**OOP HW18 Software Engineering Principles**

**DATE: 01th february 2021**

1 ALMACHE LITARDO ANDERSON MOISES

The Solid Principles

SOLID is an acronym:

* **S: Single Responsibility Principle**  
  "An object must have a single responsibility"
* **O: Open / Closed Principle**  
   "software entities should be open to extension and closed to modification"
* **L: Liskov Substitution Principle**  
  "objects in a program must be replaceable by instances of their subtypes without altering the correctness of the program"
* **I: Interface Segregation Principle**  
   "many client-specific interfaces are preferable to one general-purpose interface"
* **D: Dependency Inversion Principle**  
   "we must depend on abstractions and not on concretions"

Modularity, abstraction and encapsulation

These mentioned methods help to improve the functionality, development and understanding of a code, among them it can be highlighted that:

* **Modularity**

Is the ability to ignore the details of the parts in order to focus attention on a higher level of a problem.

* **Abstraction**

Happens when something is wrapped in a protective layer or shielded from anything that might harm it.

* **Encapsulation**

Is the act of packing or protecting data or attributes with methods.

2 ALTAMIRANO BENALCAZAR CRISTHIAN ALEXANDER

**The solid principles**

**S**ingle Responsibility Principle

**O**pen/closed principle

**L**iskov substitution principle

**I**nterface segregation principle

**D**ependency inversion principle

**Single Responsibility Principle**

Each class is responsible for its role in the functionality of the system, and the responsibilities of a class must be encapsulated by the class and its attributes must be consistent with the responsibilities of the class. This principle is related to cohesion, which means that responsibilities are aligned and focused on a single purpose.

**Open/closed principle**

Software entities should be open for extension but closed to modification, open if it is still available for extension, closed if it is available for use by other classes, and therefore should not be modified. If the programmer makes modifications, the principle is broken.

**Liskov substitution principle**

Product a base class, with one virtual method, called save, whose intent is to save an object to a file. When implementing a specialization, Widget, of some Product: The implementation of save in Widget adheres to the purpose of save in Product don’t have it do some unrelated thing, like re-load the object from a file instead doesn’t rely on stronger assumptions than programmatically implement any special conditions that required and handle exceptions appropriately Ensure that accomplishes, as minimum, all that is supposed to accomplish. If it is supposed to save the x attribute to a file, then must do at least this much.

**Interface segregation principle**

An interface is a ”window” or “portal” into the functionality of a component An interface doesn’t have to declare all of the possible public methods of a component; a component can have many interfaces, an interface represents public methods of a component Java does support interfaces directly Core Idea: No client (user of a component) should be forced to depend on methods that it does not use The public methods of a component can be grouped by purpose or responsibility as captured and declared in interfaces, or abstract classes

**Dependency inversion principle**

Organize the system in layers, some reusable libraries will be abstract or detail oriented. Reusable layer components do not depend on detail layers. In addition, they depend on the abstraction that the detailed components implement. The abstractions must not depend on the details and the implementation must depend on the abstractions

3 ALVAREZ RAMIREZ MICHELLE ESTEFANIA

**THE SOLID PRINCIPLES** allow to have a quality software:

* **S**ingle Responsibility Principle.- Each class should be responsible for only one part of the system's functionality.
* **O**pen/Closed Principle.- A class system is open to extension and closed to modification,if Public methods (the abstractions) are declared using interfaces or Implementations and concrete classes inherit the public method declarations from the interfaces,
* **L**iskov Substitution Principle.- if S is a specialization of T, an S object must be able to do everything any T object can do
* **I**nterface Segregation Principle.- An interface is a ”window” or “portal” into the functionality of a component. An interface represents public methods of a component.
* **D**ependency Inversion Principle.- Organize the system into layers. Abstractions should not depend on details. High-level modules should not depend on low-level modules

**MULTI-PARADIGM SOFTWARE ENGINEERING**

A program must be built in the less time and must be efficiency, security and understandability.

**BEST PRACTICES, PATTERNS, AND IDIOMS.** They are techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level.

You can integrate 3 core principles (rule for creating software with certain desirable characteristics):

* **Modularity.-** It exists in a software system when it is comprised of loosely coupled and cohesive components. Can improve understandability, testability maintainability, reliability, security, extensibility, and reuse.
* **Abstraction.-** The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context
* **Encapsulation.-** The private implementation details of a component must be insulated so they cannot be accessed or modified by other components. Doing so will lead to better testability, maintainability, and reliability. It will also help with a clear separation of concerns and avoid accidental coupling.

Modularity deals with the decomposition of system into components, whereas abstraction and encapsulation deal with individual components. Therefore, modularity cannot be subsumed by either the other two. But, Abstraction and encapsulation might be considered duals of each other.

4 ANDRADE CARATE ALAN DAMIAN

**INTRODUCTION TO SOLID PRINCIPLES**

SOLID refers to a MNEMONIC ACRONYM that contains 5 principles, which are: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle.

**Design problem**

You have been hired to build a maze generator.

The program should produce mazes containing N x M quarter squares, where N and M are the user-specified width and height of the maze.

The program must be able to print the mazes using ASCII characters or draw them on an image.

**Open – closed principle**

• Core Ideas:

• Software entities (e.g., classes, generics) should be open for extension but closed to

· modification

• Original definitions:

• A class is open if it is still available for extension

• A class is closed if it is available for use by other class, and therefore should not be modified

• Revised definitions:

• A system of classes is open for extension and closed for modification, if

• Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes in Java

• Ways to achieve the open/closed principle

**Inheritance**

• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual clases

**Aggregation**

• Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

**Parameterization**

• Use a generic to capture a template solution and instantiate it with the specific data types

• Following the Open/Closed Principle can help developers

• Reduce complexity by reducing coupling (dependencies among components)

• Increase extensibility

**Interfaces, abstract classes, pure virtual classes**

• Review: Inheritance allows a specialization (a derived class) to re-use the generalization’s:

• Data members

• Method declarations

• Method definitions (i.e., their implementations)

**Following the liskov substitution principle**

• Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file.

**Dependency inversión principle**

• How to apply the Dependency Inversion Principle

• Abstractions should not depend on details

• High-level modules should not depend on low-level modules

• Both low-level and high-level modules should depend on abstractions

• “Program to the abstraction”

• Following the Dependency Inversion Principle helps Developers

• Increase Reusability

• Increase Maintainability

**UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION AS A STEP TOWARD FOUNDATIONAL MULTI PARADIGM SOFTWARE ENGINEERING PRINCIPLES**

**Software engeniering goals**

• Software engineers aim to build quality products on time and within budget

• Some Desirable Qualities:

• understandability

• testability

• maintainability

• efficiency

• reliability

• security

• extensibility

• openness

• interoperability

• reusability

**Common paradigms**

• Object orientation (OO)

• Aspect orientation (AO)

• Functional programming (FP)

• Logic programming (LP)

• Genetic programming (GP)

• Structured program (SP)

**Problema blackground**

• Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees

• However, the concepts and definitions of these principles differ across

• paradigms

• In some cases, there are conflicting definitions within the same paradigm

• There are also many other proposed principles that overlap and break up

• the ideas differently.

**Contributions of the initial paper**

• The purpose of this paper is not to reinvent the concepts of modularity, abstraction, or encapsulation

• Instead, it is to stimulate discussion about the unification of existing ideas.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

• Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level.

• Patterns exemplify principles, by providing proven solutions to recurring problems in specific contexts.

• Idioms are techniques or solution for expressing a certain algorithm or

• data structure in a specific programming language, in a way that is consistent with certain principles.

**OBSERVATIONS RELATIVE TO MODULARITY**

• Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways

• Two concepts that can help achieve this desirable characteristic:

• Coupling: the degree to which components depend on each other

• Cohesion: the degree to which the properties of a component relate to the component’s primary responsibility

**Paradigm independet definition for Modulation**

Practices and Criteria:

• Localization of design decisions

• Low Coupling

• High Cohesion

• Modular Reasoning

**OBSERVATIONS RELATIVE TO ABSTRACTION**

• Creating good software abstractions is hard

• Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts

• Abbott et al. described an abstraction as the “reification and conceptualization of a distinction”

**Paradigm independet definition for Abstraction**

• •Practices and Criteria:

• Meaningful labels and identifiers

• Context-aware labels and identifiers

• Abstraction completeness

• Abstraction sufficiency

**Paradigm independet definition for Encapsulation**

• Practices and Criteria:

• Conceptual barriers

• Programmatic barriers

• Usage barriers

**Non-redundancy and complimentary criterion N1**

• Modularity deals with the decomposition of system into components, whereas abstraction and encapsulation deal with individual components

• Therefore, modularity cannot be subsumed by either the other two

• And, conversely

• Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different

**Non-redundancy and complimentary criterion N2**

We show satisfaction of the second criteria, namely that developers and choose to follow each principle independent, with an example consisting of four functional-identical code snippets

· A simple program snippet with good Modularity, Abstraction, Encapsulation

· Same as #1, but with just good Modularity

· Same as #1, but with just good Abstraction

· Same as #1, but with just good Encapsulation

5 ANDRANGO ESPINOSA ALEX PAUL

**S**ingle responsibility principle

**O**pen/close principle

**L**iskov substitution principle

**I**nterface segregation principle

**D**ependency inversion principle

**Single responsibility principle**

· Every class is responsible for a single part of the system.

