Object-Oriented Design and Analysis Reference

by Andranik Khandanyan

Content

Preface

Object-oriented analysis and design overview

Object-oriented thinking

It is a critical skill for OOA/D to think in systems to be able to break down a whole into components with correctly assigned responsibilities, behavior, interaction.
Object-oriented thinking involves examining the problems or concepts at hand, breaking them down into component parts, and thinking of those as objects.

When translated to object-oriented modeling, object-oriented thinking involves representing key concepts through objects in your software.
In object-oriented thinking, often everything is considered an object, even if animate or live.

Analysis and design

Figure 1
Do the *right* thing and do the thing *right*

Analysis emphasizes an investigation of the problem and requirements, rather than a solution.

Analysis includes finding and describing the objects or concepts in the problem domain.

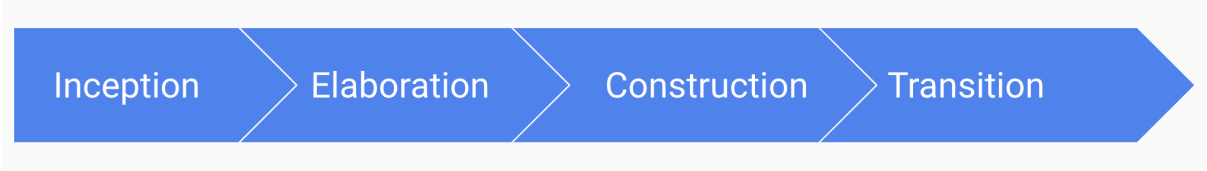
Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation.
Design includes defining software objects and how they collaborate to fulfill the requirements.

In other terms the analysis is doing the *right* thing and the design is doing things *right*.

Analysis and design in software development lifecycle

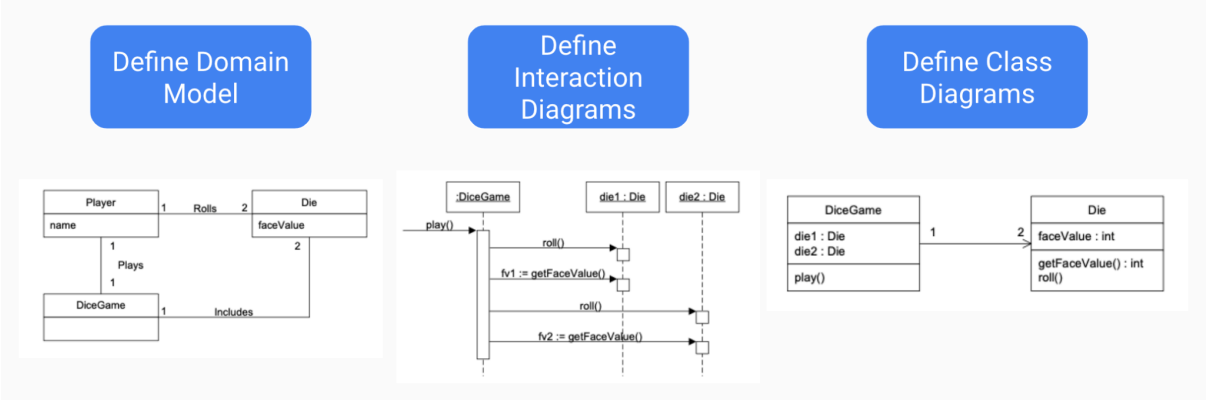
In the modern realm of software development, iterative methodologies, such as Agile, reign supreme. This methodology is a project management approach that entails breaking the project into phases and places emphasis on continuous collaboration and improvement. While terminology and phases may vary, we can distill common phases and actions into four key steps. (reference from the book)

Inception - approximate vision, business case, scope, vague estimates.
Elaboration - refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
Construction—iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
Transition - beta tests, deployment



Analysis and design are also iterative and continuous actions in this process, however some of those processes need to be addressed in early phases of the project development. Domain analysis and conceptual design mostly relies on the elaboration phase to construct the big picture. However, it is worth mentioning that it does not include the precise design of the entire scope.

Diagrams are useful tools to represent software aspects in a sample way without actual implementation. Diagrams, along with documentation are also permanent knowledge shared within interested stakeholders. Diagrams could be artifacts of object-oriented analysis and design. This can include domain model, business interaction and class diagrams.



Actions and techniques described here are also parts of the iterative process, whenever new features and changes are introduced in the software diagrams can and should be updated accordingly.

Requirements

Requirements are conditions or capabilities that must be implemented in a product, based on client or user request.

It is more than simply the client's vision. Instead, eliciting requirements involves actively probing the client vision, clarifying what may not have been told, and asking questions about issues the client may not have even considered.
It is also important to establish potential trade-offs the client may need to make in the solution.

