstanding requests for services
      - cut power to all assembly line machines
      - notify the operator console of the failure
      - make an entry in a database of failure records

Module Cohesion

- Procedural cohesion
  - A *procedurally cohesive* module is one whose elements are involved in different activities, but the activities are sequential.
  - Action based on the ordering of steps
  - Related by usage in ordering
    - Module **read part number from an input file and update directory count**
  - Changes to the ordering of steps or purpose of steps requires changing the module abstraction
  - Limited situations where this particular sequence is used is limited

- Communicational cohesion
  - A *communicationally cohesive* module is one whose elements perform different functions, but each function references the same input information or output.
  - Actions are related but still not completely separated
    - Module **update record in database and write it to the audit trail**
    - **Module calculate new trajectory and send it to the printer**
  - Module cannot be reused

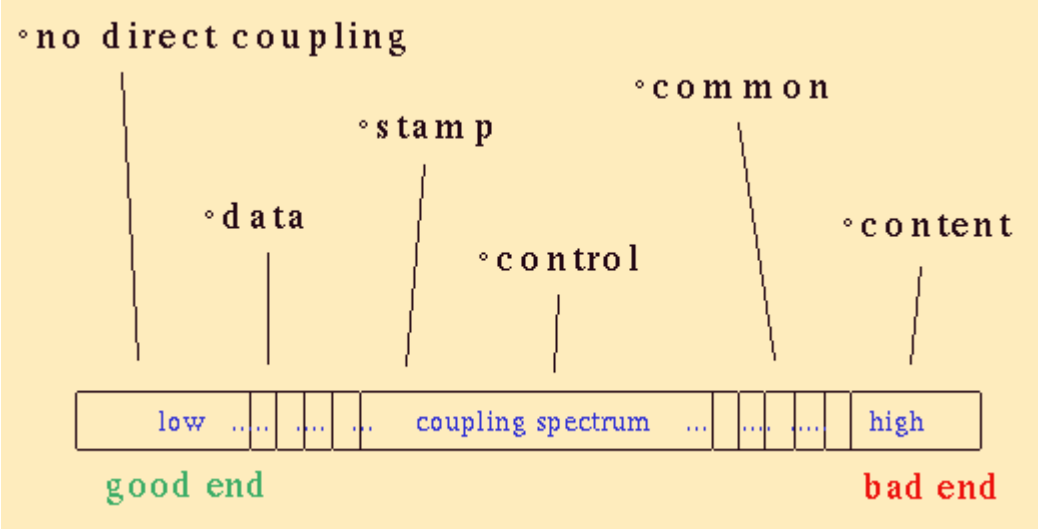
Module Cohesion

- Sequential cohesion
  - A *sequentially cohesive* module is one whose functions are related such that output data from one function serves as input data to the next function.
  - if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.
    - in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

- Functional cohesion
  - A functionally cohesive module is one in which all of the elements contribute to a single, well-defined task. Object-oriented languages tend to support this level of cohesion better than earlier languages do.
  - Module that performs a single action or achieves a single goal
  - Maintenance involves the entire single module
  - Very reusable because the module is completely independent in action of other modules
  - Can be replaced easily

Module Coupling

- Coupling is the measure of the interdependence of one module to another.
- Modules should have low coupling.
- Low coupling minimizes the "ripple effect" where changes in one module cause errors in other modules.



Module Coupling

- Data coupling
  - Two modules are data coupled, if they communicate through a parameter.
  - An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc.
- Stamp coupling
  - Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.
- Control coupling
  - Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another.
  - An example of control coupling is a flag set in one module and tested in another module.

Module Coupling

- Common coupling
  - Two modules are common coupled, if they share data through some global data items.

- Content coupling
  - Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

9.6 Requirements Traceability

- Requirements traceability tries to link each requirement with a design element that satisfies the requirement. Requirements should influence design. If a requirement does not have a corresponding part of the design or a part of the design does not have a corresponding part in the requirements, there is a potential problem. Of course,some requirements do not have a specific part of the design that reflects the requirement, and some parts of the design may be so general that no part of the requirements requires that section.
- One approach to check traceability is to draw a matrix. On one axis will be listed all the requirements items, and on the other will be the list of design items. A mark will be placed at the intersection when a design item handles a requirement.

