Abdallah Salhab (21110181)

(1.1):

Object oriented programming is a programming paradigm the focuses in breaking down a code in smaller pieces of codes that is called objects, OOP makes it easy to reuse a piece of a code, because the whole code is made into parts, one part that does a job can be used in other places or codes to make that same job, so it makes the work easier on the developer to make complex applications.

<https://www.interviewbit.com/blog/characteristics-of-object-oriented-programming/>

\_inheritance:

inheritance allows an object to take the properties or characteristics of another objects, the class that got inherited from is called the (super class) and the class that inherits from the super class is called the (sub class).

There are three types of inheritance:

Single inheritance: where the sub object takes the super object properties and adds it to his.

Multi-level inheritance: where the sub class has his own sub class and so it is like a chai.

Hierarchical inheritance: where that super class can have more than one sub class.

\_encapsulation: it is putting together the methods and the attributes within the object. Encapsulation is important to provide and control the access to its data using access modifiers, and it is important for data hiding.

\_polymorphism: it allow different object types to be treated as an object of a common type of them, there is an interface or a base class that would represent the related objects. It gives the ability to the same method for all the related objects and get different responses of each of them.

\_constructor: a constructor is a function that is called to initialize a newly crated object, a constructor is like an instance of a class, and it is used to set the initial state of the object.

\_Abstraction: abstraction goal is to simplify the representation of a complex system, abstraction uses the classes to define that attributes and methods of objects. Abstraction makes it easier for the user to work with objects without needing to know to the inner working of them.

\_Interface: interface is a collection of abstract methods that can be used by any class that implements the interface, the interface defines the behaviors that the classes that implemented it should do without specifying how to do it.

\_Collection: Collections are data structures used in object-oriented programming (OOP) to store and manage groups of objects. Different operations and behaviors are offered by these collections to manage and arrange the objects they contain.

\_static: static is a keyword modifier that can be applied to a method, variable or maybe classes, and when ever some thing is static the class that has that static member owns that member, and it can be shared by all instances of that class (there is only one copy of the static member).

\*Class relationships:

1-Dependancies:

It is a relationship between two classes where one class depends on the other class, maybe because of a method or a variable or a return, and a change on the class that is depended on can change the class that depends on it, and it is resembled by a dotted arrow pointing to the class that is depended on. 

2- Generalization:

it is a relationship between classes that represent the term (is a)

Or (a kind of) relationship, and it is a form of inheritance where a sub class extends a super class to get its features and methods, and it is represented by a solid line that ends with an arrow head.



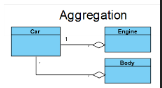
3-Realization: realization represents the relationship where a class implements an interface, and it is represented by a dotted line that ends with a closed arrow head. 

4-Association: in which there are an object of one class that is connected to an object of another class (uses a) relationship (they depend on each other), association has a role (what is the role or job of the class) and a multiplicity (how many objects maybe connected across an instance of an association) of each object.

A close-up of a line

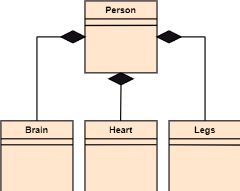
Description automatically generated with low confidence

5-Aggregation:

It is a type of association that emphasizes the ownership of an object by another object like a (has a) relationship. And the owned object can exist without the owner. The owner can be represented by a square near the owner class

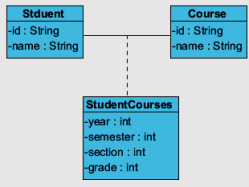
6-Composition:

It is a strong relationship that represent a (owns a)relation ship and the owned object cant exist without its owner.



7-Assosiation class:

It is a class that enables you to expand an association between classes by adding attributes and operations. It serves as a link between two classes, representing a unique object with association-specific properties and methods.



\*\*SOLID principles:

Solid principles are a set of design principles that aims to make the code maintainable and easy to understand by any one, the principals were introduced by Robert martin and his group and they are used as almost as a standard for coding in OOP.

What does SOLID mean:

The S is for single responsibility.

The O is for open closed principal.

The L is for liskov substitution.

The I is for Interface segregation.

