II.9. OOP Software Design Methodology

The software engineering process

Systematic object-oriented software development follows the steps shown in Figure 1. This is known as the waterfall model. In practice, it is iterative and each stage is revisited several times.

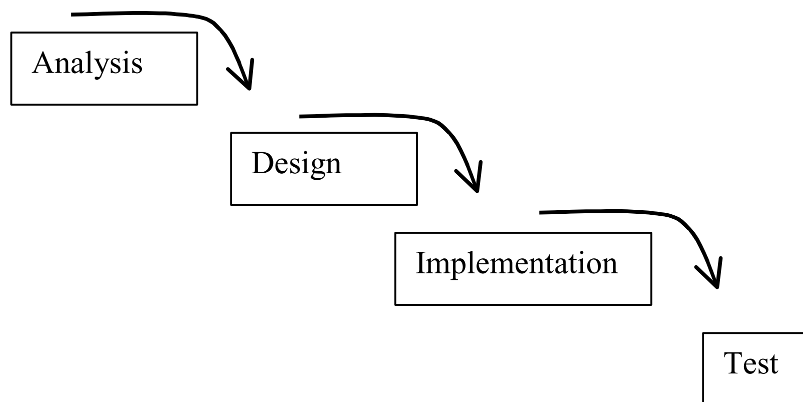


Figure 1. The waterfall model of software developement

Analysis

The analysis stage takes a software vision and articulates it into a set of requirements. This is done through:

Use Cases Analysis

A "use case" is a high-level description of how the software will be used. A use case identifies an "actor" (a software user) and how he/she will interact with the system (which at this stage is as a black box). E.g.:

Use case: Customer withdraws cash from ATM machine.

Actor: Customer

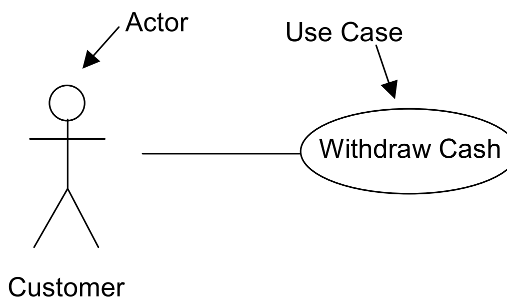


Figure 2. Use case diagram

Each use case can occur under different situations, called "scenarios". E.g.:

- Customer requests $300 from checking account; machine puts cash in slot; machine prints receipt
- Customer requests $300 from checking account; checking account only has $200; machine informs customer "Insufficient funds".
- Customer requests $300 from checking account; daily withdrawal limit is only $200; machine informs customer "Over daily limit".

Domain Analyis

A "domain analysis" identifies the domain objects and their relationships. It can be done by examining the use case scenarios and creating a class for every noun. E.g.:

> *"**Customer** chooses to withdraw **cash** from **checking**. Sufficient cash is in the **account**. Sufficient cash and **receipts** are in the **ATM**, and the **network** is up and running. The ATM asks the customer to indicate an **amount** for the **withdrawal**, and the customer asks for $300, a legal amount to withdraw at this time. The **machine** dispenses $300 and prints a receipt, and the customer takes the **money** and the receipt."*

might result in the following objects:

Customer

Cash (money, amount, withdrawal)

Checking

Account

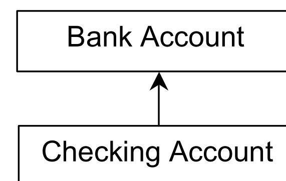
Receipts

ATM (machine)

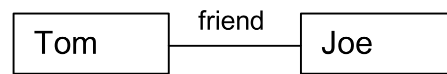
Network

Relationships between objects fall into the following categories:

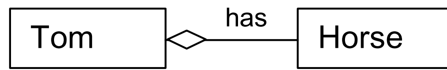
Generalization – One class is a general type of a more specific class. E.g., "Checking account" is a type of "bank account." This relationship is drawn with an arrow.



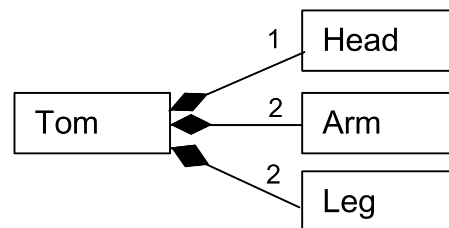
Association – One object knows about and interacts with another object. E.g., Tom has a friend Joe. This is drawn with a simple line.



Aggegration - One object owns or contains another object, but both objects can exist independently. E.g., Tom has a horse. This is drawn with a line with an open diamond.



Composition – One object is made up of the other object(s). The other objects only exist if the first object exists, and vice versa. E.g., Tom has a head, two arms and two legs. This is drawn with a line with a filled-in diamond.