· A class´s encapsulation by the class.

· A class´s narrowed alignment.

**Open/close principle**

· Class open if it is still available for extension.

· Class close for use by other classes.

**Liskov substitution principle**

· If S is a specialization of T, then an S object can be used wherever a t object is required.

**Interface segregation principle**

· No client is forced to depend on methods that it does not use.

· The public methods of a component can be grouped by purpose or responsibility as captured in a declaration in interfaces, or abstract classes.

**Dependency inversion principle**

· Organize the system, into layers.

· Components from the abstract layers should not depend on components from detail layers.

· Abstraction does not depend on details.

· Implementation details depend on abstractions.

6 ARROBA SOLORZANO CRISTIAN ALEXANDER

**INTRODUCTION TO THE SOLID PRINCIPLES**

**OVERVIEW OF THE SOLID PRINCIPLES**

SOLID is a mnemonic acronym for five principles

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

**DESIGN PROBLEM**

You’ve been contracted to build a Maze Generator

• The program must produce mazes that

• Contain N x M square rooms, where N and M are the user-specified width and height of the maze

The program must be able to print the mazes using ASCII characters or draw them in an image.

**OPEN/CLOSED PRINCIPLE**

• Core Ideas:

• Software entities (e.g., classes, generics) should be open for extension but closed to

modification

• Original definitions:

• A class is open if it is still available for extension

• A class is closed if it is available for use by other class, and therefore should not be modified

• Revised definitions:

• A system of classes is open for extension and closed for modification, if

• Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes (in Java)

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

• **Review:** Inheritance allows a specialization (a derived class) to re-use the generalization’s (a base class’s):

• Data members

• Method declarations

• Method definitions (i.e., their implementations)

**OPEN/CLOSED PRINCIPLE**

• Ways to achieve the open/closed principle

**Inheritance**

• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual clases

**Aggregation**

• Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

**Parameterization**

• Use a generic to capture a template solution and instantiate it with the specific data types

• Following the Open/Closed Principle can help developers

• Reduce complexity by reducing coupling (dependencies among components)

• Increase extensibility

**FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE**

• Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file.

**DEPENDENCY INVERSION PRINCIPLE**

• How to apply the Dependency Inversion Principle

• Abstractions should not depend on details

• High-level modules should not depend on low-level modules

• Both low-level and high-level modules should depend on abstractions

• “Program to the abstraction”

• Following the Dependency Inversion Principle helps Developers

• Increase Reusability

• Increase Maintainability

**UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION AS A STEP TOWARD FOUNDATIONAL MULTI-PARADIGM SOFTWARE ENGINEERING PRINCIPLES**

**SOFTWARE ENGINEERING GOALS**

Software engineers aim to build quality products on time and within budget

Some Desirable Qualities:

• understandability

• testability

• maintainability

• efficiency

• reliability

• security

• extensibility

• openness

• interoperability

• reusability

**COMMON PARADIGMS**

• Object orientation (OO)

• Aspect orientation (AO)

• Functional programming (FP)

• Logic programming (LP)

• Genetic programming (GP)

• Structured program (SP)

**PROBLEM BACKGROUND**

• Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees

• However, the concepts and definitions of these principles differ across

paradigms

• In some cases, there are conflicting definitions within the same paradigm

• There are also many other proposed principles that overlap and break up

the ideas differently.

**CONTRIBUTIONS OF THIS INITIAL PAPER**

• The purpose of this paper is NOT to reinvent the concepts of modularity, abstraction, or encapsulation

• Instead, it is to stimulate discussion about the unification of existing ideas.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

• Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level.

• Patterns exemplify principles, by providing proven solutions to recurring problems in specific contexts.

• Idioms are techniques or solution for expressing a certain algorithm or

data structure in a specific programming language, in a way that is consistent with certain principles.

**OBSERVATIONS RELATIVE TO MODULARITY**

• Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways

• Two concepts that can help achieve this desirable characteristic:

• Coupling: the degree to which components depend on each other

• Cohesion: the degree to which the properties of a component relate to the component’s primary responsibility

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

Practices and Criteria:

• Localization of design decisions

• Low Coupling

• High Cohesion

• Modular Reasoning

**OBSERVATIONS RELATIVE TO ABSTRACTION**

• Creating good software abstractions is hard

• Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts

• Abbott et al. described an abstraction as the “reification and conceptualization of a distinction”

**PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION**

•Practices and Criteria:

• Meaningful labels and identifiers

• Context-aware labels and identifiers

• Abstraction completeness

• Abstraction sufficiency

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION**

•Practices and Criteria:

• Conceptual barriers

• Programmatic barriers

• Usage barriers

**THE NON-REDUNDANCY AND COMPLIMENTARY NATURE OF THE MAE PRINCIPLES**

**• Criteria:**

**1.** No general principle can be a special case of or subsumed by another principle or combination of principles

**2.** Developers should be able to choose to follow one principle but not the others.

**NON-REDUNDANCY AND COMPLIMENTARY – CRITERION #1**

• Modularity deals with the decomposition of system into components, whereas abstraction and encapsulation deal with individual components

• Therefore, modularity cannot be subsumed by either the other two

• And, conversely

• Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different

**NON-REDUNDANCY AND COMPLIMENTARY – CRITERION #2**

• We show satisfaction of the second criteria, namely that developers and choose to follow each principle independent, with an example consisting of four functional-identical code snippets

**1.** A simple program snippet with good Modularity, Abstraction, Encapsulation

**2.** Same as #1, but with just good Modularity

**3.** Same as #1, but with just good Abstraction

**4.** Same as #1, but with just good Encapsulation

7 ASUMAZA GUALOTO DYLAN ALEXANDER

**INTRODUCTION TO THE SOLID PRINCIPLES**

**OPEN/CLOSED PRINCIPLE**

· Software entities (e.g., classes, generics) should be open for extension but closed to modification

· A class is open if it is still available for extension

· A class is closed if it is available for use by other class, and therefore should not be modified

· A system of classes is open for extension and closed for modification, if

· Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes (in Java)

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

Review: Inheritance allows a specialization (a derived class) to re-use the generalization’s (a base class’s):

· Data members

· Method declarations

· Method definitions (i.e., their implementations)

**OPEN/CLOSED PRINCIPLE**

· Ways to achieve the open/closed principle

**Inheritance**

· Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual clases

**Aggregation**

· Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

**Parameterization**

· Use a generic to capture a template solution and instantiate it with the specific data types

· Following the Open/Closed Principle can help developers

· Reduce complexity by reducing coupling (dependencies among components)

**DEPENDENCY INVERSION PRINCIPLE**

· How to apply the Dependency Inversion Principle

· Abstractions should not depend on details

· High-level modules should not depend on low-level modules

· Both low-level and high-level modules should depend on abstractions

· “Program to the abstraction”

· Following the Dependency Inversion Principle helps Developers

· Increase Reusability

· Increase Maintainability

**SOFTWARE ENGINEERING GOALS**

Software engineers aim to build quality products on time and within budget Some Desirable Qualities:

understandability

· testability

· maintainability

· efficiency

· eliability

· security

· extensibility

· openness

· interoperability • reusability

**PROBLEM BACKGROUND**

Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees However, the concepts and definitions of these principles differ across

paradigms.In some cases, there are conflicting definitions within the same paradigm • There are also many other proposed principles that overlap and break up the ideas differently.

**CONTRIBUTIONS OF THIS INITIAL PAPER**

The purpose of this paper is NOT to reinvent the concepts of modularity, abstraction, or encapsulation.Instead, it is to stimulate discussion about the unification of existing ideas.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level. Patterns exemplify principles, by providing proven solutions to recurring problems in specific contexts. Idioms are techniques or solution for expressing a certain algorithm or

data structure in a specific programming language, in a way that is consistent with certain principles.

**OBSERVATIONS RELATIVE TO MODULARITY**

Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways.Two concepts that can help achieve this desirable characteristic:

· Coupling: the degree to which components depend on each other

· Cohesion: the degree to which the properties of a component relate to the component’s primary responsibility

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

Practices and Criteria:

· Localization of design decisions

· Low Coupling

· High Cohesion

· Modular Reasoning

8 BRAVO RODRIGUEZ KATHERIN DAYANNE

**INTRODUCTION TO THE SOLID PRINCIPLES**

**SOLID IS A MNEMONIC ACRONYM FOR FIVE PRINCIPLES**

•Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

**DESIGN PROBLEM**

You’ve been contracted to build a Maze Generator

• The program must produce mazes that

• Contain N x M square rooms, where N and M are the

user-specified width and height of the maze

The program must be able to print the mazes using ASCII characters or draw them in an image.

**OPEN/CLOSED PRINCIPLE**

• Core Ideas:

• Software entities (e.g., classes, generics) should be open for extension but closed to

modification

• Original definitions:

• A class is open if it is still available for extension

• A class is closed if it is available for use by other class, and therefore should not be modified

• Revised definitions:

• A system of classes is open for extension and closed for modification, if

• Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes (in Java)

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

• **Review:** Inheritance allows a specialization (a derived class) to re-use the generalization’s (a base class’s):

• Data members

• Method declarations

• Method definitions (i.e., their implementations)

**OPEN/CLOSED PRINCIPLE**

• Ways to achieve the open/closed principle

**Inheritance**

• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual clases

**Aggregation**

• Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

**Parameterization**

• Use a generic to capture a template solution and instantiate it with the specific data types

• Following the Open/Closed Principle can help developers

• Reduce complexity by reducing coupling (dependencies among components)

• Increase extensibility

**FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE**

• Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file.

