UNIT-4

CLASS DESIGN

Overview of Class Design, Bridging the Gap, Realizing Use Cases, Designing Algorithms - Choosing Algorithms, Choosing Data structures, Defining Internal classes and Operations, Assigning Operations to Classes, Recursing Downward - Functionality Layers, Mechanism Layers, Refactoring, Design Optimization - Adding Redundant associations for Efficient Access, Saving derived values to avoid Re-computation, Rectification of Behavior, Adjustment of Inheritance - Rearranging Classes and Operations, Abstracting out Common Behavior, Using Delegation to share Behavior

Organizing a Class Design - Information Hiding, Coherence of Entities, Fine Tuning Packages

Case Study – ATM, Library Management System (Class Diagram, Object Diagram ,Use case Diagram ,Sequence Diagram, Collaboration Diagram ,State Diagram ,Activity Diagram ,Component Diagram ,Deployment Diagram)

- The analysis phase determines *what* the implementation must do
- The system design phase *determines the plan of attack*
- The purpose of the class design is to complete the *definitions of the classes* and *associations* and choose *algorithms* for operations

Overview of Class Design – steps

- Bridging the gap
- Realizing Use Cases
- Designing Algorithms
- Recursing Downward
- Refactoring
- Design Optimization
- Reification of Behavior
- Adjustment of Inheritance
- Organizing a Class Design

1. Bridging the gap

Bridge the gap from high-level requirements to low-level services

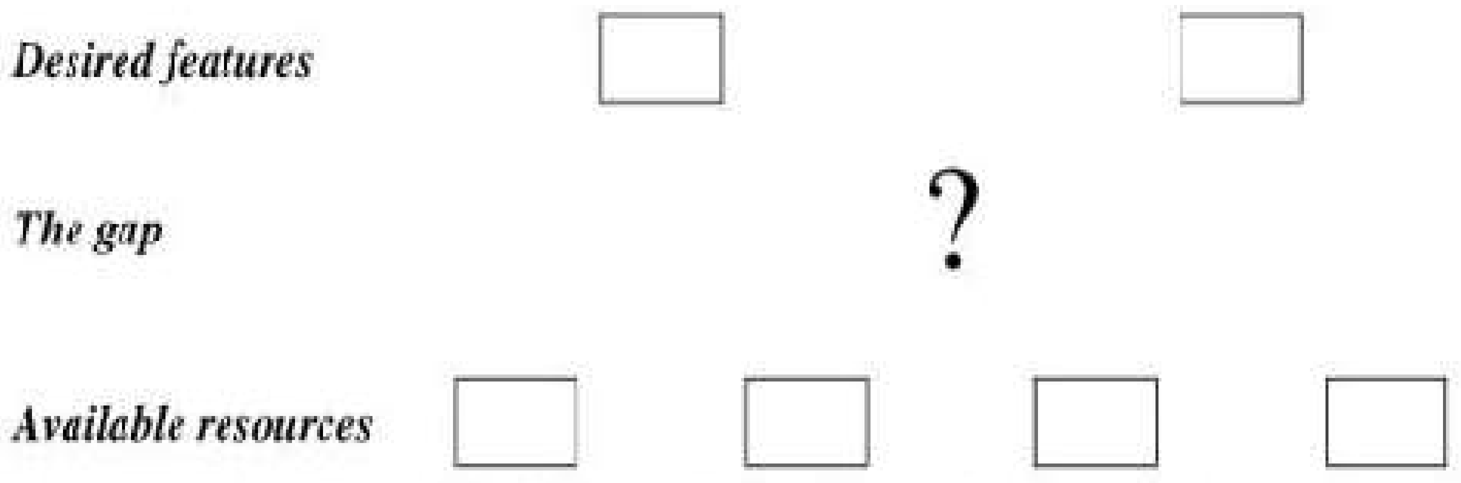


Figure 15.1 The design gap. There is often a disconnect between the desired features and the available resources.

- Salesman can use a spreadsheet to construct formula for his commission – readily build the system
- Web-based ordering system – cannot readily build the system because too big gap between the resources and features

❖The intermediate elements may be operations, classes or other UML constructs.