The D is for dependency inversion.

single responsibility:

every part of the code(classes or methods) should only be responsible for one responsibility (job) only, and they should have only one reason to change that is modification or update, and it promotes high cohesion and lose coupling.

open closed principal:  
which means that the (modules, classes, and functions) should be open for extension but is closed for modification. Extension can be done using inheritance, overriding the method, or overloading it.

liskov substitution:

that is subtypes can be substitutes to their base type, that is if you replace an object of super class into an object of the subclass the code should work just fine.

(Subclasses must adhere to the contact and behavior of the superclass)

Interface segregation:

The client should not be forced to use an interface that they don’t need, it enforces the partition of an interface into smaller more specific ones, the clients should implement the interface with the methods that they need.

dependency inversion:

high level modules should not depend on low level modules and both should depend on abstractions, Abstractions should not depend on details, details should depend on abstractions.

(1.2)

Naming and clarity: clean coding concentrates about the use of meaningful descriptive names for the classes, methods and variables. Choosing proper names can help make it clear what a data structure's purpose and contents are as well as what the goals of certain operations are. And this can help the developer understand the code better and using the data structures and operations in a better more efficient way.

organization: Clean code encourages the breakdown of complex algorithms into more manageable, simple functions or methods. With the help of this flexible strategy, you can break down a difficult problem into smaller, more manageable problems that can all be resolved separately. You can create algorithms that properly utilize particular data structures and actions by separating concerns and responsibilities, keeping the code focused and accessible.

Reusable code components: Clean coding places a strong emphasis on their development. You can encapsulate data structures and operations inside of reusable functions or classes. Because the same data structures and operations can be applied across different parts of your codebase, this encourages code reuse. Reusable code components speed up development, guarantee consistency, and lower the risk of bug introduction.

Clean coding encourages the creation of algorithms that are effective in terms of time and space complexity. You can optimize your algorithms to eliminate unnecessary operations, use the least amount of memory, and perform better by utilizing the right data structures and operations.

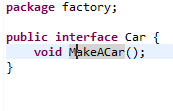
(1.2)

#Creational design patterns:

They are a set of design patterns that deal with the way of creating an object, and they help create an object with maintainability and reusability in mind.

\*Factory:

It defines an interface to create objects, and they allow the subclasses to choose the type of object they want to create.

 A screenshot of a computer code

Description automatically generated with medium confidence A picture containing text, screenshot, font, line

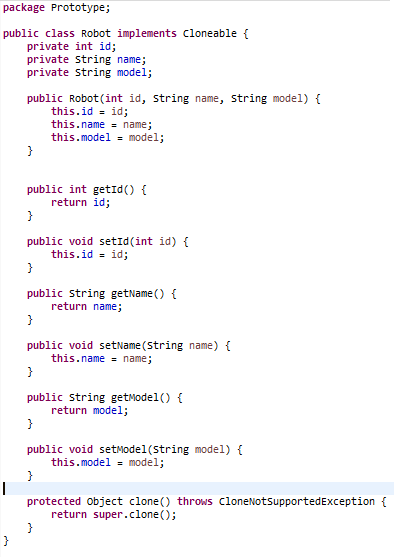
Description automatically generated A screenshot of a computer code

Description automatically generated with low confidence A screenshot of a computer program

Description automatically generated with low confidence

\*Prototype:

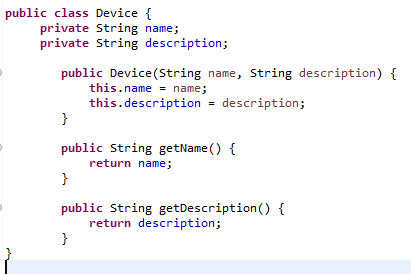
It lets the developer to create an exact copy of an existing object by cloning it to avoid overhead of creating an object from scratch.

 A picture containing text, screenshot, font, number

Description automatically generated

\*Builder:

It allows you to build complex objects piece by piece. Using the same building code, the pattern enables you to create many kinds and representations of an object.