**DEPENDENCY INVERSION PRINCIPLE**

• How to apply the Dependency Inversion Principle

• Abstractions should not depend on details

• High-level modules should not depend on low-level modules

• Both low-level and high-level modules should depend on abstractions

• “Program to the abstraction”

• Following the Dependency Inversion Principle helps Developers

• Increase Reusability

• Increase Maintainability

**UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION AS A STEP TOWARD FOUNDATIONAL MULTI-PARADIGM SOFTWARE ENGINEERING PRINCIPLES**

**SOFTWARE ENGINEERING GOALS**

Software engineers aim to build quality products on time and within budget

Some Desirable Qualities:

• understandability

• testability

• maintainability

• efficiency

• reliability

• security

• extensibility

• openness

• interoperability

• reusability

**COMMON PARADIGMS**

• Object orientation (OO)

• Aspect orientation (AO)

• Functional programming (FP)

• Logic programming (LP)

• Genetic programming (GP)

• Structured program (SP)

**PROBLEM BACKGROUND**

• Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees

• However, the concepts and definitions of these principles differ across

paradigms

• In some cases, there are conflicting definitions within the same paradigm

• There are also many other proposed principles that overlap and break up

the ideas differently.

**CONTRIBUTIONS OF THIS INITIAL PAPER**

• The purpose of this paper is NOT to reinvent the concepts of modularity, abstraction, or encapsulation

• Instead, it is to stimulate discussion about the unification of existing ideas.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

• Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level.

• Patterns exemplify principles, by providing proven solutions to recurring problems in specific contexts.

• Idioms are techniques or solution for expressing a certain algorithm or

data structure in a specific programming language, in a way that is consistent with certain principles.

**OBSERVATIONS RELATIVE TO MODULARITY**

• Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways

• Two concepts that can help achieve this desirable characteristic:

• Coupling: the degree to which components depend on each other

• Cohesion: the degree to which the properties of a component relate to the component’s primary responsibility

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

Practices and Criteria:

• Localization of design decisions

• Low Coupling

• High Cohesion

• Modular Reasoning

**OBSERVATIONS RELATIVE TO ABSTRACTION**

• Creating good software abstractions is hard

• Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts

• Abbott et al. described an abstraction as the “reification and conceptualization of a distinction”

**PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION**

•Practices and Criteria:

• Meaningful labels and identifiers

• Context-aware labels and identifiers

• Abstraction completeness

• Abstraction sufficiency

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION**

•Practices and Criteria:

• Conceptual barriers

• Programmatic barriers

• Usage barriers

**THE NON-REDUNDANCY AND COMPLIMENTARY NATURE OF THE MAE PRINCIPLES**

**• Criteria:**

**1.** No general principle can be a special case of or subsumed by another principle or combination of principles

**2.** Developers should be able to choose to follow one principle but not the others.

**NON-REDUNDANCY AND COMPLIMENTARY – CRITERION #1**

• Modularity deals with the decomposition of system into components, whereas abstraction and encapsulation deal with individual components

• Therefore, modularity cannot be subsumed by either the other two

• And, conversely

• Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different

**NON-REDUNDANCY AND COMPLIMENTARY – CRITERION #2**

• We show satisfaction of the second criteria, namely that developers and choose to follow each principle independent, with an example consisting of four functional-identical code snippets

**1.** A simple program snippet with good Modularity, Abstraction, Encapsulation

**2.** Same as #1, but with just good Modularity

**3.** Same as #1, but with just good Abstraction

**4.** Same as #1, but with just good Encapsulation

9 BRAVO VILLALOBOS CHRISTIAN DAVID

10 BUSTILLOS MONTENEGRO PABLO SEBASTIAN

**MODULARITY, ABSTRACTION, AND ENCAPSULATION AS A STEP TOWARD FOUNDATIONAL MULTI-PARADIGM SOFTWARE ENGINEERINGSOFTWARE ENGINEERING GOALS**

Software engineers aim to build quality products on time and within budget

**Some Desirable Qualities:**

• Understandability

• Testability

• Maintainability

• Efficiency

• Reliability

• Security

• Extensibility

• Openness

• Interoperability

• Reusability

**THREE CORE PRINCIPLES**

Modularity:

Modularity exists in a software system when it is comprised of loosely coupled and cohesive components that isolate each significant or changeable design decision in one component and ensure that related ideas are as close as possible. Modularity can improve understandability, testability maintainability, reliability, security, extensibility, and reuse. It can also help with collaboration during the software development process by outlining loosely coupled work units

Abstraction:

For each component, there is an explicit and clear declaration of the component’s accessible features or functionality. Depending on the paradigm and programming language, this declaration may be part of the source code, meta data, or documentation. The exposed features and functionality should be no more and no less than what other components may need or depend on. Adherence to the abstraction principle can improve understandability, testability, maintainability, and reusability. It can also allow developers to follow modularity more effectively, because it will bring to light weakness with localization of design decisions, unnecessary coupling, and low cohesion.

Encapsulation:

Ensure that the private implementation details of a component are insulated so they cannot be accessed or modified by other components. Doing so will lead to better testability, maintainability, and reliability. It will also help with a clear separation of concerns and avoid accidental coupling.

**OVERVIEW OF THE SOLID PRINCIPLES**

• SOLID is a mnemonic acronym for five principles

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

• Some argue that these are the ”first five” principles

• That claim has not been justified or widely accepted

• However, whether they are the “first five” principles is not very important

• Following these principles can help ensure quality software, primarily from a developers’ perspective

11 CADENA ROMAN BENJAMIN ABEL

**INTRODUCTION TO THE SOLID PRINCIPLES**

Dependency Inversion Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation

Single Responsibility Principle

**OVERVIEW**

Some argue that these are the ”first five” principles

• That claim has not been justified or widely accepted

• However, whether they are the “first five” principles is not very important

• Following these principles can help ensure quality software, primarily from a developers’ perspective

**DESING PROBLEM**

• You’ve been contracted to build a Maze Generator

• The program must produce mazes

• The program must be able to print the mazes using ASCII character or draw them in an image

**SINGLE RESPONSIBILITY PRINCIPLE**

• Core ideas

• This principle is vey closely related to the more general principle of Cohesion

• This principle is also related to the principles of:

• Localization of design decisions

• Encapsulation

• Following this principle can help

**OPEN/CLOSED PRINCIPLE**

• Core Ideas:

• Software entities

• Original definitions:

• A class is open if it is still available for extension

• Revised definitions:

• A system of classes is open for extension and closed for modification

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

• Review: Inheritance allows a specialization

• An interface is like a base class, but only allows for Method declarations

• In UML, the names of abstract and pure virtual classes and methods are written in italics

• An abstract and a pure virtual class (C++) may include data members and some method implementations

• The modern Open/Closed Principle encourages developers to

• use interfaces, abstract classes, and pure virtual classes to declare public data members

**OPEN/CLOSED PRINCIPLE**

• Ways to achieve the open/closed principle

• Inheritance

• Aggregation

• Parameterization

• Following the Open/Closed Principle can help developers

**LISKOV SUBSTITUTION PRINCIPLE**

• if S is a specialization of T, then an S object can be used wherever a T object is required,

e.g.,

• Strong behavioral subtyping – an S object must be able to do everything any T object

• Following the Liskov Substitution Principle can help developers

• Increase Reuse

• Increase Extensibility

• Increase Maintainability

**FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE**

• Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file

• When implementing a specialization, Widget, of some Product, ensure that

**INTERFACE SEGREGATION PRINCIPLE**

• An interface is a ”window” or “portal” into the functionality of a component

• An interface represents public methods of a component

• Following the Interface Segregation Principle, when used with other principles, can help Developers

• Reduce complexity by increasing Cohesion and reducing Coupling

• Increase extensibility

• Increase reuse

• Increate maintainability

**DEPENDENCY INVERSION PRINCIPLE**

• Organize the system into layers: some layers, like reusable libraries or frameworks will be more abstract or policy-setting layer, others will be detail oriented

• How to apply the Dependency Inversion Principle

• Abstractions should not depend on details

• High-level modules should not depend on low-level modules

• Following the Dependency Inversion Principle helps Developers

• Increase Reusability

• Increase Maintainability

**SOFTWARE ENGINEERING GOALS**

• Software engineers aim to build quality products on time and within budget

• Some Desirable Qualities:

• understandability

• testability

• maintainability

• efficiency

• reliability

• security

• extensibility

• openness

• interoperability

• reusability

**COMMON PARADIGMS**

• Object orientation (OO) • Aspect orientation (AO) • Functional programming (FP) • Logic programming (LP) • Genetic programming (GP) • Structured program (SP)

**PROBLEM BACKGROUND**

• Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees

• However, the concepts and definitions of these principles differ across paradigms

**CORE PROBLEM**

• There are no general, unifying definitions, especially for multiparadigm software development