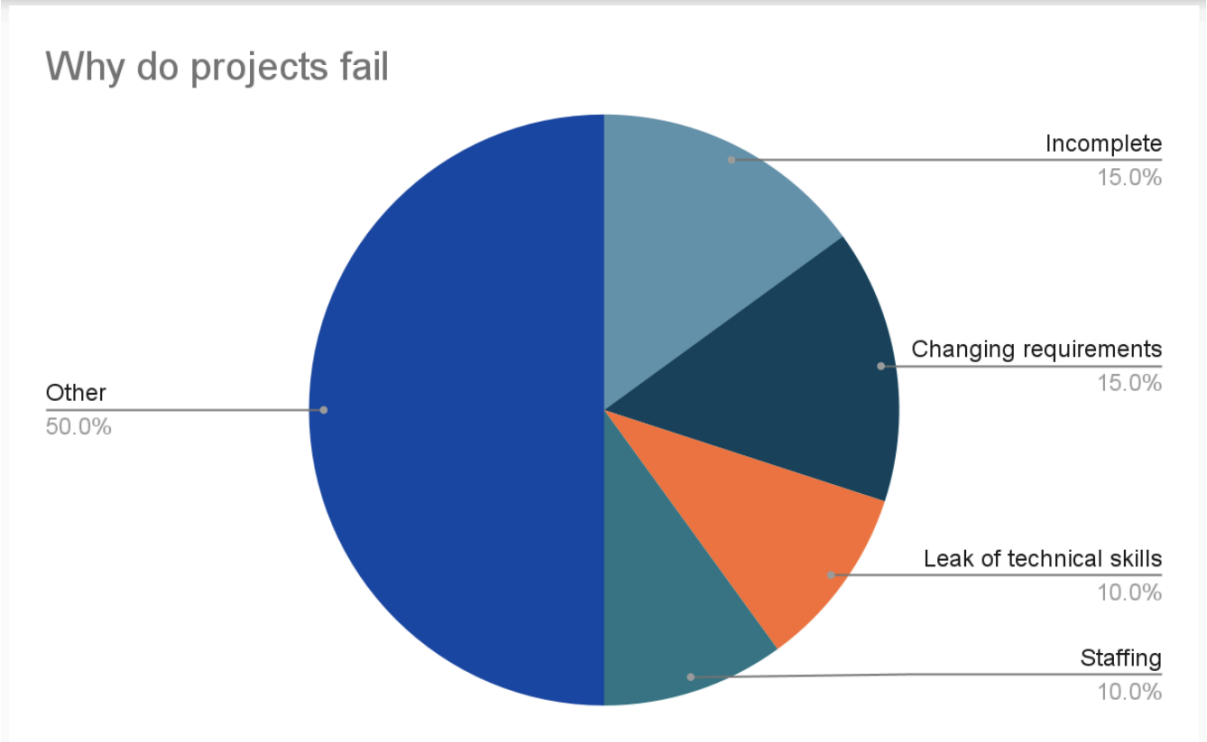


Figure 2.
Why do projects fail?
Please note that the data is collected from different sources and can't be 100% correct.

Types of requirements

Functional - what the system or application is expected to do. Features, capabilities, security.
Non-functional - how well the system or application does what it does. Performance, resource usage, and efficiency.

Non-functional requirements is a generic term and does not fully describe specifications often included in the scope.

FURPS+ model defines those requirements in detail.

Functional - what the system or application is expected to do.
Usability - human factors, help, documentation.
Reliability - frequency of failure, recoverability, predictability.
Performance - response times, throughput, accuracy, availability, resource usage.
Supportability - adaptability, maintainability, internationalization, configurability.
Other - Implementation, Interface, Operations, Legal.

Quality attributes

Some of these requirements are collectively called the **quality attributes**, quality requirements, or the "-ilities" of a system.
From the developer's perspective, during object-oriented analysis and design, outlined quality attributes need to be emphasized during design and implementation.

Maintainability	The ease at which your system is capable of undergoing changes. Systems will undergo changes throughout its life cycle, so a system should be able to accommodate these changes with ease.
Reusability	The extent in which functions or parts of your system can be used in another. Reusability helps reduce the cost of re-implementing something that has already been done.
Flexibility	How well a system can adapt to requirements change.

	A highly flexible system can adapt to future requirements changes in a timely and cost-efficient manner.
Modifiability	The ability of a system to cope with changes, incorporate new, or remove existing functionality. This is the most expensive design quality to achieve, so the cost of implementing changes must be balanced with this attribute.
Testability	How easy it is to demonstrate errors through executable tests. Systems should be tested as tests can be done quickly, easily, and do not require a user interface. This will help identify faults so that they might be fixed before system release.

Supplementary specification

The Supplementary Specification captures other requirements, information, and constraints not easily captured in the use cases, including quality attributes or requirements.

- reports
- hardware and software constraints (operating and networking systems, etc.)
- development constraints (for example, processes or development tools)
- other design and implementation constraints
- internationalization concerns (units, languages, etc.)
- documentation (user, installation, administration) and help
- licensing and other legal concerns
- standards (technical, safety, quality)
- operational concerns (for example, how do errors get handled, or how often to do backups?)

- domain or business rules
- information in domains of interest (for example, what is the entire cycle of credit payment handling?)

Trade-offs

Decisions may involve trade-offs in different quality attributes, such as performance, convenience, and security, and these attributes need to be balanced.
Teams must find the best balance between quality attributes often by evaluating which one is more important. Deadlines can also influence what is feasible to do within a certain time frame.

Examples of trade-offs are
Performance and maintainability – High performance code may be less clear and less modular, making it harder to maintain. Alternatively, extra code for backward compatibility may affect both performance and maintainability.
Performance and security – Extra overhead for high security may affect performance.

Organizing requirements

Organize requirements and iterations by risk, coverage, and criticality.
Risk - includes both technical complexity and other factors, such as uncertainty of effort or usability.
Coverage - implies that all major parts of the system are at least touched on in early iterations, perhaps a "wide and shallow" implementation across many components.
Criticality - refers to functions of high business value.

These criteria are used to rank work across iterations. Use cases or use case scenarios are ranked for implementation early iterations should implement high ranking scenarios. In addition, some requirements are expressed as high-level features unrelated to a particular use case, such as a logging service. These are also ranked.

Context and consequences

Context provides important information when deciding on the balance of qualities in design. For example, software that stores personal information, which the public can access, may have different security requirements than software that is only used by corporate employees.
Consequences - sometimes, choices made in software design have unintended consequences. For example, an idea that seems to work fine for a small amount of data may be impractical for large amounts of data.

Conceptual design

Conceptual designs are created with an initial set of requirements as a basis. The conceptual design recognizes appropriate components, connections, and responsibilities of the software product.

Conceptual design outlines the more high-level concepts of the final software product.
High-level components can be identified in this stage. Every component has a task it needs to perform, as known as its responsibility.

Conceptual designs are expressed or communicated through conceptual mock-ups. These are visual notations that provide initial thoughts for how requirements will be satisfied. Mock-ups for software involving user interfaces are often presented as wireframes, which are a kind of blueprint or basic visual representation of the product.
Visual

Use-case UML diagrams

A useful technique, while analyzing scenarios, are use cases. Writing use cases, stories of using a system, is an excellent technique to understand and describe requirements. Use cases are a mechanism to help keep it simple and understandable for all stakeholders. Informally, they are stories of using a system to meet goals.
In the Unified Modeling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system. To build one, you'll use a set of

specialized symbols and connectors. An effective use case diagram can help your team discuss and represent:

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system

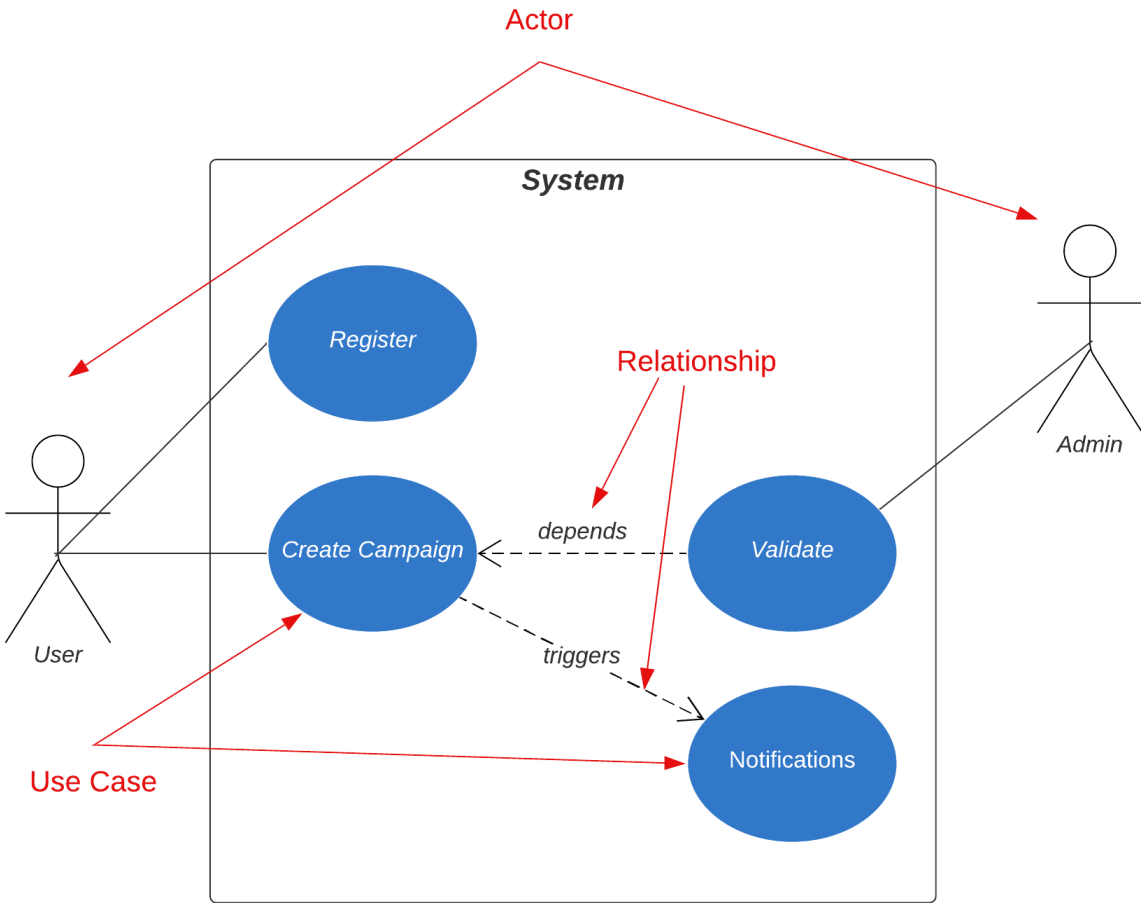


Figure 4.
UML Use-Case diagram

Class Responsibility Collaborator

CRC cards are used to record, organize, and refine the components of system design. CRC cards are designed with three sections: the class name at the top of the card, the responsibilities of the class on the left side of the card, and the collaborators on the right side of the card.

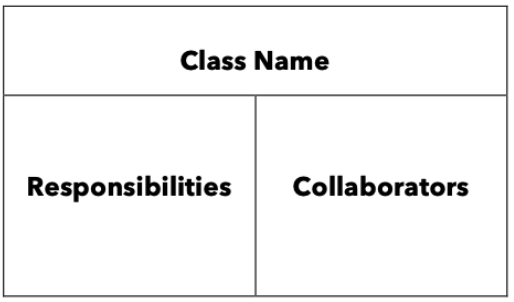


Figure 5.
CRC card.

Domain Modeling

Domain model concepts

A domain model is a conceptual representation of a problem domain.
Key components and characteristics

- Entities
- Attributes
- Relationships
- Methods or Behaviors
- Constraints
- Abstraction

Entities are objects, concepts, or things that exist within the problem domain. They are typically represented as classes in the domain model. Each class represents a specific entity and includes attributes to describe the characteristics of that entity.
Attributes are properties or characteristics of entities. They provide information about the entities and are represented as variables or fields within the corresponding class.
Relationships: Relationships define how entities are related to each other within the domain. There are different types of relationships, such as one-to-one, one-to-many, and many-to-many. These relationships are represented in the domain model to show how entities interact or associate with each other.
Methods or Behaviors: While the primary focus of a domain model is on the structure of entities and their attributes and relationships, it can also include high-level descriptions of behaviors or methods that entities exhibit. However in early stages of analysis and design only responsibilities can be defined and assigned and detailed behavioral modeling is often addressed in later stages of OOAD.
Domain models may also capture constraints or rules that apply to entities and their relationships. These constraints can be expressed as business rules, validation rules, or other domain-specific requirements.
Abstraction: Domain models abstract away implementation details and focus on the essential elements of the problem domain. They provide a

high-level, easy-to-understand view of the domain, making it easier to communicate and reason about the domain's complexities.

Strategies to identify entities

Identifying entities sometimes can be done intuitively, but mostly and preferably it needs to be analyzed properly so major actors can be identified in early stages. The simplest approach is to identify **noun** phrases.
This simple scenario clearly exposes vehicle, parking lot and parking spot entities.
The **parking lot** should have the capacity to park 200 **vehicles**.
Two different types of **parking spots** are compact and large.
Three types of **vehicles** should be allowed to park in the parking lot, which are as follows…

However entities can be identified by looking for categories of entities. Such categories are

- Other Computer or Electro-Mechanical Systems
- Abstract Noun Concepts
- Organizations
- Events
- Processes (often not represented as a concept, but may be)
- Rules and Policies
- Catalogs
- Records of Finance, Work, Contracts, Legal Matters
- Financial Instruments and Services
- Manuals, Documents, Reference Papers, Books

Steps to design domain model

1. List the candidate conceptual classes using the Conceptual Class Category. List and noun phrase identification techniques related to the current requirements under consideration.
2. Draw them in a domain model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory.

4. Add the attributes necessary to fulfill the information requirements.

What are so many types of classes

To keep things clear, this book will use class-related terms as follows, which is consistent with the UML and the UP:
• Conceptual class - real-world concept or thing. A conceptual or essential perspective.
• Software class - a class representing a specification or implementation perspective of a software component, regardless of the process or method.
• Design class - a member of the Design Model. It is a synonym for software class, but for some reason I wish to emphasize that it is a class in the Design Model. A design class to be either a specification or implementation perspective, as desired by the modeler.
• Implementation class — a class implemented in an object-oriented language such as Java.
• Class — as in the UML, the general term representing either a real-world thing (a conceptual class) or software thing (a software class).

UML Class Diagrams

Classes of the system can be modeled using Unified Modeling Language (UML).
Since classes are the building block of objects, class diagrams are the building blocks of UML. The various components in a class diagram can represent the classes that will actually be programmed, the main objects, or the interactions between classes and objects.

