CLASS DESIGN

Unit 5: System Design and Class Design

Review of analysis:

- Domain analysis: Domain class model

1. Find Classes.
2. Prepare a data dictionary.
3. Find associations.
4. Find attributes of objects and links.
5. Organize and simplify classes using inheritance.
6. Verify that access paths exist for likely queries.
7. Iterate and refine the model.
8. Reconsider the level of abstraction.
9. Group classes into packages

Application analysis: Application Class model

1. Specify user interface

2. Define boundary classes

3. Define controllers

4. Check against the interaction model

Class design:

- The purpose of the class design is to complete the definition of the classes, associations and choose algorithms for operations.

- During class design, the developer expands and optimize analysis model.

Overview of Class Design:

- The analysis model describes the information that the system must contain and the high-level operations that it must perform.

- During design, choose among the different ways to realize the analysis classes for **minimizing execution time, memory, and other cost measures.**

- OO design is an iterative process.

- New classes may be introduced to store intermediate results during program execution and avoid re-computation.

Class design involves the following steps.

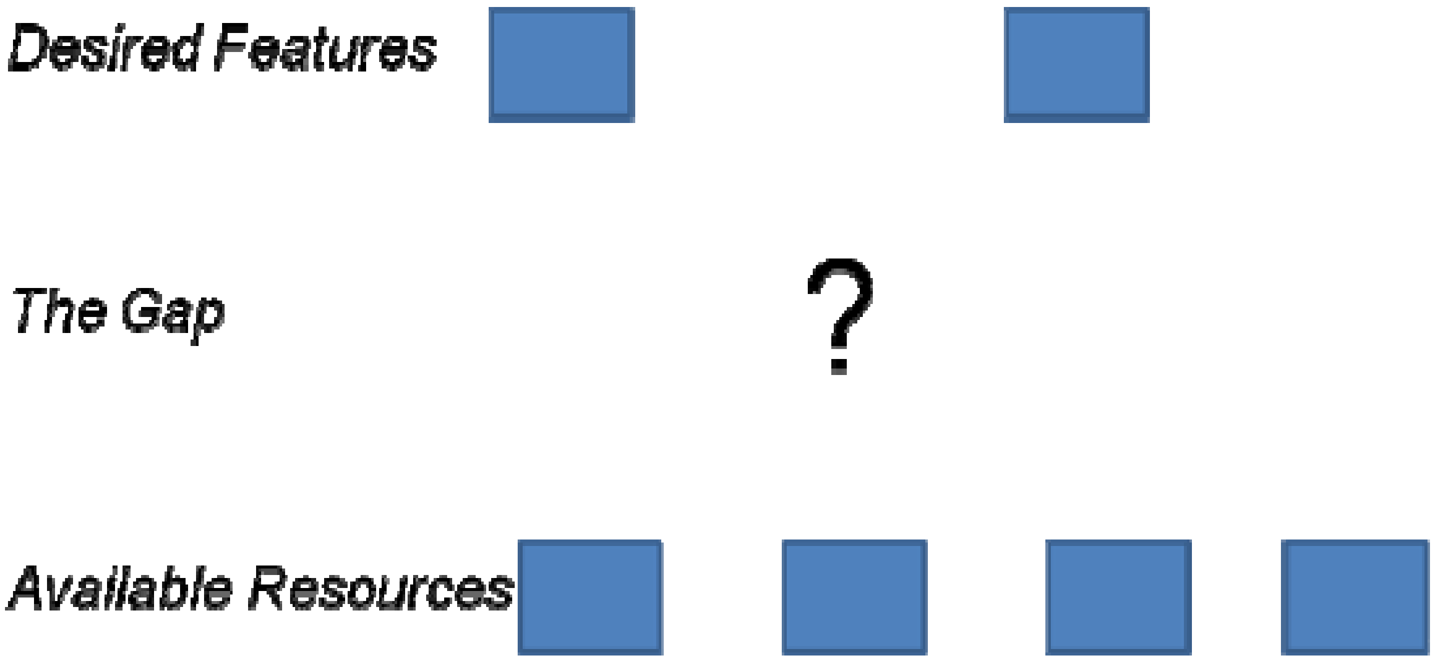
1. Bridge the gap from high-level requirements to low-level services.
2. Realize use cases with operations
3. Formulate an algorithm for each operation.
4. Recurse downward to design operations that support higher-level operations.
5. Refactor the model for a cleaner design.
6. Optimize access paths to data.
7. Reify behaviour that must be manipulated.
8. Adjust class structure to increase inheritance.
9. Organize classes and associations.