**CONTRIBUTIONS OF THIS INITIAL PAPER**

• Clarify the purpose of software-engineering (SE) principles, in general, and distinguish them from “best practices”, idioms, and patterns

• Propose drafts of paradigm-independent definitions for the MAE principles

**CONTRIBUTIONS OF THIS INITIAL PAPER**

• The purpose of this paper is NOT to reinvent the concepts of modularity, abstraction, or encapsulation

• Instead, it is to stimulate discussion about the unification of existing ideas

**PURPOSE OF PRINCIPLES**

• According to the Merriam-Webster and Oxford dictionaries

**SOFTWARE ENGINEERING PRINCIPLES**

1) a truth or proposition that supports reasoning about the desirable characteristics of a software system

2) a rule for creating software with certain desirable characteristics

3) an aspect of software design that imparts certain desirable characteristics

**BEST PRACTICES, PATTERNS, AND IDIOMS**

• Best practices are procedures or techniques that help developers adhere to principles

• Patterns exemplify principles

• Idioms are techniques or solution for expressing a certain algorithm or data structure in a specific programming language

**OBSERVATIONS RELATIVE TO MODULARITY**

• Earlier and seminal work by David Parnas (1972, plus later works)

• Design decisions need to be identified and implemented in one place. We call this Localization of Design Decisions

• Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways

• Low Coupling and High Cohesion were first explored for SP, but have been heavily discussed for OO and AO

• The first principle of Robert Martin’s SOLID principles is a restatement of High Cohesion

• Martin Fowler and other experts believe that modularity will avoid common code smells, like Long Method, Large Class, Long Parameter List, Feature Envy, and Inappropriate Intimacy

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

Practices and Criteria:

• Localization of design decisions

• Low Coupling

• High Cohesion

• Modular Reasoning

Tradeoffs:

• Localization of design decisions and high cohesion can lead to many fine grain components, which is good for testability, extensibility, and reuse, but may hinder readability

• However, the lack modularity will compromise desirable characteristics

• Adherence or violation of the modularity principles typically affects multiple components

Paradigm Notes:

• For LP, the components are primarily predicates, rules, and facts.

• Developers must ensure every predicate represents a single idea or responsibility. In other words, every predicate should be highly cohesive.

**OBSERVATIONS RELATIVE TO ABSTRACTION**

• From a process perspective, abstraction is the act of bringing certain details to the forefront while suppressing all others

• Creating good software abstractions is hard

• Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts

**TWO COMMON PROBLEMS WITH ABSTRACTION**

• Leaky abstraction – other components end up relying on details not explicitly stated in the abstraction

• External characteristics are not defined as such

**PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION**

Practices and Criteria:

• Meaningful labels and identifiers

• Context-aware labels and identifiers

• Abstraction completeness

• Abstraction sufficiency

**OBSERVATIONS RELATIVE TO ENCAPSULATION**

• Three categories of existing definition for encapsulation:

• The bundling of data with operations

• The hiding decisions behind logical barriers

• The organization of components to minimize ripple effects

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION**

Practices and Criteria:

• Conceptual barriers

• Programmatic barriers

• Usage barriers

**SUMMARY**

• Clarified the purpose of software-engineering principles

• Proposed a template for documenting principles

• Proposed drafts of paradigm-independent definitions for the MAE principles

**FUTURE WORK**

• Setup concrete empirical studies to explore those questions

• Explore metrics for systematically assessing quality in mixed-paradigm software system

• Investigations into other design principles beyond MAE

12 CAISATOA RAMIREZ SEBASTIAN BERNARDO

***Software Engineering aim to:***

Build quality products on time and within budget.

Some Desirable qualities: understandability, testability, maintainability, efficiency, reliability, security, extensibility, openness, interoperability, reusability.

***Common paradigms***

Object orientation, aspect orientation, functional programming, logic programming, genetic programming, structured program.

· Today, many languages and development environments support multi-paradigm software development.

· There are no general, unifying definitions, especially for multi-paradigm software development.

· The principles are hard to teach.

· The proliferation and variation of principle definitions causes confusion among developers.

***Software Engineering Principle:***

1. A truth or proposition that supports reasoning about the desirable characteristics of a software system.

2. A rule for creating software with certain desirable characteristics.

3. An aspect of software design that imparts certain desirable characteristics.

***SOLID PRINCIPLES***

SOLID is a mnemonic acronym for five principles:

• Single Responsibility Principle.

• Open/Closed Principle.

• Liskov Substitution Principle.

• Interface Segregation Principle.

• Dependency Inversion Principle.

***Original definitions:***

• A class is open if it is still available for extension.

• A class is closed.

***Core Ideas:***

• Software entities (e.g., classes, generics) should be open for extension but closed to modification.

***Inheritance:***

• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes.

• The public methods of one class can be grouped into multiple abstractions.

• Each abstraction should focus on a single purpose, as per the Single Responsibility Principle.

• Have concrete classes inherit from these abstractions.

• Java does not support multiple inheritance, so a class can have multiple base classes.

***Aggregation:***

• Encapsulate behaviors in sub-part objects and allow those sub-part objects to change dynamically.

• This technique has been embodied in something called the strategy pattern – more on this later.

***Parameterization:***

• Use a generic to capture a template solution and instantiate it with the specific data types.

13 CALDERON MERCHAN ANDY JOSUE

**INTRODUCTION TO THE SOLID PRINCIPLES**

**OVERVIEW OF THE SOLID PRINCIPLES**

SOLID stands for five principles and is a mnemonic acronym for them.

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

Following these guidelines can help to ensure high-quality software, especially from the perspective of developers.

**DESIGN PROBLEM**

You've been hired to construct a Maze Generator.

The software must create mazes with N x M square rooms, where N and M are the maze's width and height, respectively.

The mazes must be able to be printed as ASCII characters or drawn in an image by the application.

**SINGLE RESPONSIBILITY PRINCIPLE**

Each class should be in charge of a specific aspect of the system's operation.

The responsibility of a class should be totally encapsulated within the class.

The idea of cohesion states that any component's responsibilities (methods, classes, sub-systems, etc.) should be firmly coordinated and focused on a single goal.

This concept is also linked to the concepts of:

- Encapsulation

- Localization of design

- Increase Reuse and Maintainability

**OPEN/CLOSED PRINCIPLE**

Software entities (for example, classes and generics) should be extensible but not modifiable.

- If you can still extend a class, it's considered open.

- If a class is open for usage by other classes, it is closed and should not be updated.

Interfaces or abstract classes in Java are used to declare public methods such as the abstractions.

The public method declarations from interfaces, abstract classes, and pure virtual classes are inherited by implementations or concrete classes.

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

The ability to especialize a derivado class in order to repurpose the generalization a base class is provided by herencia. An interface is similar to a base class, except that it only enables method definitions. It prohibits the declaration or definition of data members.

Abstract and pure virtual classes and methods are written in italics in UML.

Interfaces are supported by Java.

Data members and some method implementations may be included in an abstract and a pure virtual class (C++), at least one method declaration without an implementation is common.

**OPEN/CLOSED PRINCIPLE**

The current Open/Closed Principle encourages developers to specify public data elements using interfaces, abstract classes, and pure virtual classes.

**- Inheritance**

Public methods should be placed in their own abstractions, such as interfaces, abstract classes, or pure virtual classes.

A single class's public methods can be divided into several abstractions, every abstraction should have a single goal in mind.

**- Aggregation**

Sub-part objects can be used to encapsulate activities and allow them to alter dynamically.

- **Parameterization**

Assist developers in decreasing complexity by eliminating coupling.

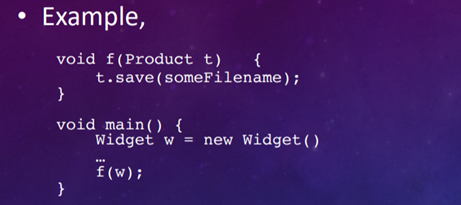
**LISKOV SUBSTITUTION PRINCIPLE**

Allow Product to be a base class with only one virtual method, save, that saves an object to a file.

- Widget.save() does not make the same assumptions as Product.save()

- Any special circumstances required by Widget.save() should be implemented programmatically, and errors should be handled accordingly.

As a minimum, make sure Widget.save() does everything Product.save() is meant to do.



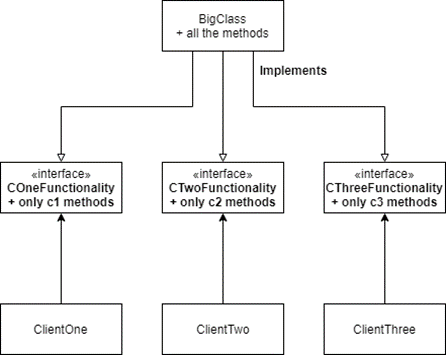
**INTERFACE SEGREGATION PRINCIPLE**

An interface is a "window" or "portal" into a component's functionality, represents public methods of a component.

Interfaces are directly supported by Java.

No client (component user) should be made to rely on methods it doesn't use. Purp can be used to group a component's public methods.

**Example:**



**DEPENDENCY INVERSION PRINCIPLE**

Layer the system: some levels, such as reusable libraries or frameworks, will be more abstract or policy-setting, components from the abstract layers should not rely on components from the concrete layers.