 A screenshot of a computer code

Description automatically generated with low confidence A screenshot of a computer code

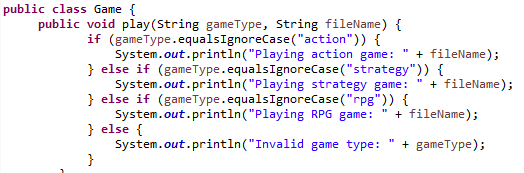
Description automatically generated with medium confidence

#Structural design patterns:

They are design patterns that deal with the composition of classes and objects to form larger structures while keeping them flexible and efficient. They primarily focus on simplifying the structure of a system by identifying relationships between objects and providing ways to organize them.

\*Adapter:

It allows an object with incompatlable interfaces to work togather.

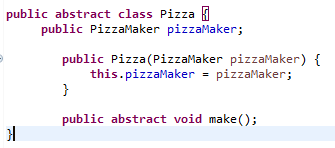
 A screen shot of a computer program

Description automatically generated with low confidence A screenshot of a computer program

Description automatically generated with medium confidence

\*bridge:

It allows you to divide a huge class or a group of classes that are closely related into two distinct hierarchies, it decouples an abstraction from its implementation.

 A close-up of a computer screen

Description automatically generated with low confidence A picture containing text, screenshot, font, line

Description automatically generated A picture containing text, screenshot, font, line

Description automatically generated A screen shot of a computer

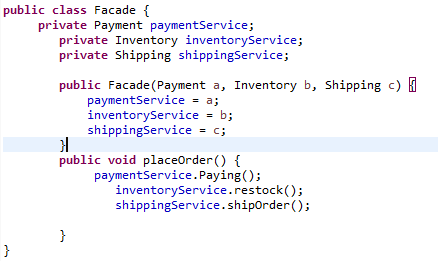
Description automatically generated with medium confidence A picture containing text, font, screenshot, line

Description automatically generated A screenshot of a computer code

Description automatically generated with low confidence

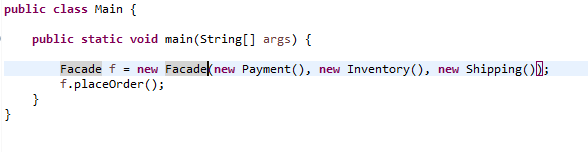
\*Façade:

Provides a simplified interface to a complex subsystem, hiding its complexities and making it easier to use. It acts as a higher level interface that makes the subsystem more accessible and easier to use.

A picture containing text, font, screenshot, line

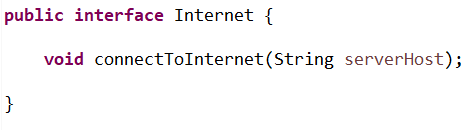
Description automatically generatedA screen shot of a computer code

Description automatically generated with low confidenceA picture containing text, font, screenshot, line

Description automatically generated

\*Proxy:

It lets you provide a placeholder for another object, and it controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

A picture containing text, screenshot, font, line

Description automatically generatedA picture containing text, screenshot, font, line

Description automatically generatedA picture containing text, screenshot, font, line

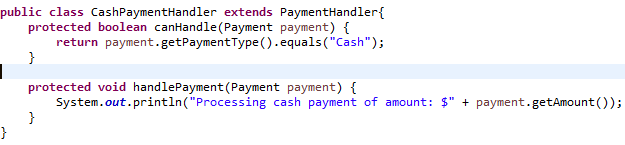
Description automatically generated

#Behavural design patterns:

They are design patterns that emphasize the communication and interaction between classes and objects. They deal with how tasks and algorithms are distributed across objects, defining how those items operate and act inside a system.

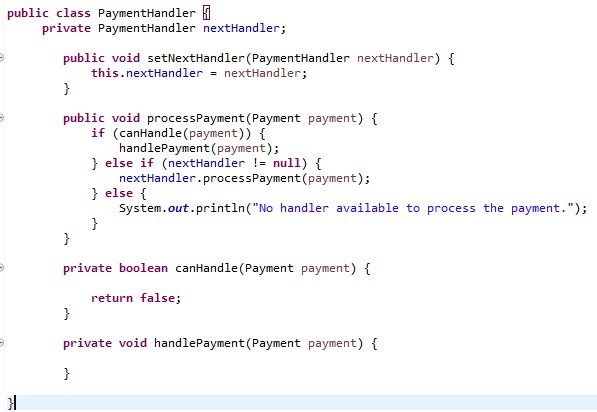
\*Chain of responsibility:

Allows an object to pass a request along a chain of potential handlers until the request is handled or reaches the end of the chain.