1. Bridging the Gap

- There is a **set of features** that you want your system to achieve.
- You have a **set of available resources**
- Think of the distance between them as a gap.
- Your job is to build a bridge across the gap.

Example:

- The design gap: there is often disconnection between the desired features and available resources.



2. Realizing Use Cases

- We added major operations to class model.
- Now during class design we elaborate the complex operations most of which comes from use cases.
- Use cases define the required behaviour, but they do not define its realization i.e., purpose of design – to choose among options and prepare for implementation.

3. Designing algorithms

- Now formulate an algorithm for each operation.
- The analysis specification tells what the operation does for its clients, but algorithm shows how it is done.

Steps followed to design algorithm:

1. Choose algorithms that minimize the cost of implementing operations.

2. Select data structures appropriate to the algorithms.

3. Define new internal classes and operations as necessary.

4. Assign operations to appropriate classes

4. Recursing Downward

- Organize operations as layers: operations in higher layer invoke operations in lower layers.

- The design process generally works top-down. you start with the higher level operation and proceed to define lower level operations.

Downward recursion proceeds in two main ways:

- By functionality
- By mechanism

Functionality Layer

- Take the required high-level functionality and break it into lesser operations.

- Combine similar operations and attach the operations to classes.

- Operations that are carefully attached to classes have much less risk than free-floating functionality.

Mechanism Layers:

- Build the system out of layers of needed support Mechanisms.
- Various mechanisms are needed to
  - store information,
  - sequence control,
  - coordinate objects,
  - transmit information,
  - perform computations, and to provide other kinds of computing infrastructure .

example:

- In constructing tall buildings you need an infrastructure of support girders, utility conduits, a building controlled devices.

- These are not directly part of the user's needs for space, but they are needed to evaluate chosen architecture.

5. Refactoring

- Initial design of set of operations contain inconsistencies, redundancies and inefficiencies because it is impossible to get a large design correct in one pass.

- Refactoring includes:

"Changes to the internal structure of software to improve its design without altering its external functionality ".

6. Design Optimization

- The good way to design a system: first get the logic correct and then optimize it .

- It is difficult to optimize a design at the same time as you create it.

- So once you have logic in place, run the application, measure its performance and then fine tune it.

Design Optimization involves the following tasks:

- Provide efficient access paths: adding redundant associations to minimize access cost and maximize convenience
- Rearrange the computation for greater efficiency: identify and eliminate dead path and optimize the algorithm.
- Save intermediate results to avoid re-computation: saving the derived attributes to avoid recomputation of complicated expressions.

7. Reification of Behaviour

- Reification is the promotion of something that is not an object into an object.
- Behaviour usually meets this description.
- By reifying behaviour, it can be stored, passed to other operations, and can be transformed.
- Reification adds complexity but expands the flexibility of the system.

8. Adjustment of Inheritance

❖ You can often Adjust the definitions of classes and operations to increase inheritance by performing some steps :

1. Rearrange classes and operations to increase inheritance.

2. Abstract common behaviour out of groups of classes.

3. Use delegation to share behaviour when inheritance is semantically invalid

- **Rearranging classes and operations:** use the following kinds of adjustments to increase the chance of inheritance

1. **Operations with optional arguments:** for eg: a draw operation on a monochrome display does not need a color parameters, but it can accept the parameter and ignore it for consistency with color display.

2. **Operations that are special cases:** for eg: appending an element to a list is a special case of inserting an element into list; the insert point simply follows that last element. These special operations can be implemented by calling the general operations with appropriate parameters.

3. **Inconsistent names:** similar attributes in different classes may have different names. Give the attributes the same name and move them to a common ancestor class.

Also watch for similar operations with different names.

4. **Irrelevant operations:** an operation may be defined on several different classes in a group, but may not be required in other classes. Define those operation on the common ancestor class and can be declared as no-operation on the classes that don’t care about it.

Abstracting out common behavior out of group of classes:

- During design phase, new classes & operations are often added which lead to commonality between classes.
- When there is common behavior, you can create a common superclass for the shared features, leaving only the specialized features in the subclasses.
- This transformation of the class model is called **abstracting out a common superclass** or common behavior.

- **Using Delegation to share behavior:**
- Often when you are tempted to use inheritance as an implementation technique, you could achieve the same goal in a safer way by making one class as an attribute or associate of the other class.
- Then one object can selectively invoke the desired operations of another class, using delegation rather than inheritance.
- **Delegation** consists of catching an operation on one object and sending it to a related object.

9. Organizing a Class Design

- Programs consist of discrete physical units that can be edited, compiled, imported, or otherwise manipulated.

- The organization of a class design can be improved with the following steps.