Class diagrams offer a number of benefits for any organization. Use UML class diagrams to:

- Illustrate data models for information systems, no matter how simple or complex.
- Better understand the general overview of the schematics of an application.
- Visually express any specific needs of a system and disseminate that information throughout the business.

- Create detailed charts that highlight any specific code needed to be programmed and implemented to the described structure.
- Provide an implementation-independent description of types used in a system that are later passed between its components.

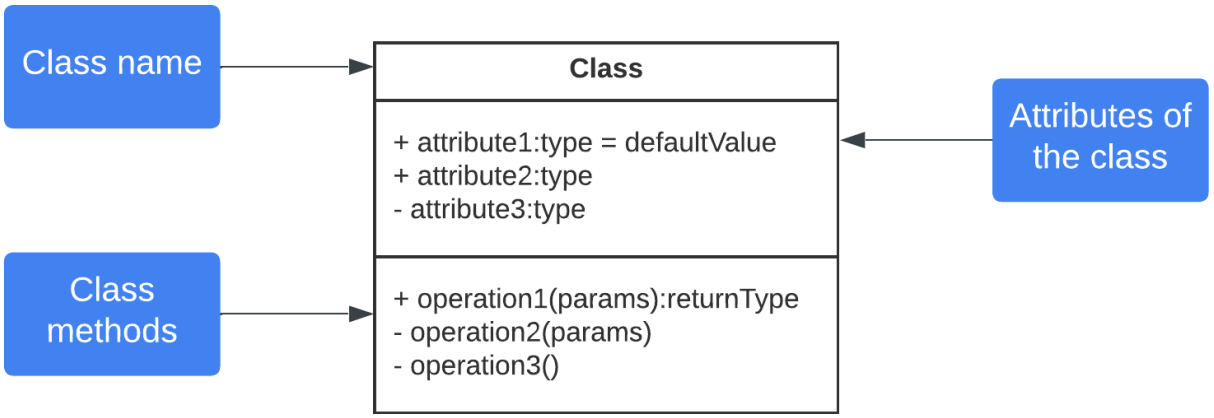


Figure 6.
Sections of class in UML class diagram

Member access modifiers

All classes have different access levels depending on the access modifier (visibility). Here are the access levels with their corresponding symbols:

- Public (+)
- Private (-)
- Protected (#)
- Package (~)
- Derived (/)
- Static (underlined)

Best practices

- Constant communication and feedback is key to creating the right solution that meets client needs and works within any restrictions that may exist.

- During the analysis of functional requirements, attention to detail within the relevant context is highly beneficial.
-

The base of design principles

To create an object-oriented program, you must examine the major design principles of such programs. Four of these major principles are: abstraction, encapsulation, decomposition, and generalization.

Abstraction

Abstraction breaks a concept down into a simplified description that ignores unimportant details and emphasizes the essentials needed for the concept, within some context.

An abstraction should follow the rule of **least astonishment**. This rule suggests that essential attributes and behaviors should be captured with no surprises and no definitions that fall beyond its scope.

The essential characteristics of an abstraction can be understood in two ways: through basic attributes and through basic behaviors or responsibilities.

Differentiate what from how.

Encapsulation

- The ability to *bundle* attribute values (or data) and behaviors (or functions) that manipulate those values, into a self-contained object.
- The ability to *expose* certain data and functions of that object, which can be accessed from other objects, usually through an interface.
- The ability to *restrict* access to certain data and functions to only within the object.

An object's data should only contain what is relevant for that object.
One of the ideas of encapsulation is restricting access to certain data and functions to only within an object, this naturally links encapsulation to data **integrity and the security** of sensitive information.

As the ability to **expose** data is separate from the **bundle** of attributes itself, this means that the implementation of attributes and methods can change, but the accessible interface of a class can remain the same.

Encapsulation achieves an abstraction barrier through **black box thinking** where the internal workings of a class are not relevant to the outside world.

Decomposition

The design principle of decomposition takes a whole thing and divides it into different parts. It also does the reverse, and takes separate parts with different functionalities, and combines them to form a whole.
The general rule for decomposition is to look at the different responsibilities of a whole and evaluate how the whole can be separated into parts that each have a specific responsibility.

There are three types of relationships in decomposition, which define the interaction between the whole and the parts:

Association

indicates a loose relationship between two objects, which may interact with each other for some time. One object does not belong to another, and they may have numbers that are not tied to each other.

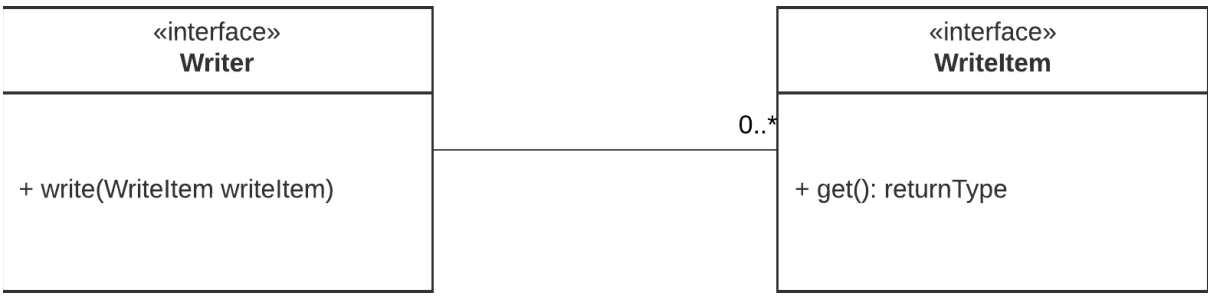


Figure 7.
Association in UML class diagram.

The “zero dot dot star” (0...*) on the **right** side of the line shows that a **Writer** object is associated with **zero or more Hotel** objects.

Aggregation

Is a “*has-a*”, weak relationship where a whole has parts that belong to it.

Although parts can belong to wholes, they can also exist independently.

Aggregation can be represented in UML class diagrams with the symbol of an empty diamond as below:

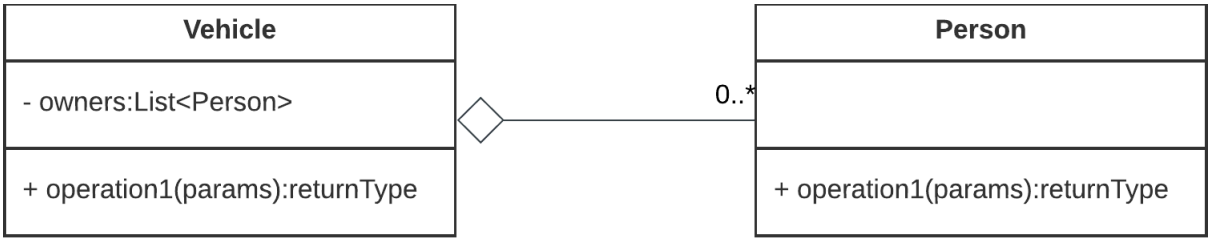


Figure 8.
Aggregation in UML class diagram.