❖You must invent intermediate elements to bridge the gap

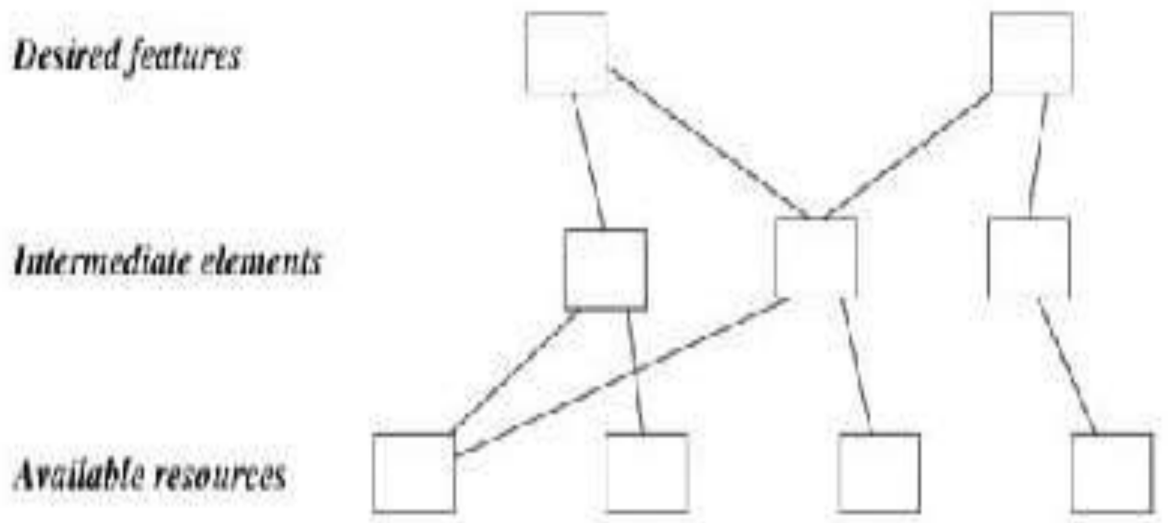


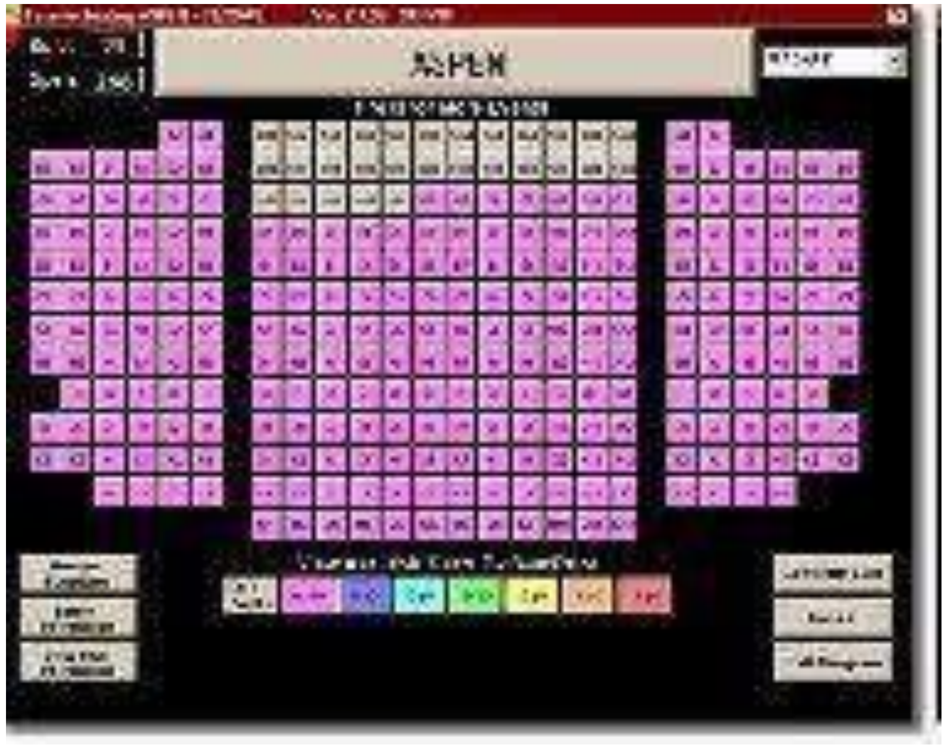
Figure 15.2 Bridging the gap. You must invent intermediate elements to bridge the gap between the desired features and the available resources.

2. Realizing Use Cases

- Realize use cases with operations.
- The cases define system-level behavior.
- During design you must invent new operations and new objects that provide this behavior.

Step1: List the **responsibilities** of a use case or operation.

- A ***responsibility*** is something that an object knows or something it must do.
- For Example:
  - An **online theater ticket system**
  - Making a reservation has the **responsibility** of
    - Finding unoccupied seats to the desired show,
    - Marking the seats as occupied,
    - Obtaining payment from the customer,
    - Arranging delivery of the tickets, and
    - Crediting payment to the proper account



Step2: Each operation will have various responsibilities.

Group the responsibilities into **clusters** and try to make each cluster coherent.

Step3: Define an operation for each responsibility cluster.

Step4: Assign the new lower-level operations to classes

- ATM Example
- *Process transaction* includes:
  - Withdrawal includes responsibilities:
    - Get amount from customer,
    - verify that amount is covered by the account balance,
    - verify that amount is within the bank‟s policies,
    - verify that ATM has sufficient cash, ….
    - A database transaction ensures all-or-nothing behavior.

- Deposit
  - Get amount from customer
  - Verify the entered amount
  - Displays the total amount
  - Verify that source amount within the bank’s policy
  - Credit bank account

- Transfer
  - get source account
  - Get target account
  - Get amount
  - Verify that source account covers amount
  - Verify that source amount within the bank’s policy
  - Debit the source account
  - Credit the target account
  - Post an entry on the customer’s receipt

- Use Case for the ATM

3. Designing Algorithms

- Formulate an ***algorithm*** for each operation
- The analysis specification tells *what* the operation does for its clients
- The algorithm show *how* it is done

Designing Algorithms- steps

- Choose *algorithms* that minimize the cost of implementing operations.
- Select *data structures* appropriate to the algorithms
- Define new internal classes and operations as necessary.
- Assign operations to appropriate classes.