1. Hide internal information from outside view.

   One way to improve the viability of a design is by carefully separating external specification from internal implementation. This is called information hiding.

2. Maintain coherence of entities.

- Coherence is another important design principle. An entity, such as a class, an operation, or a package , is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal.
- An entity should have single major theme; it should not be a collection of unrelated parts.

3. Fine-Tune definition of packages.

- During analysis you partitioned the class model into packages.
- This initial organization may not be suitable or optimal for implementation.

- Define packages so that their interfaces are minimal and well defined

- The interface between two packages consists of the association that relate classes in one package to classes in the other and operations that access classes across package boundaries.

Eg: Architecture of ATM system

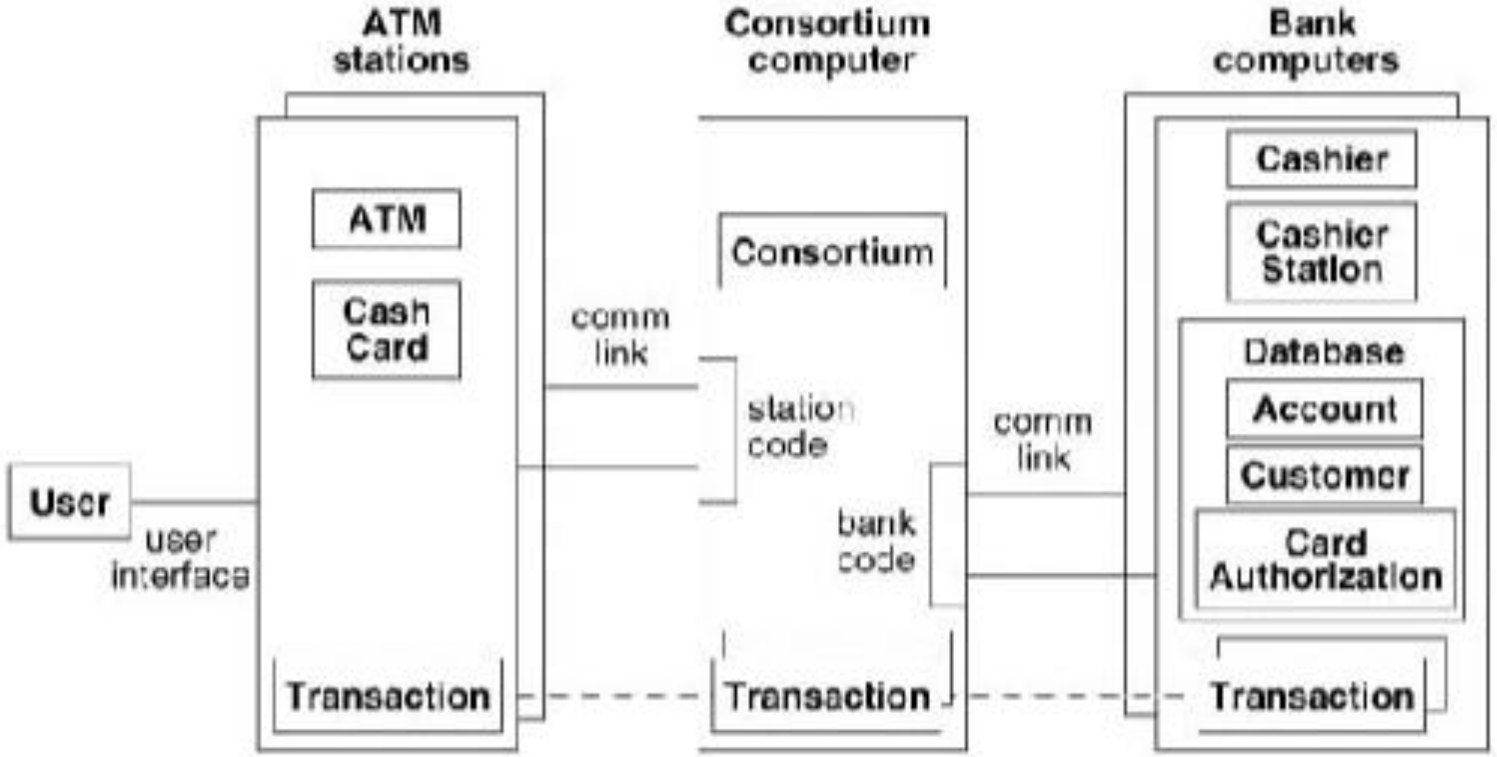


Figure 14.2 Architecture of ATM system. It is often helpful to make an informal diagram showing the organization of a system into subsystems.

END OF UNIT-5