9.6 Requirements Traceability

EXAMPLE 9.10

Draw a matrix showing the tracing of the requirements in the following description of the B&B problem and the design.

Requirements:

Tom and Sue are starting a bed-and-breakfast in a small New England town. [1] They will have three bedrooms for guests. [2] They want a system to manage the [2.1] reservations and to monitor [2.2] expenses and profits. When a potential customer calls for a [3] reservation, they will check the [4] calendar, and if there is a [5] vacancy, they will enter [6.1] the customer name, [6.2] address, [6.3] phone number, [6.4] dates, [6.5] agreed upon price, [6.6] credit card number, and [6.7] room number(s). Reservations must be [7] guaranteed by [7.1] 1 day's payment. Reservations will be held without guarantee for an [7.2] agreed upon time. If not guaranteed by that date, the reservation will be [7.3] dropped.

Design:

```
Class[A]B&B attributes: [A.1]day*daylist[DAYMAX];
  [A.2]reservation*reslist[MAX];[A.3]transaction*
  translist[TRANSMAX]
  methods:[A.4]display calendar by week
     [A.5]display reservations by customer
     [A.6]display calendar by month
Class[B]day attributes:[B.1]date thisdate
  [B.2]reservation*rooms[NUMBEROFROOMS]
  methods:[B.3]create(),[B.4]addreservation(),
  [B.5]deletereservation()
Class[C]reservation attributes:[C.1]string name
  [C.2]string address[C.3]string creditcardnumber
  [C.4] date arrival[C.5]date guaranteeby
  [C.6] int numberofdays[C.7]int roomnumber
  methods:[C.8]create()[C.9]guarantee()
  [C.10]delete()
Class[D]transaction attributes:[D.1]string name
  [D.2] date postingdate[D.3]float amount
  [D.4] string comments
```

9.6 Requirements Traceability

	A	A.1	A.2	A.3	A.4	A.5	A.6	B	B.1	B.2	B.3	B.4	B.5	C	C.1	C.2	C.3
1										X							
2	X																
2.1			X							X	X	X	X	X			
2.2				X													
3			X														
4					X	X	X										
5																	
6.1															X		
6.2																X	
6.3																	
6.4																	
6.5																	
6.6																	X
6.7																	
7																	
7.1																	
7.2																	
7.3																	

	C.4	C.5	C.6	C.7	C.8	C.9	C.10	D	D.1	D.2	D.3	D.4
1												
2												
2.1												
2.2								X				
3												
4												
5												
6.1												
6.2												
6.3												
6.4	X		X									
6.5												
6.6												
6.7				X								
7						X						
7.1												
7.2		X										
7.3							X					

9.6 Requirements Traceability

As shown in the tables above, there are a number of blank rows and blank columns. Requirement 5 is related to vacancies. There is not explicit handling of vacancies, although a vacancy should be the absence of a reservation on a particular date. Requirement 6.3 is the customer phone number, and it is missing. Requirement 6.5 is the agreed upon price, which is missing from the reservation information. Requirement 7.1 mentions the 1 day's payment, which is also not in the attributes.

Column A.1 is the daylist, which is included to help search for vacancies. B and B.1 are necessary but not specific to a requirement. C.8 is a constructor. D.1 through D.4 are details of the transactions, which are neglected in the requirements.