- **Choosing algorithms (Choose algorithms that minimize the cost of implementing operations)**
  - When efficiency is not an issue, you should use simple algorithms.
  - Typically, 20% of the operations consume 80% of execution time.
  - Considerations for choosing alternative algorithms
    - Computational complexity
    - Ease of implementation and understandability
    - Flexibility
      - Simple but inefficient
      - Complex efficient

- ATM Example
  - Interactions between the consortium computer and bank computers could be complex.
  - Issues are
    - Distributed computing
    - The scale of consortium computer (scalability)
    - The inevitable conversions and compromises in coordinating the various data formats.
  - All these issues make the choice of algorithms for coordinating the consortium and the banks important

❖Choosing Data Structures (select data structures appropriate to the algorithm)

- Algorithms require data structures on which to work.
- They organize information in a form convenient for algorithms.
- Many of these data structures are instances of ***container classes***.
- Such as arrays, lists, queues, stacks, set…etc.

❖Defining New Internal Classes and Operations

- To invent new, low-level operations during the decomposition of high-level operations.
- The expansion of algorithms may lead you to create new classes of objects to hold intermediate results
- ATM Example:
  - *Process transaction* uses case involves a customer receipt.
  - A *Receipt* class is added.

❖Assigning Operations to Classes (assign operations to appropriate classes)

- How do you decide what class owns an operation?
  - Receiver of action
  - To associate the operation with the target of operation, rather than the initiator.
- Query vs. update
- The object that is changed is the target of the operation
- Focal class
- Class centrally located in a star is the operation‟s target
- Analogy to real world

ATM Example

***Process transaction* includes:**

- Withdrawal includes responsibilities:
  - Get amount from customer, verify that amount is covered by the account balance, verify that amount is within the bank‟s policies, verify that ATM has sufficient cash, ….
  - A database transaction ensures all-or-nothing behavior.
- Deposit
- Transfer
- Customer.getAccount(),
- account.verifyAmount(amount),
- bank.verifyAmount(amount),
- ATM.verifyAmount(amount)
- ATM.disburseFunds(amount)

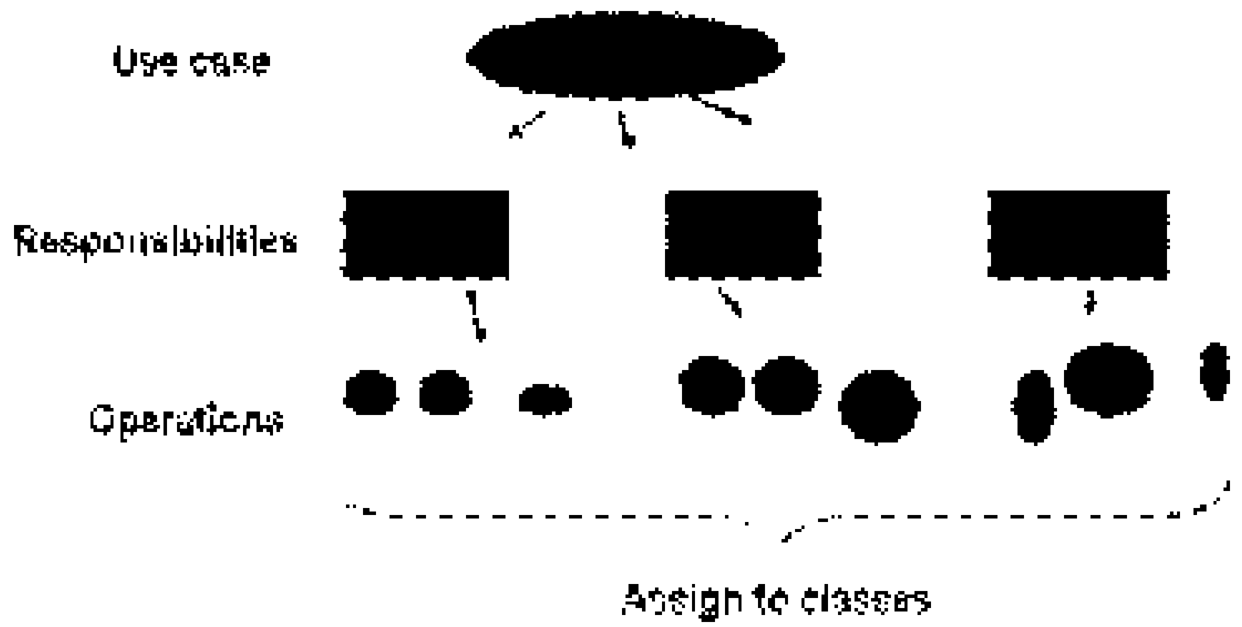
- Account..debit(amount)
- Reciept.postTransaction()
- ATM.receiveFunds(amount)
- Account.credit(amount)

Recursing Downward

- To organize operations as layers.
  - Operations in higher layers invoke operations in lower layers.
- Two ways of downward recursion:
  - By functionality
  - By mechanism
- Any large system mixes functionality layers and mechanism layers.

Functionality Layers

- Take the required high-level functionality and break it into lesser operations.
- Make sure you combine similar operations and attach the operations to classes.
- An operation should be coherent meaningful, and not an arbitrary portion of code.
- ATM eg., **use case** decomposed into **responsibilities [withdrawal]**
  - Resulting **operations** are assigned to classes [Account.verifyAmount(amount)
  - If it is not satisfied rework them


Use case
Responsibilities
Operations
Assign to classes

Mechanism Layers

- ❑Build the system out of layers of needed support mechanisms.
- ❑These mechanisms don‟t show up explicitly in the high-level responsibilities of a system, but they are needed to make it all work.
- ❑E.g. Computing architecture includes
- ❑Data structures, algorithms, and control patterns.
- ❑A piece of software is built in terms of other, more mechanisms than itself.