<u>Interaction Diagrams</u>

Each scenario in the use cases requires interactions between objects. These interactions are shown in an "interaction diagram". Arrows in this diagram, going from one object to another, are numbered in chronological order, according to the sequence of interactions. For the ATM withdrawal example the interaction diagram is:

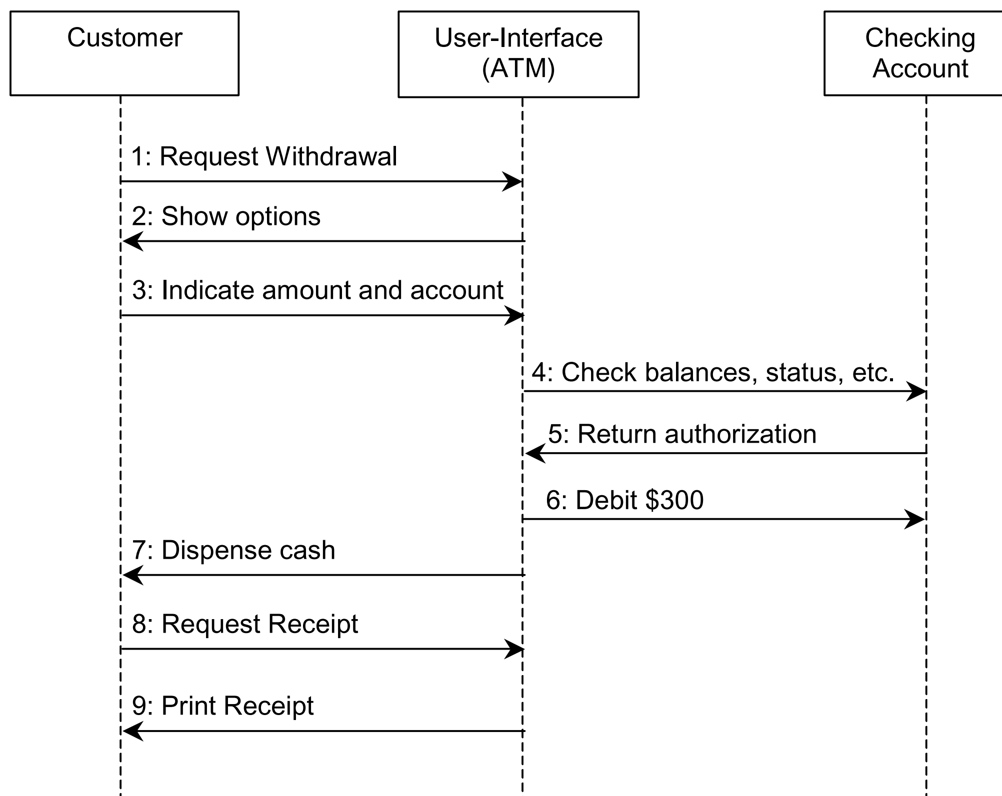


Figure 3. Interaction Diagram

These diagrams and their conventions are part of the "Unified Modeling Language (UML)".

Design

Whereas the analysis stage focuses on understanding the requirements, the design stage focuses on creating a solution. The vague objects above are defined more precisely in "design" classes. These design classes are not in terms of any particular programming language.

Implementation

In the implementation, "design" classes are coded as C++ class declaration and function definitions. Each generalization relationship is implemented through inheritance. Association, aggregation and composition are implemented through member variables. For composition relationships, objects are responsible for creating and deleting objects they are composed of. For each interaction, an object must have a function.

Test

The code is then tested to verify that it meets the requirements from the analysis phase.

Design Patterns

After accumulating many man-years of experience, object-oriented program developers have discovered patterns that re-occur often in programs. These are common situations for which the solutions have now become well understood. These patterns have been published in various books.[1] The solutions for handling these situations have been given names so that it is easy for programmers to communicate with each other about them. We have already seen one of these patterns when we discussed "adapters."

Another pattern is the "object factory" which solves the problem of creating the correct type of object when requested. For example, we may have code that creates "engine" objects corresponding to different types of engines. A function CreateEngine(string engine_type) returns a new engine object of the type requested by the string variable engine_type. Later, someone creates sub-classes for each type of engine in order to interface with a Drawable class that allows drawing shapes in a window. If we were to use the CreateEngine() function it would still return the old type

---

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley,

of object without the Drawable interface. The solution to this problem is to make the CreateEngine() function a member function in a "factory" class. We then request CreateEngine() from the factory object. If the set of objects to be created changes then we can create a new factory sub-class that over-rides the CreateEngine() function to yield the new set of objects. It is a simple matter to use the new factory object instead of the old factory object.

In this course we will not discuss design patterns anymore as a topic. You just need to be aware that they exist.