The “zero dot dot star” (0...*) on the **right** side of the line shows that an **Vehicle** object aggregates **zero or more Person** objects.

Composition

is a “*has-a*”, strong relationship where a whole has parts that belong to it.

A whole cannot exist without its parts, and if the whole is destroyed, then the parts are destroyed too. In this relationship, you can typically only access the parts through its whole.

Composition can be represented with a filled-in diamond using UML class diagrams, as below:

Figure 9.
Composition in UML class diagram.

Recognizing decomposition

- A is a physical/logical part of B (Water reservoir/Coffee Machine, license/Insurance company)
- A is physically/logically contained in/on B (Item/Stock, Vehicle/Contract)
- A is recorded in B
- A is a description for B
- A uses or manages B (Pilot / Airplane)
- A communicates with B
- A is next to B
- A is owned by B
- A is an event related to B

Generalisation

The design principle of generalization takes repeated, common, or shared characteristics between two or more classes and factors them out into another class, so that code can be reused, and the characteristics can be inherited by subclasses.
In object-oriented modeling, generalization is a main design principle, but beyond creating a method that can be applied to different data, object-oriented modeling achieves generalization by classes through

inheritance. In generalization, common, or shared characteristics between two or more classes taken and factored out into another class.

In standard terminology, a parent class is known as a **superclass** and a child class is called a **subclass**.

When to define subclass

- 100% of the conceptual superclass's definition should be applicable to the sub-class.
- Is-A rule
- The subclass has additional attributes of interest.
- The subclass has additional associations of interest.
- The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses, in ways that are of interest.

When to define superclass

- The potential conceptual subclasses represent variations of a similar concept.
- The subclasses will conform to the 100% and Is-a rules.
- All subclasses have the same attribute which can be factored out and expressed in the superclass.
- All subclasses have the same association which can be factored out and related to the superclass.

Liskov substitution principle

A principle that states that a subclass can replace a superclass, if and only if the subclass does not change the functionality of the superclass.

In simple words if a test passes for a superclass, it should pass for subtype as well, when superclass replaced by subtype.

- Code smell to understand this principle is a method that accepts a parameter of a certain type or interface, but then checks the actual type of the parameter passed in and deals with different types differently.

- If you do something to one type then the result should match doing the same thing on a type that is substitutable.

Abstract classes and interfaces

Oftentimes there is a decision needed to be made whether to use abstract class or interface. Programming languages can provide different capabilities of abstract classes and interfaces or even those can be just concepts in language. By understanding the concepts of abstract classes and interfaces it is easier to make the decision.

Abstract classes

- Abstract classes are used to define a common base class for related classes that share some common behavior and have some differences.
- They provide a way to define methods with a default implementation that can be inherited by subclasses. That is a concrete behavior.
- Can define and use state. Abstract classes can have instance variables (fields) holding the state of the object.
- Abstract classes are useful when you want to provide a common base implementation and define a contract (via abstract methods) for subclasses.

Interfaces

- Interfaces define a contract that classes can implement. It defines what behavior it provides and the concrete behavior.
- Interfaces are useful when you want to define a contract that multiple classes can adhere to, regardless of their inheritance hierarchy.
- They are also used to achieve a form of polymorphism called "interface-based polymorphism," where objects of different classes that implement the same interface can be treated uniformly.

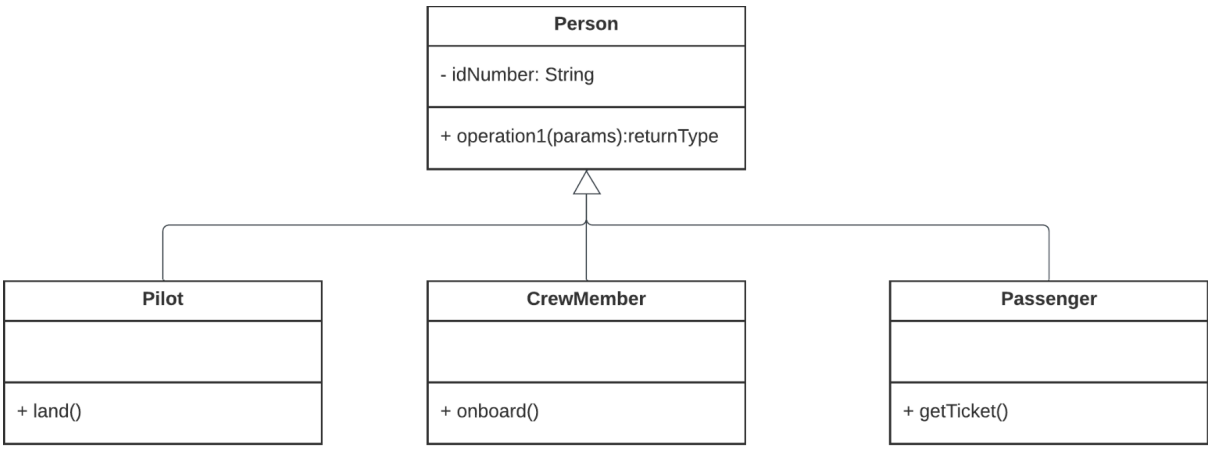
Prefer composition over inheritance

Composition over inheritance is a principle in object-oriented programming that suggests prioritizing the use of composition to achieve polymorphic behavior and code reuse, instead of relying heavily on class inheritance.

- Inheritance is tightly coupled whereas composition is loosely coupled.
- There is no access control in inheritance whereas access can be restricted in composition. We expose all the superclass methods to the other classes having access to subclass.

An illustrative example of this principle can be reviewed by examining airline domain. There are 3 possible roles in the system: pilot, crew member and passenger. All of them are persons with their data, such as an ID number. Firstly we can think of inheriting *Pilot*, *Crew member* and *Passenger* from the *Person* class. As soon as one of the crew members wants to travel as a passenger, challenges of this design start to show up. It is not possible to present this new behavior without recreating a person, which restricts flexibility of the system.

When designed with composition, *Pilot*, *Crew member* and *Passenger* can refer to Person, so anytime whenever it is needed a role for a Person can be created and removed without affecting Person lifecycle. Also, when accessed by *Role* interface it is possible to expose Person's behavior and attributes selectively, which is not possible in case of inheritance.