Refactoring

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

- You must revisit your design and rework the classes and operations so that they clean satisfy all their uses and are conceptually sound.

ATM Example

Operations of process transaction

Account.credit(amount)

Account.debit(amount)

Combine into

Account.post(amount)

Overview of Class Design – steps

- Bridging the gap
- Realizing Use Cases
- Designing Algorithms
- Recursing Downward
- Refactoring
- Design Optimization
- Reification of Behavior
- Adjustment of Inheritance
- Organizing a Class Design

Design Optimization

- To design a system is to first get the logic correct and then optimize it.
- Often a small part of the code is responsible for most of the time or space costs.
- It is better to focus optimization on the **critical areas,** than to spread effort evenly.
- Design Optimization
- Optimized system is **more obscure** and less **likely to be reusable**.
- You must strike an appropriate **balance between efficiency and clarity.**

Tasks to optimization:

- Provide efficient access paths.
- Rearrange the computation for greater efficiency.
- Save intermediate results to avoid recomputation.

Adding Redundant Associations for Efficient Access

- Rearrange the associations to optimize critical aspects of the system.

Consider employee skills database

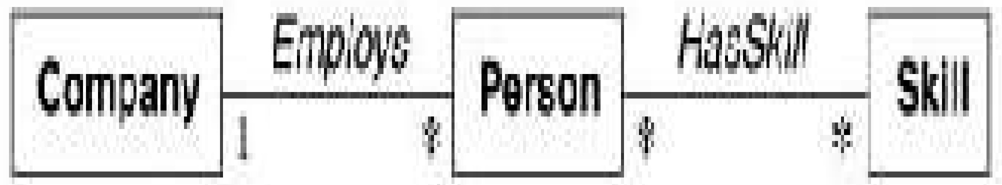


Figure 15.5 Analysis model for person skills. Derived data is undesirable during analysis because it does not add information.

- *Company.findSkill*( ) returns a set of persons in the company with a given skill.
- Suppose the company has 1000 employees,.
- In case where the number of hits from a query is low because few objects satisfy the test, an *index* can improve access to frequently retrieved objects.

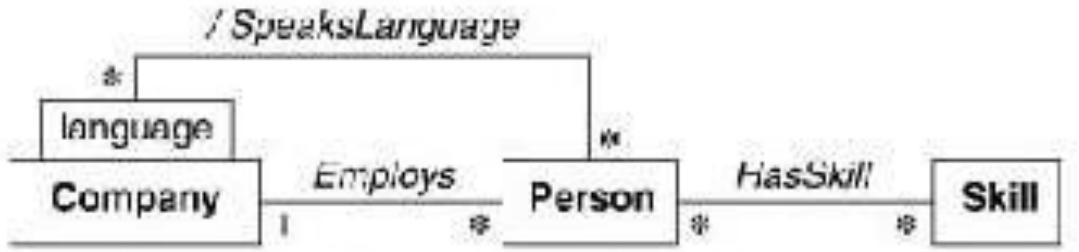


Figure 15.6 Design model for person skills. Derived data is acceptable during design for operations that are significant performance bottlenecks.

- Examine each operations and see what associations it must traverse to obtain its information.
- Next, for each operation, note the following,
  - Frequency of access
  - Fan-out
  - Selectivity

ATM Example

Banks must report cash deposits and withdrawals greater than $10,000 to the government.

Rearranging execution order for efficiency

- Optimizing the algorithm
- One key to algorithm optimization is to eliminate dead paths as early as possible
- Example: suppose an application must find all employees who speak both Japanese and French
- Suppose 5 employees speak Japanese and 100 speak French
- It is better to test the Japanese speakers first, then test if they speak French
- ATM example: sometimes bank requires not only the report of individual updates that are greater than normal limit but also report suspicious activities

Saving Derived values to avoid recomputation

- Sometimes it is helpful to define new classes to cache derived attributes and avoid recomputation
- We must update the cache if any of the objects on which it depends are changed
- There are 3 ways to handle updates
  - Explicit update
  - Periodic recomputation
  - Active values

Reification of Behavior

- Reification is the promotion of something that is not an object into an object
- Reification is made to store, pass or modify the behavior of the code at run time
- Reification adds complexity but can expand the flexibility of a system
- Reifying of the code can be made by encoding it into an object and decoding it when it is run
- The resulting run-time cost may or may not be significant
  - If the encoding invokes high-level procedures, cost is only a few indirections
  - If the entire behavior is encoded in a different language , it must be interpreted
  - If encoded behavior constitutes a small part of the run-time execution time the performance overhead may not matter

Adjustment of inheritance

Adjusting inheritance by performing following steps

- Rearrange classes and operations to increase inheritance
- Abstract common behavior out of groups of classes
- Use delegation to share behavior when inheritance is semantically invalid

Rearranging classes and operations

- More often operations in different classes are similar but not identical
- By adjusting the definitions of the operations, we may be able to cover them with a single inherited operation
- We can use the following kinds of adjustments to increase the chance of inheritance
  - Operation with optional arguments
  - Operations that are special cases
  - Inconsistent names
  - Irrelevant operations

Abstracting out common behavior

When there is common behavior, we 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**

The meaning of abstract superclass that there are no direct instances of it, but the behavior it defines belongs to all instances of its subclasses

Example : draw() operation in **“Figure” class**

The rendering varies among different figures, such as circles, lines and rectangle but the figures can inherit featues like color, line thickness etc parameters from abstract class **“Figure”**

- The creation of abstract superclasses also improves the extensibility of a software product
- Example : we are developing temperature sensing module for a large computerized control system
  - We must use specific type of sensor(model J55)
  - Particular way of reading the temperature
  - Formula for converting the raw numeric reading into degrees celsius

Using delegation to share behavior

- In Generalization, operation in baseclass and subclass must provide the same service
- Sometimes programmers use inheritance as an implementation technique with no intension of guaranteeing the same behavior
- Example: stack class inherit from list class
- Delegation consists of catching an operation on one object and sending it to a related object
- We delegate only meaningful operations so there is no danger of inheriting meaningless operations by accident

- Example:

List class is available ,

we need a **Stack** class.