A screen shot of a computer code

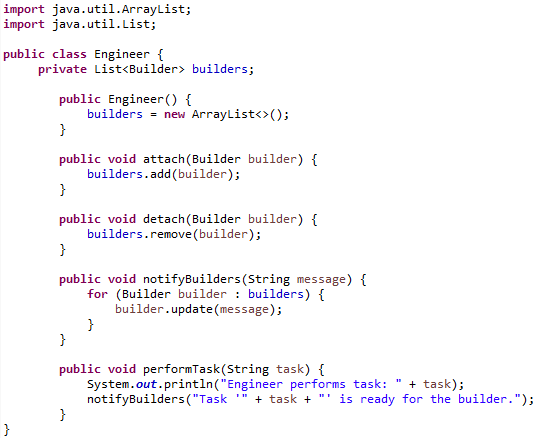
Description automatically generated with low confidenceA screen shot of a computer code

Description automatically generated with low confidenceA screenshot of a computer code

Description automatically generated with medium confidence

\*Observer:

It is a design pattern that enables you to specify a subscription mechanism to alert other objects to any events involving the object you're watching.

A picture containing text, font, screenshot, line

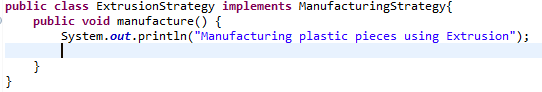
Description automatically generatedA screenshot of a computer code

Description automatically generated with low confidenceA picture containing text, screenshot, font

Description automatically generated

\*Strategy:

It is a design pattern that define and encapsulate a family of interchangeable algorithms, which enables them to be used interchangeably depending on the situation or requirements.

 A screen shot of a computer

Description automatically generated with low confidence A picture containing text, font, screenshot, line

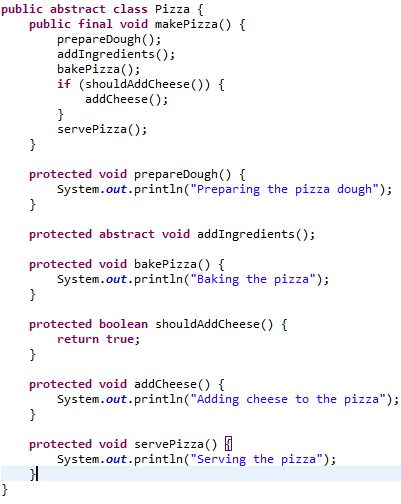
Description automatically generated A picture containing text, screenshot, font

Description automatically generated A picture containing text, screenshot, font, line

Description automatically generated

\*Template:

This is a design pattern that allows the superclass describes the basic framework of an algorithm, while allowing subclasses to customize certain stages without changing the overall structure.

 A picture containing text, screenshot, font, line

Description automatically generated A screenshot of a computer code

Description automatically generated with low confidence A picture containing text, screenshot, font

Description automatically generated

(1.3)

SOLID principles have a significant influence on the creation of object-oriented applications. They help in the development of modular, extensible, maintainable, and change-resistant codebases. Developers can write code that is simpler to read, understand, test, and change by following SOLID principles. By offering a common set of guidelines and practices, SOLID principles also encourage code reuse, separation of concerns, and better team collaboration. SOLID principles help build reliable, expandable, and flexible software systems overall.

(2.1)

A picture containing diagram, text, plan, technical drawing

Description automatically generated

(2.2)

\*collect requirements:

-Define the goals, parameters, and restrictions for the test.

-Determine the expectations of the target audience.

-The test levels and their order (for example, unit, integration, system, and acceptance) should be determined.

-List the different types of tests that will be run (such as functional, performance, and security).

-Define the specifications for the test environment.

\*Planning:

-Make a thorough test plan outlining the testing procedures, materials, and timetable.

-Determine the deadlines for the test deliverables.

-Define the entry and exit criteria for the test

-Determine risks and mitigation techniques.