Details should not be relied upon in abstracts.

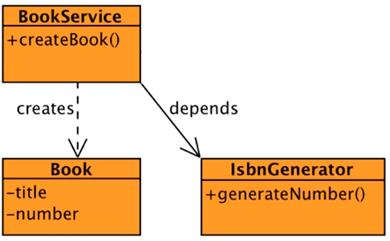
Details of implementation should be based on abstractions.

- Low-level modules should not be relied upon by high-level modules and should depend on abstractions.

The Dependency Inversion Principle is beneficial to developers.

- Increase Reusability

- Increase Maintainability

**UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION AS A STEP TOWARD FOUNDATIONAL MULTI-PARADIGM SOFTWARE ENGINEERING PRINCIPLES**

**SOFTWARE ENGINEERING GOALS**

The goal of software engineers is to deliver high-quality products on time and on budget.

Desirable Qualities:

- understandability

- security

- testability

**THREE CORE PRINCIPLES**

- Modularity

- Abstraction

- Encapsulation

**COMMON PARADIGMS**

Multi-paradigm software development is now supported by several languages and development environments (MPSD).

**PROBLEM BACKGROUND**

All of these common software development approaches prioritize modularity, abstraction, and encapsulation to varying degrees. Different paradigms have different notions and definitions for these principles.

**CORE PROBLEM**

Particularly for multiparadigm software development, there are no common, unifying definitions.

- Programmers, especially in multi-paradigm software development, frequently do not understand the essential ideas and hence do not profit from their guidance. The lack of unified definitions makes tool support difficult.

**CONTRIBUTIONS OF THIS INITIAL PAPER**

- The goal of this study is not to reintroduce modularity, abstraction, or encapsulation.

Allows the definition of a principle to go beyond simply expressing the underlying concepts also provides a foundation for evaluating adherence to the principle.

Propose drafts of paradigm-independent MAE principles definitions.

**PURPOSE OF PRINCIPLES**

1. a truth or proposition that can be used to back up a claim

2. a set of rules or guidelines

3. an element that gives a product a distinct quality (e.g., a desired quality)

**SOFTWARE ENGINEERING PRINCIPLES**

A principle is a fundamental concept (truth, statement, rule, etc.) that leads to and supports reasoning about desirable qualities like maintainability, efficiency, openness, reusability, and so on.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

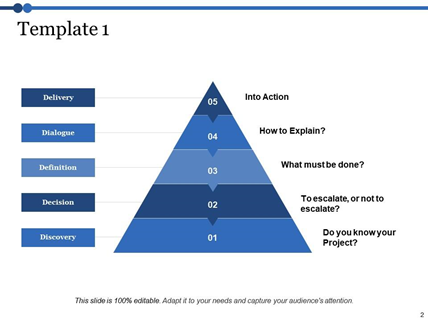
Best practices are procedures or strategies that enable developers to follow principles without having to think about the specifics of a situation on a theoretical level. Idioms are ways to express a certain method or data structure in a programming language.

**PRINCIPLES VS. BEST PRACTICES, PATTERNS, AND IDIOMS VS. DESIRABLE CHARACTERISTICS**

Although practices, patterns, and idioms are not principles in and of themselves, they aid developers in adhering to the principles.

Principles aren't desirable qualities, but sticking to them is.

**PRINCIPLE-DEFINITION TEMPLATE**

****

**OBSERVATIONS RELATIVE TO MODULARITY**

Design decisions must be made and applied in one location, users must be "hidden" from design decisions.

When software evolves in expected (and some unexpected) ways, good modularity should reduce ripple effects.

• Coupling: the degree to which components depend on each other.

• Cohesion: the degree to which the properties of a component relate to the component’s primary responsibility.

- "To construct modules that are coherent (by grouping conceptually related abstractions) and loosely connected (by minimizing dependencies among modules)," according to Grady Booch.

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

When a software system is made up of loosely linked and cohesive components, each critical or changeable design decision is isolated in one component, and related ideas are kept as near together as possible. Understanding, testability, maintainability, dependability, security, extensibility, and reuse can all benefit from modularity.

**PRACTICES AND CRITERIA:**

• Localization of design decisions

• Low Coupling

• High Cohesion

• Modular Reasoning

**TRADEOFFS:**

Localization of design decisions and strong cohesion can result in a large number of fine-grained components, which is beneficial for testability, extensibility, and reuse but can make it difficult to understand.

**PARADIGM NOTES:**

Packages and composite components sometimes have many tasks, but those responsibilities should be consistent, as indicated in the practices and criteria section.

Every predicate must reflect a single notion or obligation, according to the developers. To put it another way, each predicate should be really cohesive.

Every design decision that is interesting or potentially adjustable must be localized. For each design decision, this is accomplished by specifying a predicate and a set of rules.

**OBSERVATIONS RELATIVE TO ABSTRACTION**

"The core of abstractions," according to John Guttag, "is preserving knowledge that is important in a given context while forgetting information that is unnecessary in that context."

Software abstraction necessitates sifting through enormous and diverse collections of details to identify the most important and distinct notions. It's difficult to make good software abstractions.

**TWO COMMON PROBLEMS WITH ABSTRACTION**

Other components end up relying on details that aren't explicitly specified in the abstraction because the characteristics aren't established as such.

Too much is "forgotten" or deemed irrelevant due to abstraction.

Users of the abstraction are given insufficient control.

**PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION**

The available capabilities and functionality should be limited to what other components may require or rely on.

Understanding, testability, maintainability, and reusability can all be improved by adhering to the abstraction principle. It may also make it easier for developers to follow modularity because it will highlight flaws such as design decision localization, excessive dependency, and low cohesion.

**PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION**

**PRACTICES AND CRITERIA:**

• Meaningful labels and identifiers

• Context-aware labels and identifiers

• Abstraction completeness

• Abstraction sufficiency

**OBSERVATIONS RELATIVE TO ENCAPSULATION**

The absence of a statement, rule, or practice qualifies these definitions as principle definitions. Furthermore, these notions frequently overlap when it comes to modularity and its associated criteria.

- The bundling of data with operations

In class-based languages, definitions have given rise to access-restricting language constructs such as the private and protected modifiers. Although definitions of this category are useful, they fall short of capturing the entire potential of encapsulation.

- The hiding decisions behind logical barriers.

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION**

Ensure that a component's secret implementation details are separated from other components so that they cannot be viewed or modified. As a result, testability, maintainability, and dependability will improve. It will also aid in the separation of concerns and the avoidance of unintentional coupling.

**PRACTICES AND CRITERIA:**

• Conceptual barriers

• Programmatic barriers

• Usage barriers

**THE NON-REDUNDANCY AND COMPLIMENTARY NATURE OF THE MAE PRINCIPLES**

**CRITERIA:**

1. There can be no particular case of or subsumption of a general principle by another principle or combination of principles.

2. Developers should have the option of adhering to one principle but not the others.

**SUMMARY**

The aim of software-engineering principles has been clarified

- A framework for documenting principles was proposed.

- These definitions have been shown to be non-redundant and complementary.

**FUTURE WORK**

Form research questions about the use of MAE principles in mixed-paradigm environments.

- Construct concrete empirical studies to investigate those issues.

- Investigate metrics for evaluating quality in a mixed-paradigm software system in a systematic manner.

- Extensive research into different design principles than MAE.

14 CORREA RUIZ KERLY YADIRA

**INTRODUCTION TO THE SOLID PRINCIPLES**

Overview of the solid principles

Solid has five principles and by their acronyms they are:

Single Responsibility Principle

Open/ closed principle

Liskov Substitution Principle

Interface Segregation Principle

Interface Segregation Principle

Dependency Inversion Principle

Some argue that these principles are not important, however, following them can help ensure quality software from the developers' perspective.

**Design Problem**

Generate a program where a maze must be produced, which must be able to be printed using ASCII characters or drawn on an image.

Single responsibility principle

This principle is related to the more general principle of Cohesion that says that the responsibilities of any component (method, class, subsystem, etc.) must be closely aligned and focused on a single purpose.

It also relates to the principles of Localization of design decisions and encapsulation. This principle can help increase reusability and maintainability, reduce complexity, although the number of classes may increase.

**Open / Closed Principle**

A class is open if it is still available for extension, a class is closed if it is available for use by another class and therefore should not be modified.

A class system is open to extension and closed to modification if.

-public methods are declared using interfaces or abstract classes

- Implementations or concrete classes inherit public method declarations from interfaces, abstract classes or pure virtual classes.

Interfaces, Abstract classes, Pure virtual classes

-Revision: inheritance allows a specialization to reuse generalization: data members, method declarations, method definitions.

-An interface is like a base class, but only allows method declarations; It does not allow data members to be declared or defined, it has no method implementation.

-In UML, pure and abstract virtual class and method names are written in italics.

-A pure abstract virtual class (C++) may include data members and some method implementations.

- The Open/Closed Principle encourages developers: use abstract class and virtual class interfaces to declare public data members.

Open/ Closed Principle

Ways to achieve the Open/Closed Principle: Inheritance, Aggregation and Parameterization.

Following the open/closed principle can help developers: Reduce complexity and increase extensibility.