How about subclassing the **Stack** class from the **List** class

Using delegation instead of inheritance

Inheritance: Extending a Base class by a new operation or overwriting an operation.

Delegation: Catching an operation and sending it to another object.

We can delegate only meaningful operations so there is no danger of inheriting meaningless operations by accident

Organizing a class design

Can improve the organization of a class design with the following steps

- Hide internal information from outside view
- Maintain coherence of entities
- Fine-tune definition of packages

1. Information hiding

- One way to improve viability of a design is by carefully separating external specification from internal implementation. This is called information hiding
- Then we can change the implementation of a class without requiring to modify code
- Several ways to hide information
  - Limit the scope of class-model traversal
  - Do not directly access foreign attributes
  - Define interfaces at high level of abstraction
  - Hide external objects
  - Avoid cascading method calls

Limit the scope of class-model traversal

- A method could traverse all associations of the class model to locate and access an object
- Methods that know too much about a model are fragile and easily invalidated by changes
- During design we should try to limit the scope of any one method
- An object should access only objects that are directly related

Do not directly access foreign attributes

- Generally it is acceptable for subclasses to access the attributes of an associated class
- However, classes should not access the attributes of its associated class
- Instead , call an operation to access the attributes of associated class

Define interfaces at high level of abstraction

- It is desirable to minimize the class couplings
- One way to do this is by raising the level of abstraction of interfaces

Hide external objects

- Boundary object is used to isolate the interior of system from its external environment
- Boundary object is an object whose purpose is to mediate requests and responses between inside the system and outside the system
- It accepts external requests in a client-friendly form and transform then into a form convenient for the internal implementation

Avoid cascading method calls

- Avoid applying a method to the result of another method
- Instead , consider writing a method to combine the two operations

2. coherence of entities

- 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
- A single method should not contain both **policy** and **implementation**
- **Policy** is the making of context- dependent decisions
- **Implementation** is the execution of fully- specified algorithms

Example :

An operation to credit interest on checking account

Interest is calculated daily based on balance, but all interest for a month is lost if the account is closed

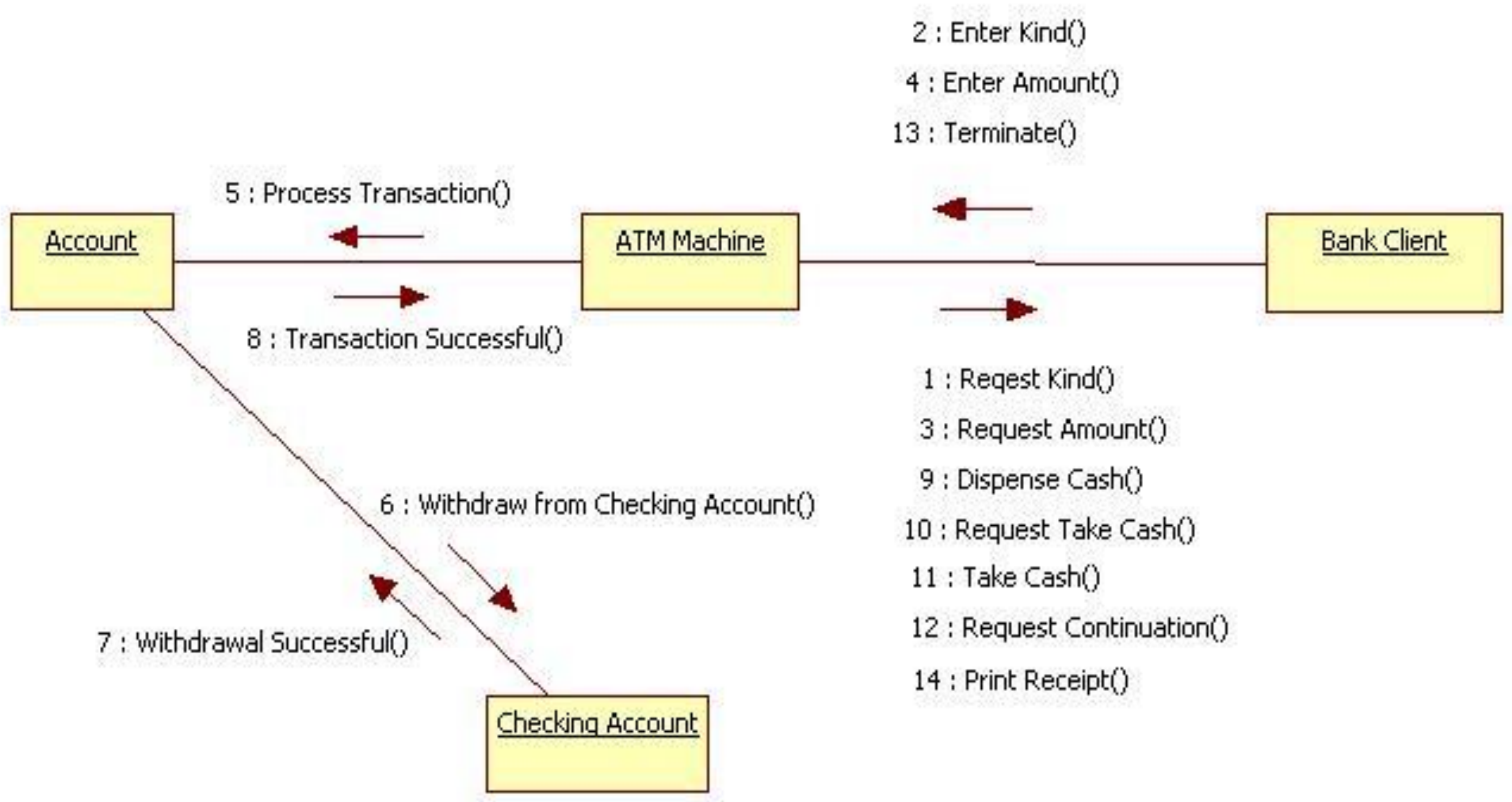
The interest logic consists of 2 parts: implementation method that computes the interest due between a pair of days