-Select the metrics that will be gathered during testing.

\*the testing design:

-Determine and order test scenarios according to requirements.

-For each scenario, specify test data and test cases.

-Establish the test coverage requirements.

\*the testing:

-Run test cases, then document the results.

-Perform any necessary performance, security, and non-functional testing.

\*reporting:

-Create test reports that include an execution summary.

-Inform others about the condition, scope, and caliber of the application that is being tested.

-Describe practical insights for advancement.

-Report test coverage metrics, defect metrics, and other pertinent information.

(2.3)

A picture containing text, diagram, plan, technical drawing

Description automatically generated

(3.2)

In my code SOLID principles, clean coding practices, and programming patterns all work together to make the system more effective by enhancing its testability, maintainability, flexibility, extensibility, and maintainability. By improving the system's quality, readability, and modularity, these techniques make it simpler to grow in the future.

(3.3)

\*Unit testing is a method of testing where individual units get tested to insure that they are working as expected, unit testing can be automated using a test automation frame work like Junit.

Advantages:

-Early bug detection is made possible by unit tests, which help developers find bugs and mistakes as they are being developed. Issues can be found and fixed by testing individual pieces of code separately before they spread to more complicated and large-scale portions of the system.

-When a unit test fails, it gives a clear indication of the problem's location, making it simpler to find and resolve the issue. This simplifies and speeds up the debugging procedure.

-They can be programmed to automatically run after every code commit, making sure that no functionality is broken by new changes. Faster feedback loops are made possible by this, and more releases are made possible.

Disadvantages:

-It can take a lot of time and effort to write extensive unit tests in the beginning. For each unit, developers must create test cases.

-They might miss complex problems that result from the interaction of multiple components or issues with connection between units.

A picture containing text, font, screenshot, algebra

Description automatically generated

\*Release testing is carried out to guarantee that a software release or update works properly in the production environment before it is made accessible to end users.

Advantages:

-Release testing offers the chance to assess the software's scalability and performance in the context of actual usage. It aids in the detection of possible bottlenecks, resource limitations, or performance degradation under production-level loads.

-Release testing ensures that the new version of the software can run properly in a production environment and is ready for deployment. It makes certain that all essential setups, dependencies, and integrations are compatible with the target environment.

Disadvantages:

-To meet deployment deadlines, release testing is frequently done in a hurry. Due to a lack of time for thorough testing, there may be a chance that some problems are missed or that not all potential scenarios are sufficiently explored.

-Hardware, software licenses, and testing environments that resemble the production setup are frequently needed for release testing. Costs may be associated with setting up and maintaining these resources, especially for organizations with limited testing infrastructure or tight budgets.

A screen shot of a computer code

Description automatically generated with low confidence

\*Software testing at the level of systems verifies the integration of the entire system or application in comparison to predetermined requirements.

Advantages:  
-A thorough validation of the entire system, including interactions between various modules, subsystems, and external dependencies, is provided by system testing. It guarantees that all parts function in unison.

-Instead of concentrating on individual components, system testing enables evaluation of the behavior of the system as a whole. It confirms that the system's overall functionality, usability, security, and performance all meet the requirements and user expectations.

Disadvantages:

-After unit and integration testing of individual components, system testing is usually carried out. This means that until the system testing stage, individual component-specific flaws or problems might not be discovered. Debugging and problem-solving efforts may take longer and require more effort if defects are discovered later in the process.

-System testing depends on the availability of the entire system as well as any external dependencies, such as network services, databases, and APIs. It can be difficult to coordinate and manage these outside factors, which could cause delays or insufficient testing coverage.

A screen shot of a computer program

Description automatically generated with medium confidence

(4.1)

\*\*Unit testing:

An important technique in software development is unit testing, which means testing separate pieces of code or units of code in isolation. These units may be classes, modules, methods, or functions. Verifying the accuracy and dependability of these separate components is the main goal of unit testing.

Unit testing's main features and advantages are as follows:

-Unit tests are intended to be run independently, which means they test a single piece of code without relying on external dependencies or communication with other components.

-By catching bugs and design flaws early in the development cycle, it is easier and less expensive to fix them in the future.