In case if a **CrewMember** needs to travel as a **Passenger**

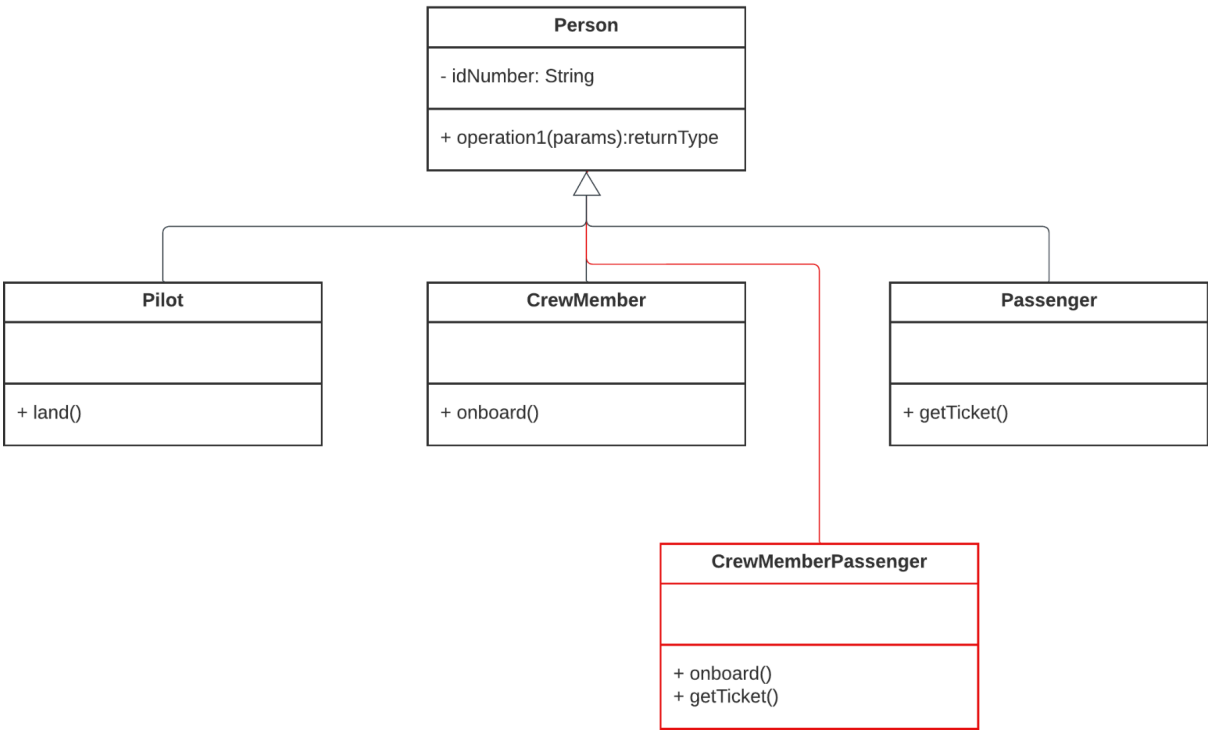


Figure 10.
Airline system design with Inheritance.

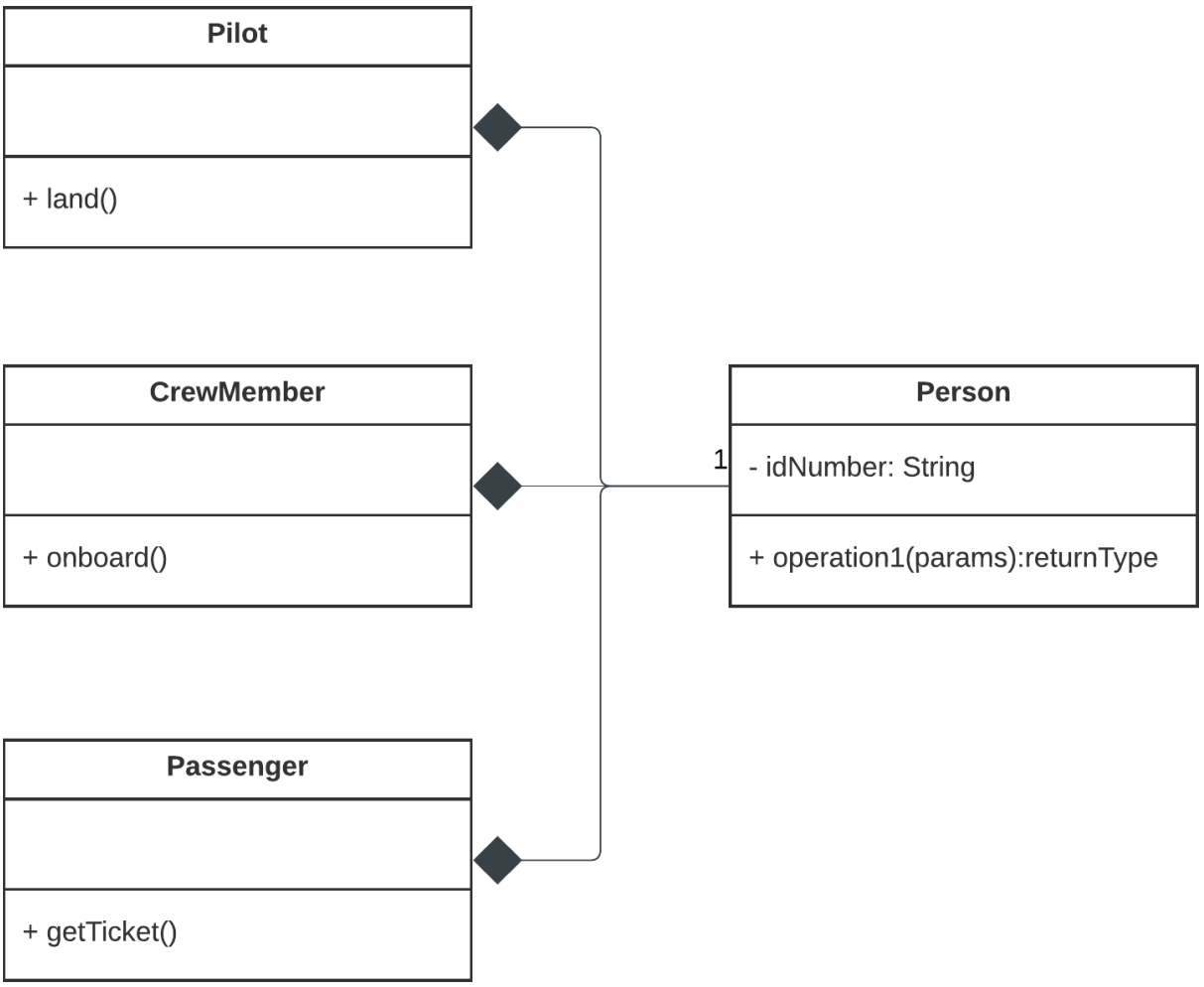


Figure 11.
Airline system design with Composition.