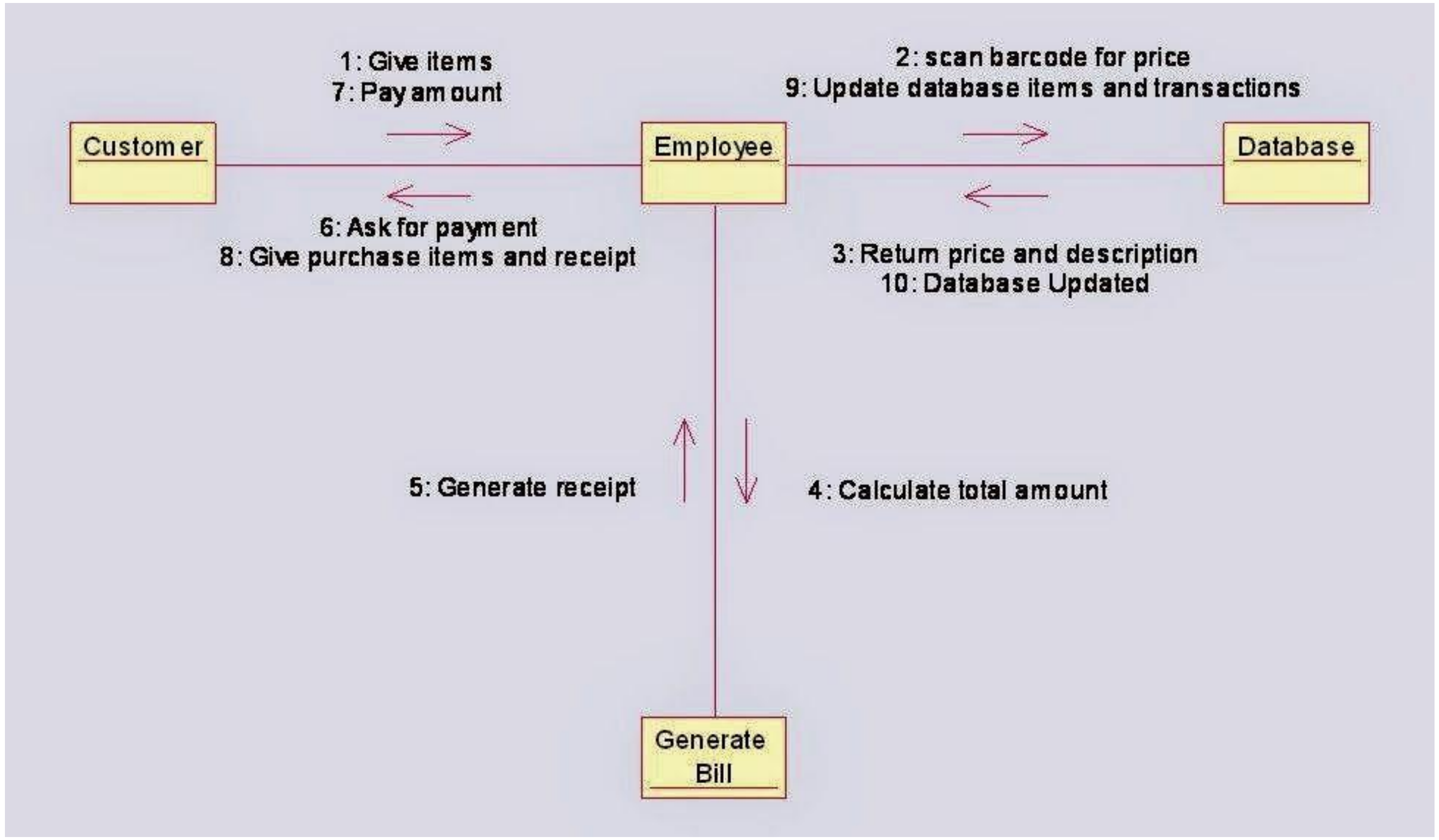
Policy method that decides whether and for what interval the implementation method is called

3. Fine- Tuning packages

- During analysis we partition the class model into packages
- But it may not be suitable during implementation
- We should define packages so that their interfaces are minimal and well defined
- The interface between 2 packages consist of associations that relate classes in one package to classes in other