**Interface segregation principle**

An interface is a "window" or "portal" to the functionality of a component, it represents the public methods of a component. Java supports interfaces directly.

The public methods of a component can be grouped by purpose or responsibility as captured and declared in interfaces or abstract classes.

Following this principle helps the developer to: Reduce complexity, Increase extensibility, Increase reusability, Increase maintainability.

Dependency inversion principle

The system is organized in layers such as: reusable libraries or frameworks, these will be more abstract and others will be detail oriented.

Components of abstract layers should not depend on components of detail layers, abstractions should not depend on details.

This principle applies when:

-Abstractions must not depend on details.

-High-level modules should not depend on low-level modules.

Both low-level and high-level modules must depend on abstractions.

Helps developers: increasing reusability and maintainability.

**Software engineering goals**

Software engineers aim to build quality products on time and within budget, some desirable Qualities : understability, efficiency,reliability, reusability ,extensibility.

**Common Paradigms**

Object Orientation (OO) Logic Programming (LP) Aspect Orientation (AO)

Genetic Programming (GP) Functional programming (FP) Structures program (SP)

Java supports OO and FP, as well as AO, LP, GP when used with appropriate algorithms or libraries.

**Problem Background**

Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees. In some cases, there are conflicting definitions within the same paradigm.

15 CRUZ PANTOJA DARLING MICAELA

TOPIC: INTRODUCTION TO THE SOLID PRINCIPLES

OVERVIEW OF THE SOLID PRINCIPLES SOLID

Its main feature is a mnemonic acronym for five principles

S -ingle Responsibility Principle

O -pen/Closed Principle

L -iskov Substitution Principle

I -nterface Segregation Principle

D -ependency Inversion Principle

Some argue that these are the ”first five” principles

-That claim has not been justified or widely accepted

-However, whether they are the “first five” principles is not very important

- Following these principles can help ensure quality software, primarily from a developers’ perspective

DESIGN PROBLEM

This program is generate a program where a maze must be produced, which must be able to be printed using ASCII characters or drawn on an image.

SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility Principle is the first of the five that make up SOLID.

The Single Responsibility principle tells us that an object must do only one thing. It is very common, if we do not pay attention to this, that we end up having classes that have several logical responsibilities at the same time.

OPEN/CLOSED PRINCIPLE

This principle tells us that a software entity should be open to extension but closed to modification.

-A class is open if it is still available for extension

-A class is closed if it is available for use by other class, and therefore should not be modified

INTERFACES,ABSTRACT CLASSES,PURE VIRTUAL CLASSES

- Review: Inheritance allows a specialization (a derived class) to reuse the generalization :

-Data members

-Method declarations

-Method definitions (i.e., their implementations)

OPEN/CLOSED PRINCIPLE

Ways to achieve the open/closed principle

Inheritance

Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes

Aggregation

Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

Parameterization

Use a generic to capture a template solution and instantiate it with the specific data types and Following the Open/Closed Principle can help developers

FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE

Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file.

LISKOV SUBSTITUTION PRINCIPLE

Each class that inherits from another can be used as its parent without the need to know the differences between them.

INTERFACE SEGREGATION PRINCIPLE

Each of the three three responsibilities place in their own interfaces (or virtual classes in Java) and the product class implements all three classes

DEPENDENCY INVERSION PRINCIPLE

Core Ideas:

-Organize the system into layers: some layers, like reusable libraries or frameworks will be more abstract or policy-setting layer, others will be detail oriented

-Components from the abstract layers should not depend on components from the detail layers; instead, they should depend on abstractions that the detailed components implement

16 DE LA CRUZ QUINGA ALEJANDRO SEBASTIAN

**SOLID PRINCIPLES**

- opened closed

the software type classes, must be open to extensions and will be closed by the modification

A class is closed if it is available for use by another class

Hotfix: Inheritance makes it possible for a specialization (a derived class) to reuse the generalization (a base class):

Abstractions do not have to depend on details.

Both low-order and high-order modules have to rely on abstractions

- engineering goals

Program Engineers are committed to producing quality products on time and within budget.

LISKOV SUBSTITUTION PRINCIPLE  
  
Each class that inherits from another can be used as its parent without the need to know the differences between them.

INTERFACE SEGREGATION PRINCIPLE

An interface is a “window” into the functionality of a component

Clients should not be forced to depend upon interfaces that they do not use

DEPENDENCY INVERSION PRINCIPLE

Abstractions should not depend on details

High-level modules should not depend on low-level modules

Both low-level and high-level modules should depend on abstractions

-Background

On several occasions, there are contradictory definitions within the same paradigm • In addition, there are several other proposed principles that overlap and break the ideas in different ways.

- observations related to modularity

Good modularity should reduce domino effects once program changes happen as expected (and some unexpected).

- practical idioms and patterns

Best practices are methods or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level.

17 EIVAR DAGUA JAIME MAURICIO

**INTRODUCTION TO THE SOLID PRINCIPLES**

**OPEN/CLOSED PRINCIPLE**

Software entities (e.g., classes, generics) should be open for extension but closed to modification

A class is open if it is still available for extension

A class is closed if it is available for use by other class, and therefore should not be modified

A system of classes is open for extension and closed for modification, if

Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes (in Java)

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

Review: Inheritance allows a specialization (a derived class) to re-use the generalization’s (a base class’s):

Data members

Method declarations

Method definitions (i.e., their implementations)

**OPEN/CLOSED PRINCIPLE**

Ways to achieve the open/closed principle

**Inheritance**

Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual clases

**Aggregation**

Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

**Parmeterization**

Use a generic to capture a template solution and instantiate it with the specific data types

Following the Open/Closed Principle can help developers

Reduce complexity by reducing coupling (dependencies among components)

**DEPENDENCY INVERSION PRINCIPLE**

How to apply the Dependency Inversion Principle

Abstractions should not depend on details

High-level modules should not depend on low-level modules

Both low-level and high-level modules should depend on abstractions

“Program to the abstraction”

Following the Dependency Inversion Principle helps Developers

Increase Reusability

Increase Maintainability

**SOFTWARE ENGINEERING GOALS**

Software engineers aim to build quality products on time and within budget Some Desirable Qualities:

understandability

testability

maintainability

efficiency

eliability

security

extensibility

openness

interoperability • reusability

**PROBLEM BACKGROUND**

Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees However, the concepts and definitions of these principles differ across

paradigms.In some cases, there are conflicting definitions within the same paradigm • There are also many other proposed principles that overlap and break up the ideas differently.

**CONTRIBUTIONS OF THIS INITIAL PAPER**

The purpose of this paper is NOT to reinvent the concepts of modularity, abstraction, or encapsulation.Instead, it is to stimulate discussion about the unification of existing ideas.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level. Patterns exemplify principles, by providing proven solutions to recurring problems in specific contexts. Idioms are techniques or solution for expressing a certain algorithm or

data structure in a specific programming language, in a way that is consistent with certain principles.

**OBSERVATIONS RELATIVE TO MODULARITY**

Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways.Two concepts that can help achieve this desirable characteristic:

Coupling: the degree to which components depend on each other

Cohesion: the degree to which the properties of a component relate to the component’s primary responsibility

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

Practices and Criteria:

Localization of design decisions

Low Coupling

High Cohesion

Modular Reasoning

18 GARCIA BARRETO MAYERLY PRISSILLA

**THE SOLID PRINCIPLES**

• **S**ingle Responsibility Principle

• **O**pen/Closed Principle

• **L**iskov Substitution Principle

• **I**nterface Segregation Principle

• **D**ependency Inversion Principle

**SINGLE RESPONSIBILITY PRINCIPLE**

· The general principle of Cohesion, which says that the responsibilities of any component should be tightly aligned and focused on a single purpose.

· Reduce Complexity, even though the number of classes might increase

**OPEN/CLOSED PRINCIPLE**

· Software entities should be open for extension but closed to modification

· **open** if it is still available for extension

· **closed** if it is available for use by other class, and therefore should not be modified

**INTERFACES**

· An interface is like a base class, but only allows for method declarations

**ABSTRACT AND PURE VIRTUAL CLASSES**

· May include data members and some method implementations

· The modern Open/Closed Principle encourages developers to

use interfaces, abstract classes, and pure virtual classes to declare public data members

· Derive concrete classes from these abstract components

**OPEN/CLOSED PRINCIPLE**

· Ways to achieve the open/closed principle

• **Inheritance**

• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes

• Have concrete classes inherit from these abstraction

• **Aggregation**

• Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

• This technique has been embodied in something called the strategy pattern – more on this later

**• Parameterization**

• Use a generic to capture a template solution and instantiate it with the specific data types

**LISKOV SUBSTITUTION PRINCIPLE**

· Each class that inherits from another can be used as its parent without the need to know the differences between them.

**INTERFACE SEGREGATION PRINCIPLE**

· An interface is a “window” into the functionality of a component

· Clients should not be forced to depend upon interfaces that they do not use

**DEPENDENCY INVERSION PRINCIPLE**

· Layer the system: some levels, such as reusable libraries or frameworks, are more abstract or policy-setting, and others will be more detail-oriented.