-The code that is being tested by unit tests is documented as a result of the tests. Developers can learn about the unit's intended use and functionality by reading the tests.

Although unit testing is a crucial procedure, it shouldn't be the only testing strategy. To ensure thorough test coverage and overall system quality, it should be supplemented by other testing strategies like integration testing, system testing, and acceptance testing.

\*\*system testing:

System testing is an essential stage of the software development life cycle (SDLC) that concentrates on assessing the integrated, whole system to ensure compliance with predetermined requirements.

Key Aspects of System Testing:

-System testing that includes functional testing makes sure the system executes its intended functions as intended.

-During system testing, non-functional factors like performance, security, dependability, and usability are assessed. Performance testing evaluates how responsive, scalable, and resource-efficient the system is under various load conditions.

-System testing involves meticulous reporting and defect tracking. Issues are recorded in a defect tracking system when they are discovered, along with thorough information about the issue, how to reproduce it, and how it affects the system.

Overall, system testing is essential for guaranteeing the effectiveness, dependability, and quality of a software system. Early problem detection and correction reduces the impact on end users and enhances project success.

\*\*Release testing:

Release testing also referred to as release validation or release acceptance testing is a crucial stage in the lifecycle of software development. It aims to guarantee that a software update or release satisfies the necessary quality requirements before it is made available to users or clients.

Here are some important release testing factors and considerations:

-For release testing, the test environment should closely resemble the production environment. This includes any additional dependencies needed to accurately simulate real-world usage scenarios, as well as the hardware, software, network configurations, and other elements.

-In order to find software flaws, potential security breaches, or other weaknesses, release testing should also include security testing. This entails performing penetration tests, looking for common security flaws, and making sure that all applicable security standards and laws are being followed.

-Any issues or defects found during release testing should be carefully documented, tracked, and reported. This makes it possible to guarantee that the necessary corrective measures are taken and that the release complies with the necessary quality standards.

Overall, release testing is essential for making sure that new software releases are reliable, functional, and stable. It assists in lowering the possibility of introducing critical problems or regressions into production environments, enhancing user experience and reducing potential adverse effects on the company or organization.

For my application unit testing should be the most suitable testing method because it enabled me to test individual functions, and that is important to insure works properly and seeing that my code would not have a lot of functions in it I don’t see a problem in making a test for each. Also, my code follows the SOLID principals so using unit testing should be easy.

(4.3)

The differences between vendor-provided automated testing tools like Selenium and developer-provided automated testing tools like JUnit are as follows:

Focus and Goals:

Tools Provided by Developers: Tools provided by developers, like JUnit, are primarily geared toward unit testing. They are made to test isolated instances of a code block or component to validate its behavior and functionality.

Tools Provided by Vendors: Tools provided by vendors, such as Selenium, are more geared toward functional and comprehensive testing. The software's behavior is tested across a variety of components and interactions using simulations of actual user interactions.

The integration in the Development Process:

Tools Provided by Developers: Tools provided by developers are frequently integrated into the development process without any interruption. They can be integrated with build systems and pipelines, enabling tests to be run automatically in response to code changes or during the build procedure.

Tools supplied by the vendor: Tools supplied by the vendor can also be incorporated into the development process, but they might need more setup and configuration. Popular development and testing frameworks are frequently supported, but the degree of integration varies depending on the particular tool.

Flexibility and Customizability:

Tools Provided by Developers: Tools provided by developers are highly flexible and customizable. Developers can easily add functionality to the tool to meet their unique requirements and have control over the test structure and assertions. They have the ability to create tests that specifically target the codebase, giving them more precise control over testing scenarios.

Vendor-Provided Tools: In comparison to developer-provided tools, vendor-provided tools may be less customizable. They frequently offer a predefined set of capabilities, which might not fully address all use cases. They are made to appeal to a wider audience, but they typically offer a variety of built-in capabilities.

In conclusion, testing tools provided by developers typically focus on unit testing, and offer high customizability. An example of this is JUnit. Vendor-provided tools, such as Selenium, operate at a higher level of abstraction, are more user-friendly, and are more geared toward functional and end-to-end testing. The specific testing needs, the development environment, and the degree of control and customization required will determine which option is best.