Communication diagram




1: Give items
7: Pay amount
2: scan barcode for price
9: Update database items and transactions
Customer
Employee
Database
6: Ask for payment
8: Give purchase items and receipt
3: Return price and description
10: Database Updated
5: Generate receipt
4: Calculate total amount
Generate Bill

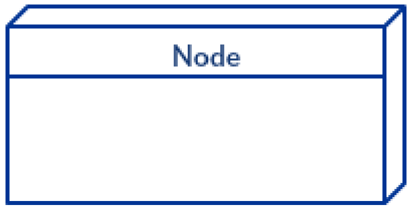
Deployment

A deployment diagram is a UML diagram type that shows the execution architecture of a system, including nodes such as hardware or software execution environments, and the middleware connecting them.

Deployment diagrams are typically used to visualize the physical hardware and software of a system.

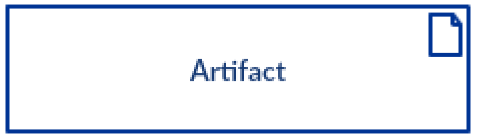
Deployment Diagram Notations

Nodes :A node, represented as a cube, is a physical entity that executes one or more components, subsystems or executables. A node could be a hardware or software element

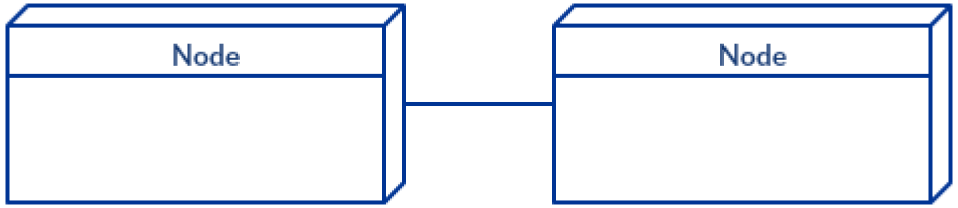


Artifacts

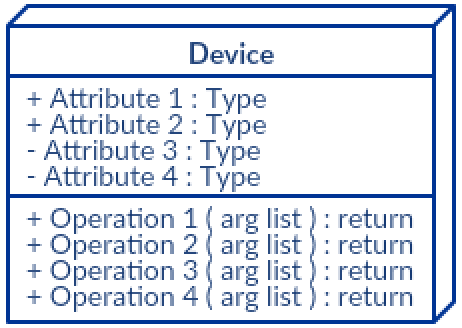
Artifacts are concrete elements that are caused by a development process. Examples of artifacts are libraries, archives, configuration files, executable files etc.



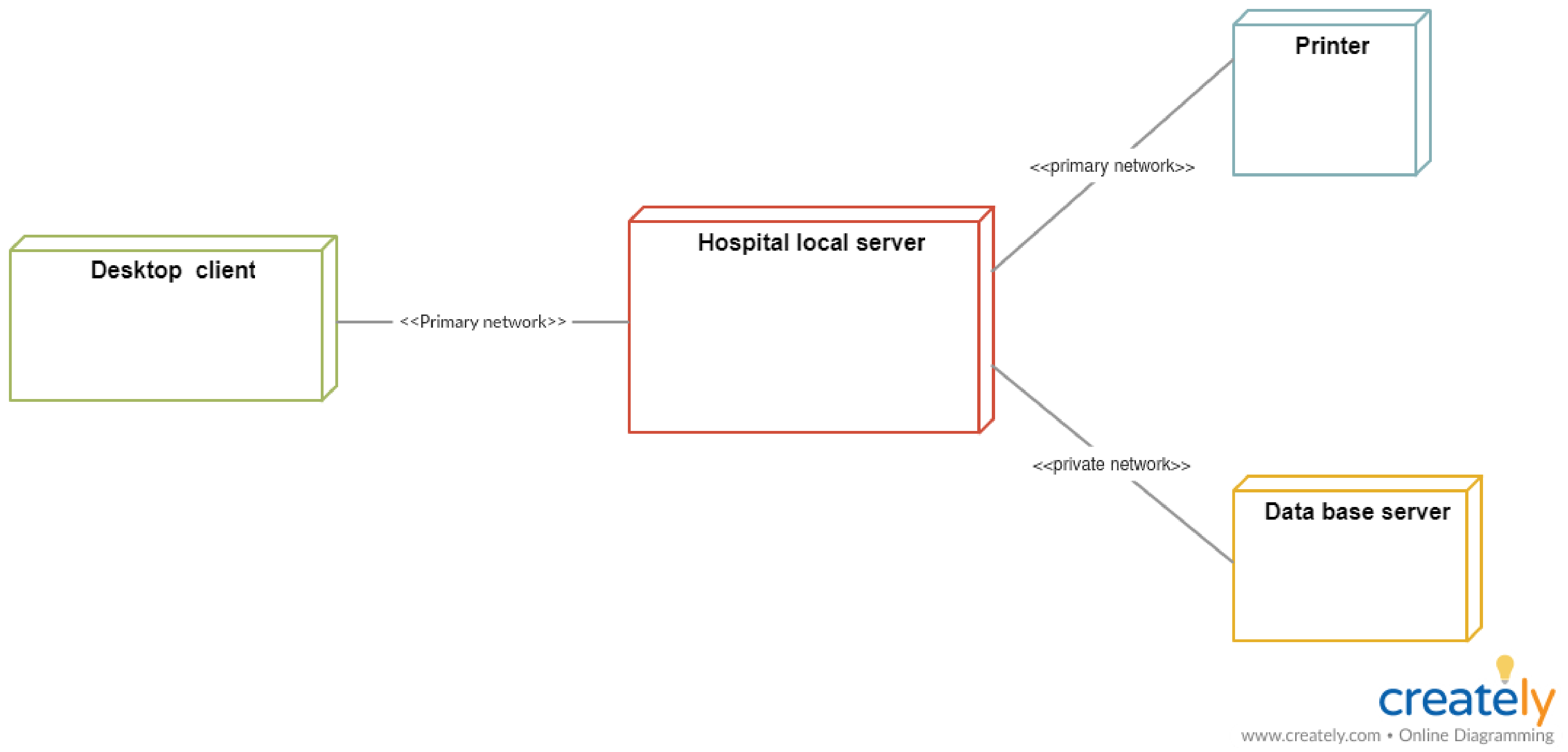
Communication Association - This is represented by a solid line between two nodes.