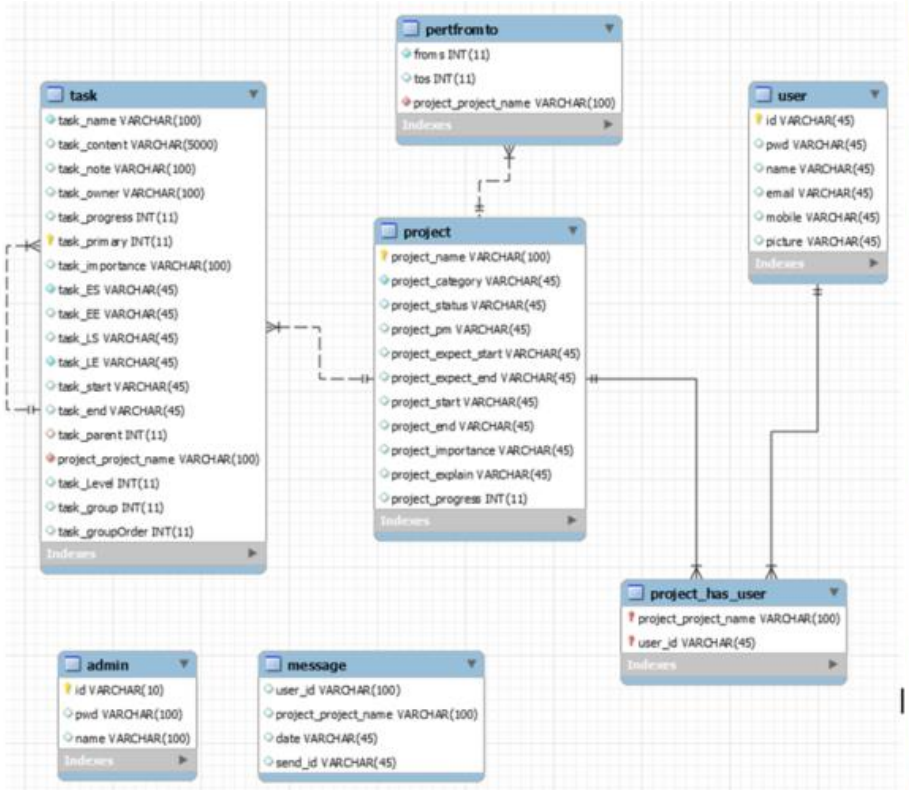
II. Contents for SDS

1. Database Design

ER(Entity Relationship) Diagram is used for database design.

Table including attributes, types, length, and description is specified for database design.

Example for Database design



Project	설명
project_name	프로젝트의 이름
project_category	프로젝트의 카테고리
project_staus	프로젝트의 진행상태
project_pm	프로젝트의 매니저 아이디
project_expect_start	프로젝트의 예상 시작일
project_expect_end	프로젝트의 예상 마감일
project_start	프로젝트의 시작일
project_end	프로젝트의 마감일
project_importance	프로젝트의 중요도
project_explain	프로젝트의 설명(상세 설명)
project_progress	프로젝트 진척도

II. Contents for SDS

2. User Interface Design

UI which was designed in requirement engineering phase is updated for implementation.

II. Contents for SDS

4. Sequence Diagram

Sequence diagram is designed for each function.

