Object-oriented concepts

The five basic concepts of object-oriented design are the implementation level features that are built into the programming language. These features are often referred to by these common names:

- **Object/Class**: A tight coupling or association of data structures with the methods or functions that act on the data. This is called a class, or object (an object is created based on a class). Each object serves a separate function. It is defined by its properties, what it is and what it can do. An object can be part of a class, which is a set of objects that are similar.

- **Information hiding**: The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as private or protected to the owning class.

- **Inheritance**: The ability for a class to extend or override functionality of another class. The so-called subclass has a whole section that is derived (inherited) from the super class and then it has its own set of functions and data.

- **Interface**: The ability to defer the implementation of a method. The ability to define the functions or methods signatures without implementing them.

- **Polymorphism**: The ability to replace an object with its sub objects. The ability of an object-variable to contain, not only that object, but also all of its sub objects.

Designing concepts

- Defining objects, creating class diagram from conceptual diagram: Usually map entity to class.
- Identifying attributes.
- Use design patterns (if applicable): A design pattern is not a finished design; it is a description of a solution to a common problem, in a context. The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

- Define application framework (if applicable): Application framework is a term usually used to refer to a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. By bundling a large amount of reusable code into a framework, much time is saved for the developer, since he/she is saved the task of rewriting large amounts of standard code for each new application that is developed.

- Identify persistent objects/data (if applicable): Identify object single runtime of the application. If a relational database is used mapping.

Input (sources) for object-oriented design

The input for object-oriented design is provided by the output of object-oriented analysis. Realize that an output artifact does not need to be completely developed to serve as input of object-oriented design; analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process. Both analysis and design can be performed incrementally, and the artifacts can be continuously grown instead of completely developed in one shot.

Some typical input artifacts for object-oriented design are:

- **Conceptual model**: Conceptual model is the result of object-oriented analysis; it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.

- **Use case**: Use case is a description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may
be end users or other systems. In many circumstances use cases are further elaborated into use case diagrams. Use case diagrams are used to identify the actor (Users or other systems) and the processes they perform.

• **System Sequence Diagram**: System Sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

• **User interface documentations (if applicable)**: Document that shows and describes the look and feel of the end product's user interface. It is not mandatory to have this, but it helps to visualize the end-product and therefore helps the designer.

• **Relational data model (if applicable)**: A data model is an abstract model that describes how data is represented and used. If an object database is not used, the relational data model should usually be created before the design, since the strategy chosen for object-relational mapping is an output of the OO design process. However, it is possible to develop the relational data model and the object-oriented design artifacts in parallel and the growth of an artifact can stimulate the refinement of other artifacts.

**Output (deliverables) of object-oriented design**

• **Sequence Diagrams**: Extend the System Sequence Diagram to add specific objects that handle the system events. A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.

• **Class diagram**: A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

**Software testing**

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of Software implementation. Test techniques include, but are not limited to, the process of executing a Program or application with the intent of finding software bugs (errors or other defects).

Software testing can be stated as the process of validating and verifying that a software Program/application/product:
1. meets the requirements that guided its design and development;
2. Works as expected;
3. Can be implemented with the same characteristics.
4. Satisfies the needs of stakeholders.

Software testing, depending on the testing method employed, can be implemented at any time in the development process.

**Purpose of Testing**

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions.

**Roles**

Software testing can be done by software testers. Until the 1980s the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different Goals in software testing, different roles have been established: **manager, test lead, test designer**,
Tester, automation developer, and test administrator.

Testing methods

1. Static vs. dynamic testing

There are many approaches to software testing. Reviews, walkthroughs, or inspections are considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing can be (and unfortunately in practice often is) omitted. Dynamic testing takes place when the program itself is used for the first time (which is generally considered the beginning of the testing stage). Dynamic testing may begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions). Typical techniques for this are either using stubs/drivers or execution from a debugger environment. For example, spreadsheet programs are, by their very nature, tested to a large extent interactively ("on the fly"), with results displayed immediately after each calculation or text manipulation.

2. The box approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

White-box testing (also known as clear box testing, glass box testing, and transparent box testing and structural testing) tests internal structures or workings of a program, as opposed to the functionality exposed to the end-user. In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT).

While white-box testing can be applied at the unit, integration and system levels of the software testing process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

Techniques used in white-box testing include:
- API testing (application programming interface) - testing of the application using public and private APIs
- Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods - intentionally introducing faults to gauge the efficacy of testing strategies
- Mutation testing methods
- Static testing methods

Black-box testing

Black-box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation. The tester is only aware of what the software is supposed to do, not how it does it. Black-box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, state transition tables, decision table testing, fuzz testing, model-based testing, use case testing, exploratory testing and specification-based testing. One advantage of the black box technique is that no programming knowledge is required.

Grey-box testing

Grey-box testing (American spelling: gray-box testing) involves having knowledge of internal data structures and algorithms for purposes of designing tests, while executing those tests at the user, or black-box level. The tester is not required to have full access to the software's source code.

Testing levels
The main levels during the development process unit-, integration-, and system testing that are distinguished by the test target without implying a specific process model. Other test levels are classified by the testing objective.

- **Unit testing**
  Unit testing, also known as component testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors. These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of each other.

- **Integration testing**
  Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together (“big bang”). Normally the former is considered a better practice since it allows interface issues to be localized more quickly and fixed. Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

- **System testing**
  System testing tests a completely integrated system to verify that it meets its requirements.

- **System integration testing**
  System integration testing verifies that a system is integrated to any external or third-party systems defined in the system requirements.

- **Top-down and bottom-up**
  Bottom up Testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested. All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage. Top down Testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.