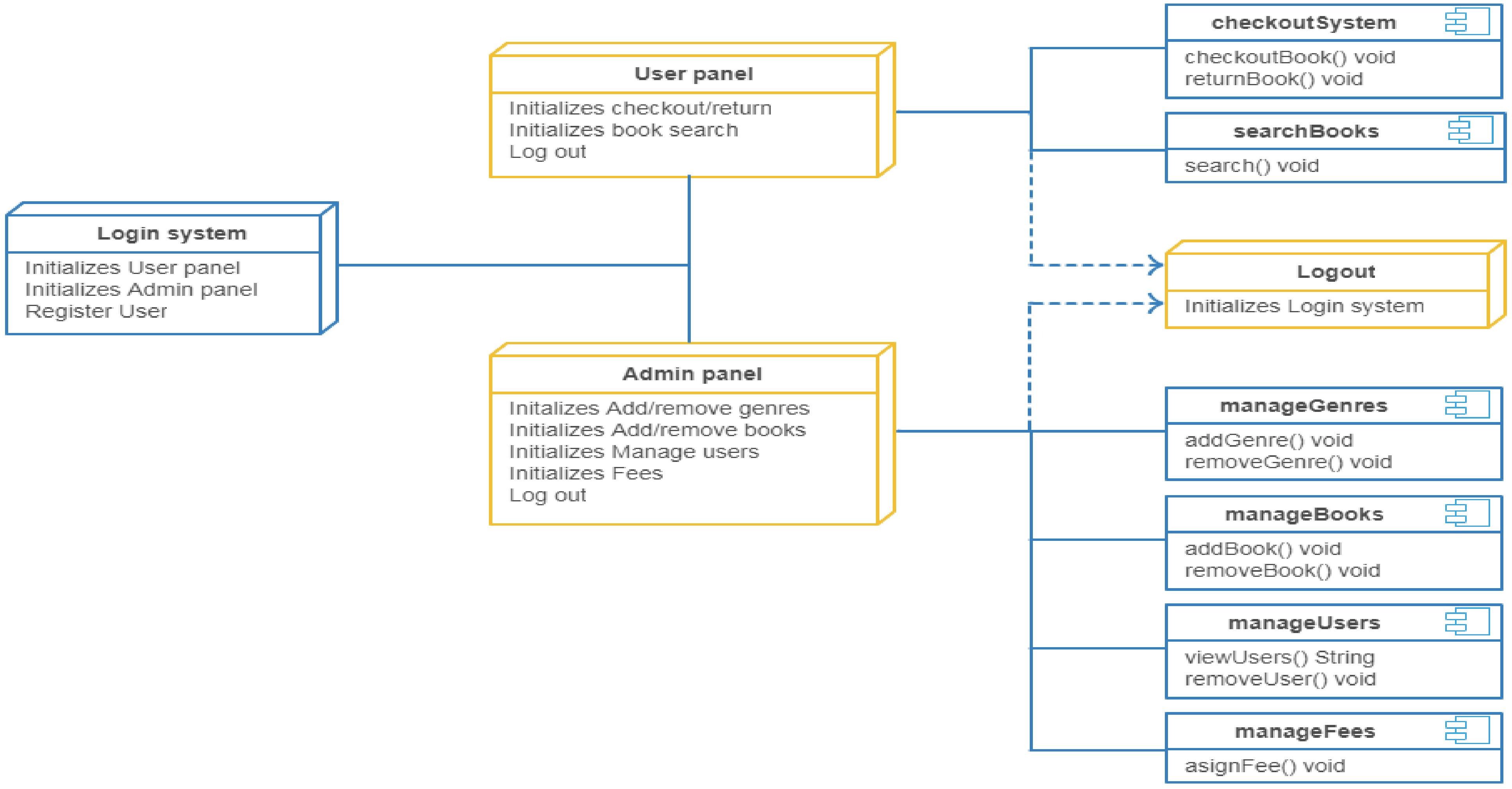
Devices - A device is a node that is used to represent a physical computational resource in a system. An example of a device is an application server.


Device
+ Attribute 1 : Type
+ Attribute 2 : Type
- Attribute 3 : Type
- Attribute 4 : Type
+ Operation 1 (arg list) : return
+ Operation 2 (arg list) : return
+ Operation 3 (arg list) : return
+ Operation 4 (arg list) : return

Ex: hospital management system



Ex: library management system



THANK YOU