D.R.Y.

Both methods and inheritance exemplify the generalization design principle through the D.R.Y. or “Don’t Repeat Yourself” rule. Methods and inheritance allow developers to reuse code, resulting in less code and repetition overall.
Please don’t take each code duplication as a dogmatic red flag. Sometimes it can make sense to have code duplication to achieve low coupled components.

Design and analysis practices

Evaluating design complexity

It is important to keep modules(classes and methods) simple when you are programming. If your design complexity exceeds what developers can mentally handle, then bugs will occur more often. To help control this, there must be a way of evaluating your design complexity.

- If the system has a bad design, then modules can only connect to other specific modules and nothing else.
- A good design allows any modules to connect together without much trouble.
- A good design, modules are compatible with one another and can therefore be easily connected and re-used.

Coupling

Coupling focuses on complexity between a module and other modules.

- If a module is too reliant on other modules, then it is "**tightly coupled**" to others (bad design).
- If a module finds it easy to connect to other modules through well-defined interfaces, it is "**loosely coupled**" to others (good design).

Coupling can be measured using degree, ease and flexibility

- **Degree** is the number of connections between the module and others. The degree should be small for coupling. For example, a module should connect to others through only a few parameters or narrow interfaces. This would be a small degree, and coupling would be loose.
- **Ease** is how obvious are the connections between the module and others. Connections should be easy to make without needing to understand the implementations of other modules, for coupling purposes.
- **Flexibility** indicates how interchangeable the other modules are for this module. Other modules should be easily replaceable for something better in the future, for coupling purposes.

Tight coupling smells

- A module connects to other modules through a great number of parameters or interfaces
- A module can only be connected to specific other modules and cannot be interchanged
- Harder to understand in isolation.
- Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

There is no absolute measure of when coupling is too high. What is important is that a developer can gauge the current degree of coupling, and assess if increasing it will lead to problems. In general, classes that are inherently very generic in nature, and with a high probability for reuse, should have especially low coupling.

Cohesion

Cohesion focuses on complexity within a module, and represents the clarity of the responsibilities of a module. Like complexity, cohesion can work between two extremes: **high cohesion** and **low cohesion**.

A module that performs one task and nothing else, or that has a clear purpose, has high cohesion. A good design has high cohesion. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion.

If a module encapsulates more than one purpose, if an encapsulation has to be broken to understand a method, or if the module has an unclear purpose, it has low cohesion. A bad design has low cohesion.

Balancing Coupling and Cohesion

It is important to balance between low coupling and high cohesion in system design. Both are necessary for a good design. However, in complex systems, complexity can be distributed between the modules or within the modules. For example, as modules are simplified to achieve high cohesion, they may need to depend more on other modules, thus increasing coupling. On the other hand, as connections between modules are simplified to

achieve low coupling, the modules may need to take on more responsibilities, thus lowering cohesion.

Design behavior with UML sequence diagram

Sequence diagrams are a popular dynamic modeling solution in UML because they specifically focus on lifelines, or the processes and objects that live simultaneously, and the messages exchanged between them to perform a function before the lifeline ends.
A sequence diagram is a type of interaction diagram because it describes how and in what order a group of objects works together. These diagrams are used by software developers and business professionals to understand requirements for a new system or to document an existing process.
Sequence diagrams are sometimes known as event diagrams or event scenarios.

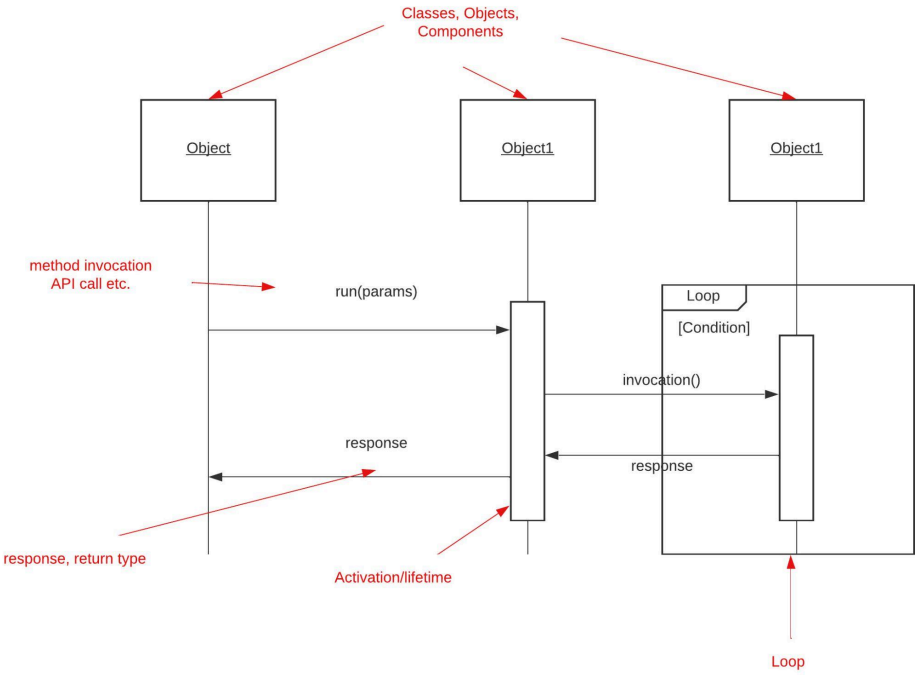


Figure 12.
UML sequence diagram

Sequence diagrams can be useful references for businesses and other organizations. Draw a sequence diagram to:

- Represent the details of a UML use case.
- Model the logic of a sophisticated procedure, function, or operation.

- See how objects and components interact with each other to complete a process.
- Plan and understand the detailed functionality of an existing or future scenario.

UML state diagram

A state diagram, sometimes known as a state machine diagram, is a type of behavioral diagram in the UML that shows transitions between various objects.

A state machine is any device that stores the status of an object at a given time and can change status or cause other actions based on the input it receives. States refer to the different combinations of information that an object can hold, not how the object behaves. In order to understand the different states of an object, you might want to visualize all of the possible states and show how an object gets to each state, and you can do so with a UML state diagram.

Each state diagram typically begins with a dark circle that indicates the initial state and ends with a bordered circle that denotes the final state. However, despite having clear start and end points, state diagrams are not necessarily the best tool for capturing an overall progression of events. Rather, they illustrate specific kinds of behavior—in particular, shifts from one state to another.