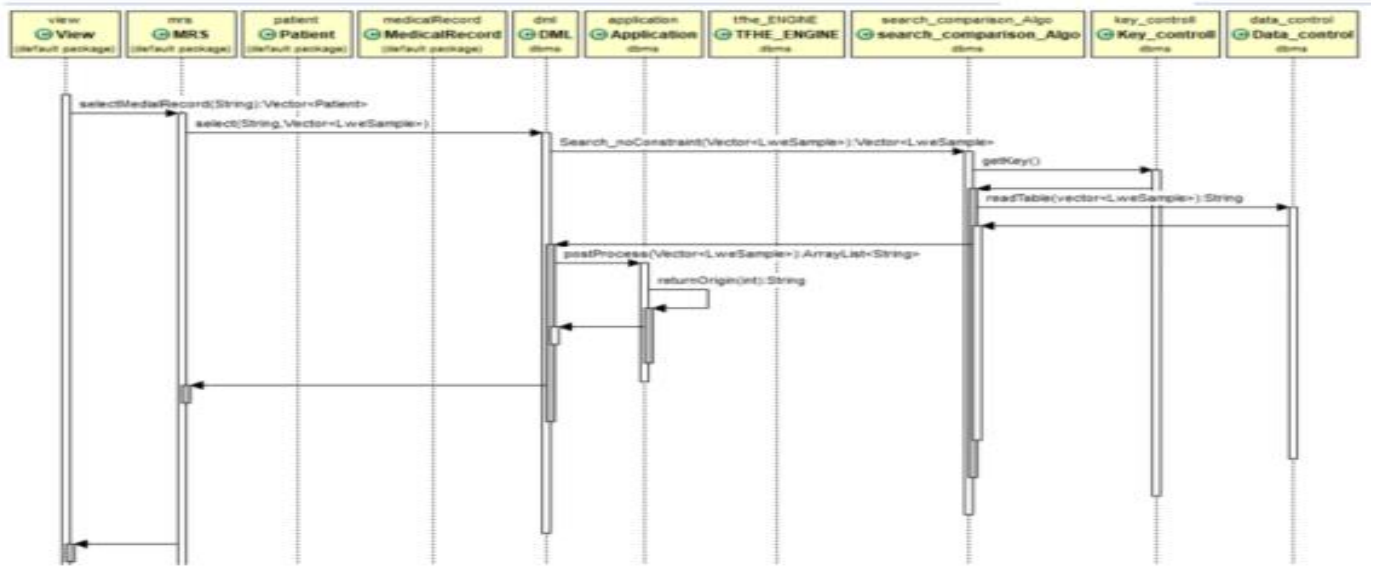
Classes(instance) and actors(instance) are predefined in usecase diagram and class diagram.

All methods which are appeared in sequence diagram should be defined in class diagram.

Example for sequence design

- **모든 데이터 열람 : selectMedicalRecord()+조건절이 없는 경우**

  모든 데이터를 열람할 경우에는 조건 절이 없기 때문에 seach_noConstraint를 수행하고 postprocess를 거쳐 View객체에서 결과 데이터들을 출력한다.



5. Interface Design between modules

Parameters and returnd values for each method in a class are defined in interface design.

Description for process of the method may be added.

Example for interface design

- MRS 클래스 : 사용자 관점에서의 기능들을 정의함.

function	parameter	return	process
insertPatientRecord	Patient	int	사용자가 입력한 환자 정보를 저장한다.
	입력할 환자 정보	1 또는 0	Patient 객체를 생성 후, Patient 클래스의 setter 함수를 이용해 객체에 데이터를 입력하고, 객체와 데이터베이스명, 테이블명, setter함수의 명(컬럼명)을 preprocess(), encryption() 순서대로 진행한 후 select()함수에 보낸다. 모든 과정을 거치고 성공하면 1, 실패하면 0을 반환한다.
insertMedicalRecord	String,MedicalRecord	int	사용자가 입력한 환자 정보를 저장한다.
	이름, 입력할 진료 정보	1 또는 0	MedicalRecord객체를 생성 후,환자의 이름과 진료기록을 MedicalRecord 클래스의 setter함수를

6. Detailed Design

Process, usually algorithm, for each method is described in psedo code form.

Outside libraries and API's are specified.

Example for detailed design

클래스명	PROJECT		
Function	Input	Output	process
Create	Project	Integer	프로젝트 생성하기
LOGIC	클라이언트에서 Project 생성 창에서 Project 객체 정보를 서버로 보낸다. 그리고 그 값을 projectservice.create(project)를 이용하여 새로운 프로젝트를 생성하는 명령을 Mybatis를 이용하여 Mapper로 보낸다. INSERT INTO project(project_name, project_category, project_explain, project_pm , project_expect_start, project_expect_end, project_start, project_end, project_progress) VALUES (#{project_name}, #{project_category},#{project_explain}, #{project_pm},#{project_expect_start}, #{project_expect_end}, #{project_start}, #{project_end}, #{project_progress}) 를 실행하여 프로젝트 정보를 저장한다.		