· Abstractions should not depend on details

· High-level modules should not depend on low-level modules

· Both low-level and high-level modules should depend on abstractions

**SOFTWARE ENGINEERING PRINCIPLES PAPER**

Software engineers aim to build quality products on time and within budget

**CORE PRINCIPLES**

· Modularity

· Abstraction

· Encapsulation

COMMON PARADIGMS

· Object orientation

· Aspect orientation

· Functional programming

· Logic programming

· Genetic programming

· Structured program

**CORE PROBLEM**

· Programmer often don’t understand the core principles, and therefore don’t benefit from their guidance, especially in multi-paradigm software development

· Lack of unifying definitions hinders tools support

**BEST PRACTICES, PATTERNS, AND IDIOMS**

· Best practices are procedures or techniques that help developers adhere to principles,

· Patterns exemplify principles, by providing proven solutions to reoccurring problems in specific contexts.

· Idioms are techniques or solution for expressing a certain algorithm or data structure in a specific programming language

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION**

· Ensure that the private implementation details of a component are insulated so they cannot be accessed or modified by other components.

Practices and Criteria:

• Conceptual barriers

• Programmatic barriers

• Usage barriers

**NON-REDUNDANCY AND COMPLIMENTARY –**

**CRITERION #1**

· Modularity deals with the decomposition of system into components, whereas abstraction and encapsulation deal with individual components

· Abstraction and encapsulation might be considered duals of each other

**NON-REDUNDANCY AND COMPLIMENTARY –CRITERION #2**

**1.** A simple program snippet with good Modularity, Abstraction, Encapsulation

2. Same as #1, but with just good Modularity

3. Same as #1, but with just good Abstraction

4. Same as #1, but with just good Encapsulation

19 GOMEZ DIAZ MELISSA MALAYCA

SOLID is a mnemonic acronym for five principles:

**1.** **Single Responsibility Principle**

• Every class should be responsible for a single part of the system’s functionality

• A class’s responsibility should be entirely encapsulated by the class

• A class’s properties should be narrowly aligned with that responsibility

This principle is very closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose.

**2.** **Open/Closed Principle**

• Software entities (e.g., classes, generics) should be open for extension but closed to modification.

Original definitions:

• A class is open if it is still available for extension.

• A class is closed if it is available for use by other class, and therefore should not be modified.

Ways to achieve the open/closed principle:

Inheritance

• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes

• The public methods of one class can be grouped into multiple abstractions

• Each abstraction should focus on a single purpose, as per the Single Responsibility Principle

• Have concrete classes inherit from these abstraction

• Java does not support multiple inheritance, so a class can have multiple base classes

Aggregation

• Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

• This technique has been embodied in something called the strategy pattern – more on this later

Parameterization

• Use a generic to capture a template solution and instantiate it with the specific data types.

**3.** **Liskov Substitution Principle**

Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file.

When implementing a specialization, Widget, of some Product, ensure that:

• The implementation of save in Widget adheres to the purpose of save in

Product

• don’t have it do some unrelated thing, like re-load the object from a file instead

• Widget.save() doesn’t rely on stronger assumptions than Product.save()

• Programmatically implement any special conditions that Widget.save() required and handle exceptions appropriately

• Ensure that Widget.save() accomplishes, as minimum, all that Product.save() is supposed to accomplish.

• If Product.save() is supposed to save the x attribute to a file, then Widget.save() must do at least this much.

**4.** **Interface Segregation Principle**

Foundational Concepts:

• An interface is a ”window” or “portal” into the functionality of a component

• An interface represents public methods of a component

• An interface doesn’t have to declare all of the possible public methods of a component; a component can have many interfaces

• Java does support interfaces directly

Core Idea:

• No client (user of a component) should be forced to depend on methods that it does not use

• The public methods of a component can be grouped by purpose or responsibility as captured and declared in interfaces, or abstract classes.

**5.** **Dependency Inversion Principle**

Core Ideas:

• Organize the system into layers: some layers, like reusable libraries or frameworks will be more abstract or policy-setting layer, others will be detail oriented

• Components from the abstract layers should not depend on components from the detail layers; instead, they should depend on abstractions that the detailed components implement

• Abstractions should not depend on details

• Implementation details should depend on abstractions

How to apply the Dependency Inversion Principle

• Abstractions should not depend on details

• High-level modules should not depend on low-level modules

• Both low-level and high-level modules should depend on abstractions

• “Program to the abstraction”

20 GUAMAN VEJARANO ANGEL DAVID

Name: Angel Guaman

NRC: 7490

Matter: object oriented programming

Teacher: Edison Lascano

**Theme: Code Quality**

**Introduction to the solid principles**

**Overview of the solid principles**

• Single responsibility prínciple

• Open/Closed principle

• Interface segregation principle

• Dependency inversión principle

**First five principle**

• That claim has not been justifield or widely accepted

• However, whether they are the forst five principles is not very imporant

**Desing Problem**

• He program must produce maxes that

- Each room has at last one door to an adjoinhing rrom

- A door between two romos

**Single responsibility principle**

Core ideas

• Every class should be responsable for a single parto f the systems functionality

• A class properties should be narrowly aligned with that responsiblity

• The more genereal principle of “COHESION” which says that the any component ( method class , sub-system,ect)

**Following this principle can help**

• Increase reuse and maintainablity

• Reduce complexity, even thougth the number of classses migth increase

**Open/Closed principle**

Core ideas

• Original definitions

- A class is open i fis still available for extensión

- A class is closed if it available for use by other class

Interfaces,Abstract classes

• Inheritance allows a specialization (a derived class)

- Data members

- Method declarations

-Method definitions

• An interface is like a base class

- It has no methods implementation

- Java supports interfaces

• The modern Open/Closed principlee encourages developers to

- Derive concrete classes from these abstract components

**Liskov Substitution principle**

• An 5 object must be able to do everything any T objetc can do

- Co-variance of return type for a method in s

- Post-conditions for a method cannot be weakened is s

- invariants of the super-type must be preserved

• When implementing a specialization , Widget , of ,some ,Product, ensure that

**Interface Segregation Principle**

• Foundational concept

- An interface represents public methods if a component

- Java does support iterface directly

- An interface is a window or portal into the funcionality of a component

• Core idea

- The public methods of a component can be groped by purpose or responsilibrt

Dependency Inversion

• Organize the system into layers: some layers, like reusable librearies or frameworks

• Components from the abstract layers should not depende on components from the detalis layers

• Abstractions should not dependo n details

• Following the depedency inversión

-Increase Reusability

-Increase Maintainalibity

**UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION AS A**

**STEP TOWARD FOUNDATIONAL MULTI-PARADIGM SOFTWARE ENGINEERING**

**PRINCIPLES**

SOFTWARE ENGINEERING GOALS

· Software engineers aim to build quality products on time anwithin Budget

· Some Desirable Qualities:

• understandability• testability • maintainability • efficiency

• reliability •security • extensibility • openness • interoperability

• reusability

**COMMON PARADIGMS**

• Object orientation (OO) • Aspect orientation (AO)

• Logic programming (LP) • Genetic programming (GP)

• Functional programming (FP) • Structured program (SP)

The all PROBLEM BACKGROUND is Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees

There are also many other proposed principles that overlap and break up the ideas differently

**CORE PROBLEM**

• There are no general, unifying definitions, especially for multiparadig softwaredevelopment

- The principles are hard to teach

- Programmer often don’t understand the core principles, and therefore don’t benefit from their guidance, especially in multi-paradigm software development

**Propose a template for documenting principles that**

**•** Allows a principle’s definition to go beyond just communicating the underlying conce• provides a basis for assessing adherence to the principle and a foundation for teachinthe principle to programmers

**CONTRIBUTIONS OF THIS INITIAL PAPER**

The hoped-for contributions of the unified definitions include

providing:

• a starting point for formulating research questions for MPSD

• a foundation for designing and conducting empirical studies

• a basis for defining metrics that can systematically assess quality for MPSD

**Software Engineering Principle:**

1) a truth or proposition that supports reasoning about the desirable characteristics of a software system

2) a rule for creating software with certain desirable characteristics

3) an aspect of software design that imparts certain desirable characteristics

- that leads to and supports reasoning about desirable characteristics, such as maintainability, efficiency, openness, reusability, etc.

- then the degree to which a software engineer adheres to P should predicate the degree to which Q is present in the software artifacts.

**PRINCIPLES VS.BEST PRACTICES, PATTERNS, AND IDIOMS VS. DESIRABLE CHARACTERISTICS**

Principles should should give developers ways to

• Reason about design decisions

• Assess whether or how well a design either conforms to a principle

• Balance choices between conflicting objectives and design alternatives.

**OBSERVATIONS RELATIVE TO MODULARITY**

• The first principle of Robert Martin’s SOLID principles is a restatement of High Cohesion

• OLID principles overlap to some degree with basic idea of Low Coupling are actually specific best practices for achieving Low Coupling in certain contexts

**PARADIGM-INDEPENDENT DEFINITION FOR**

**MODULARITY**

Practices and Criteria:

• Localization of design decisions • Low Coupling

• High Cohesion• Modular Reasoning

**PARADIGM-INDEPENDENT DEFINITION FORMODULARITY**

•Developers must ensure every predicate represents a single idea or responsibility.

•This is done by defining a predicate and set of rules for each design decision.