State diagrams mainly depict states and transitions. States are represented with rectangles with rounded corners that are labeled with the name of the state. Transitions are marked with arrows that flow from one state to another, showing how the states change.

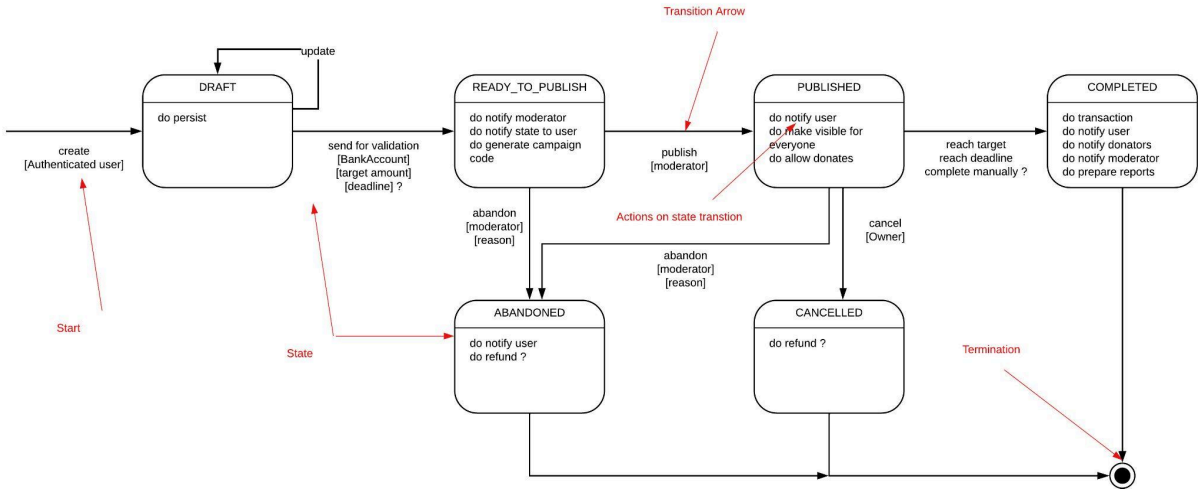


Figure 13.
UML state diagram.

Like most UML diagrams, state diagrams have several uses. The main applications are as follows:

- Depicting event-driven objects in a reactive system.
- Illustrating use case scenarios in a business context.
- Describing how an object moves through various states within its lifetime.
- Showing the overall behavior of a state machine or the behavior of a related set of state machines.

Showcase: Vehicle insurance toolkit

Showcase: Batch processing library

Design principles reference

Single responsibility (S)

A class should have only one reason to change.
In other words, a class should have a single responsibility or a single job. When a class is responsible for multiple, unrelated tasks, changes to one task can inadvertently affect the others, leading to code that is difficult to maintain and understand. The Single Responsibility Principle advises breaking such a class into smaller, more focused classes, each with a clear and distinct responsibility. This leads to more maintainable and flexible software designs.

Open/closed principle (O)

A module, class, entity should be open for extension but closed for modification.
When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with “bad” design. The program becomes fragile, rigid, unpredictable and not reusable. The open-closed principle attacks this in a very straightforward way. It says that you should design modules that never change. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.
In terms of inheritance, this principle encourages the use of abstract base classes or interfaces that can be extended by subclasses without modifying the existing code.

Liskov substitution principle (L)

A principle that states that a subclass can replace a superclass, if and only if the subclass does not change the functionality of the superclass.

Code smell to understand this principle is a method that accepts a parameter of a certain type or interface, but then checks the actual type of the parameter passed in and deals with different types differently.

If you do something to one type then the result should match doing the same thing on a type that is substitutable.

Interface segregation (I)

Clients should not be forced to depend upon interfaces that they do not use. The Interface Segregation Principle (ISP) states that a client should not be exposed to methods it doesn't need.

Code Smells:

- A Bulky Interface
- Unused Dependencies (parameters)
- Methods Throwing Exceptions (Unsupported operation)
- Superclass uses lower level knowledge

Dependency inversion (D)

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules.

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

GRASP principles

Information Expert

This principle suggests assigning the responsibility for a particular task or behavior to the class or object that possesses the most information or data required to fulfill that responsibility. In other words, delegate responsibilities to the class with the most relevant data or knowledge.

Creator

The Creator principle advises that a class should be responsible for creating instances of other classes if it aggregates or contains them. In simpler terms, a class that uses another class should be responsible for creating instances of that other class.

- Assign class B the responsibility to create an instance of class A if one or more of the following is true:
- B aggregates, contains, records A objects.
- B has the initializing data that will be passed to A when it is created (thus B is an Expert with respect to creating A).
- B is a creator of A objects.
- If more than one option applies, prefer a class B which aggregates or contains class A.

Controller

The Controller principle recommends assigning the responsibility for handling system events, such as user inputs or requests, to specific controller objects. These controller objects act as intermediaries between the user interface and the domain model, facilitating the flow of information.

Pure fabrication

In some cases, you may need to introduce artificial classes or objects that don't represent real-world concepts but help in achieving low coupling and high cohesion. This is the Pure Fabrication pattern, which allows for better design when necessary.

Separation of concerns

Separation of concerns is about keeping the different concerns in your design separate. When software is designed, different concerns should be addressed in different portions of the software.
A concern is a very general notion: it is anything that matters in providing a solution to a problem.
Separation of concerns is a key idea that underlies object-oriented modeling and programming. When addressing concerns separately, more cohesive classes are created and the design principles of abstraction, encapsulation, decomposition, and generalization are enforced

Conceptual integrity

Conceptual integrity is a concept related to creating consistent software. Conceptual integrity entails making decisions about the design and implementation of a software system, so even if multiple people work on it, it would seem cohesive and consistent as if only one mind was guiding the work.

There are multiple ways to achieve conceptual integrity. These include:

- communication
- code reviews
- using certain design principles and programming constructs
- having a well-defined design or architecture underlying the
- software
- unifying concepts
- having a small core group that accepts each commit to the code base.

Principle of least knowledge

Composing object principle

The coupling objects principle is a means of circumventing this problem, by providing a means for a high amount of code reuse without using inheritance. The principle states that classes should achieve code reuse through aggregation rather than inheritance. Aggregation and delegation offer less coupling than inheritance, since the composed classes do not share attributes or implementations of behaviors, and are more independent of each other. This means they have an “arms length” relationship.

Composing objects also allows the dynamic change of behaviors of objects at run time. A new overall combination of behavior can be built by composing objects. Inheritance, on the other hand, requires behaviors of classes to be defined during compile time, so they cannot change while the program is running.

Book Resources

References

Glossary