**OBSERVATIONS RELATIVE TO ABSTRACTION**

•From a process perspective, abstraction is the act of bringing certain details to the forefront while suppressing all others.

•Abbott et al. described an abstraction as the “reification and conceptualization of a distinction”

**OBSERVATIONS RELATIVE TO ENCAPSULATION**

• Three categories of existing definition for encapsulation:

• The bundling of data with operations

• The hiding decisions behind logical barriers

• The organization of components to minimize ripple effects

· Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different

- Proposed drafts of paradigm-independent definitions for the MAE principles

Showed that these definitions are non-redundant and complimentary

21 GUITARRA SANCHEZ JHON ALEXANDER

**OVERVIEW OF THE SOLID PRINCIPLES SOLID**

is a mnemonic acronym for five principles

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Some argue that these are the ”first five” principles

That claim has not been justified or widely accepted

However, whether they are the “first five” principles is not very important

Following these principles can help ensure quality software, primarily from a developers’ perspective

**SINGLE RESPONSIBILITY PRINCIPLE**

Core ideas:

Every class should be responsible for a single part of the system’s functionality

A class’s responsibility should be entirely encapsulated by the class

A class’s properties should be narrowly aligned with that responsibility

“A class should have only one reason to change.”, Robert C. Martin

· OPEN/CLOSED PRINCIPLE

· INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES

· OPEN/CLOSED PRINCIPLE

· LISKOV SUBSTITUTION PRINCIPLE

· FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE

· INTERFACE SEGREGATION PRINCIPLE

· DEPENDENCY INVERSION PRINCIPLE

**SOFTWARE ENGINEERING GOALS**

• Software engineers aim to build quality products on time and within Budget

**THREE CORE PRINCIPLES**

Modularity

Abstraction

Encapsulation

**COMMON PARADIGMS**

• Object orientation (OO)

• Aspect orientation (AO)

• Functional programming (FP)

• Logic programming (LP)

• Genetic programming (GP)

• Structured program (SP)

**SOFTWARE ENGINEERING PRINCIPLES**

• In other words, a principle is a foundational concept (truth, proposition, rule, etc.) that leads to and supports reasoning about desirable characteristics, such as maintainability, efficiency, openness, reusability, etc

**OBSERVATIONS RELATIVE TO ABSTRACTION**

• From a process perspective, abstraction is the act of bringing certain details to the forefront while suppressing all others.

• John Guttag said that “the essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context”

22 GUZMAN LOPEZ JOSE DAVID

**INTRODUCTION TO THE SOLID PRINCIPLES**

SOLID is a mnemonic acronym for five principles:

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

Following these principles can help ensure quality software, primarily from a developers’ perspective.

**SINGLE RESPONSIBILITY PRINCIPLE**

• Every class should be responsible for a single part of the system’s functionality

• A class’s responsibility should be entirely encapsulated by the class

• A class’s properties should be narrowly aligned with that responsibility

This principle is very closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose,

**OPEN/CLOSED PRINCIPLE**

• Software entities (e.g., classes, generics) should be open for extension but closed to modification.

Original definitions:

• A class is open if it is still available for extension.

• A class is closed if it is available for use by other class, and therefore should not be modified.

Ways to achieve the open/closed principle:

Inheritance

• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes

• The public methods of one class can be grouped into multiple abstractions

• Each abstraction should focus on a single purpose, as per the Single Responsibility Principle

• Have concrete classes inherit from these abstraction

• Java does not support multiple inheritance, so a class can have multiple base classes

Aggregation

• Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

• This technique has been embodied in something called the strategy pattern – more on this later

Parameterization

• Use a generic to capture a template solution and instantiate it with the specific data types.

**LISKOV SUBSTITUTION PRINCIPLE**

Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file.

When implementing a specialization, Widget, of some Product, ensure that:

• The implementation of save in Widget adheres to the purpose of save in

Product

• don’t have it do some unrelated thing, like re-load the object from a file instead

• Widget.save() doesn’t rely on stronger assumptions than Product.save()

• Programmatically implement any special conditions that Widget.save() required and handle exceptions appropriately

• Ensure that Widget.save() accomplishes, as minimum, all that Product.save() is supposed to accomplish.

• If Product.save() is supposed to save the x attribute to a file, then Widget.save() must do at least this much.

**INTERFACE SEGREGATION PRINCIPLE**

Foundational Concepts:

• An interface is a ”window” or “portal” into the functionality of a component

• An interface represents public methods of a component

• An interface doesn’t have to declare all of the possible public methods of a component; a component can have many interfaces

• Java does support interfaces directly

Core Idea:

• No client (user of a component) should be forced to depend on methods that it does not use

• The public methods of a component can be grouped by purpose or responsibility as captured and declared in interfaces, or abstract classes.

**DEPENDENCY INVERSION PRINCIPLE**

Core Ideas:

• Organize the system into layers: some layers, like reusable libraries or frameworks will be more abstract or policy-setting layer, others will be detail oriented

• Components from the abstract layers should not depend on components from the detail layers; instead, they should depend on abstractions that the detailed components implement

• Abstractions should not depend on details

• Implementation details should depend on abstractions

How to apply the Dependency Inversion Principle

• Abstractions should not depend on details

• High-level modules should not depend on low-level modules

• Both low-level and high-level modules should depend on abstractions

• “Program to the abstraction”

23 INSUASTI LOPEZ JONATHAN ESTEBA

24 LANDAZURI SEGOVIA MATEO ISRAEL

The 5 Principles That Will Help You Develop Quality Software

If we talk about application design and development, SOLID principles are words you should know as one of the fundamentals of software architecture and development.

The SOLID Principles

The 5 SOLID principles of software application design are:

- S - Single Responsibility Principle (SRP)

- O - Open/Closed Principle (OCP)

- L - Liskov Substitution Principle (LSP)

- I - Interface Segregation Principle (ISP)

- D - Dependency Inversion Principle (DIP)

Among the objectives of taking these 5 principles into account when writing code we find:

- To create efficient software: that fulfills its purpose and that is robust and stable.

- To write a clean and flexible code in the face of changes: that can be easily modified as needed, that is reusable and maintainable.

- Allow scalability: that accepts to be extended with new functionalities in an agile way.

In short, to develop quality software.

In this sense, the application of SOLID principles is closely related to the understanding and use of design patterns, which will allow us to maintain a high cohesion and, therefore, a low software coupling.

**What are cohesion and coupling?**

They are two very relevant concepts when designing and developing software. Let's see what they consist of.

**Coupling**

Coupling refers to the degree of interdependence that two software units have on each other, meaning by software units: classes, subtypes, methods, modules, functions, libraries, etc.

If two software units are completely independent of each other, we say that they are decoupled.

**Cohesion**

Software cohesion is the degree to which different elements of a system stay together to achieve a better result than if they worked separately. It refers to the way in which we can group different software units together to create a larger unit.

**Principle of Single Responsibility**

"A class should have one, and only one, reason to change."

The S in the acronym we are talking about today refers to Single Responsibility Principle (SRP). According to this principle "a class should have one, and only one, reason to change." It is this, precisely, "reason to change," that Robert C. Martin identifies as "responsibility."

The Single Responsibility principle is the most important and fundamental principle of SOLID, very simple to explain, but the most difficult to follow in practice.

Bob himself summarizes how to do it: "Gather together the things that change for the same reasons. Separate those things that change for different reasons", i.e.: "Gather together the things that change for the same reasons. Separate those things that change for different reasons".

**2. Open/Closed Principle**

"You should be able to extend a classes behavior, without modifying it."

The second SOLID principle was formulated by Bertrand Meyer in 1988 in his book "Object Oriented Software Construction" and says: "You should be able to extend a classes behavior, without modifying it". In other words: the classes you use should be open to be extended and closed to be modified.

In his blog Robert C. Martin defended this principle that a priori may seem a paradox. It is important to take into account the Open/Closed Principle (OCP) when developing classes, libraries or frameworks.

**3. Liskov's Substitution Principle**

"Derived classes must be substitutable for their base classes."

The L in SOLID refers to the surname of its creator, Barbara Liskov, and states that "derived classes must be substitutable for their base classes".

This means that objects must be able to be replaced by instances of their subtypes without altering the correct functioning of the system or, in other words: if we use a certain class in a program, we should be able to use any of its subclasses without interfering with the functionality of the program.

According to Robert C. Martin, violating the Liskov Substitution Principle (LSP) also implies violating the Open/Closed principle.

**4. Interface Segregation Principle.**

"Make fine grained interfaces that are client specific."

In the fourth SOLID principle, Uncle Bob suggests, "Make interfaces that are client specific," i.e., for a specific purpose.

In this sense, according to the Interface Segregation Principle (ISP), it is preferable to have many interfaces that define few methods than to have an interface forced to implement many methods that it will not use.

**5. Dependency Inversion Principle**

"Depend on abstractions, not on concretions."

We come to the last principle: "Depend on abstractions, not on concretions classes".

Thus, Robert C. Martin recommends:

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.

The objective of the Dependency Inversion Principle (DIP) is to reduce dependencies between code modules, i.e., to achieve low coupling of classes.

25 LINCANGO CRIOLLO JOSE DANIEL

**INTRODUCTION TO THE SOLID PRINCIPLES**

***OVERVIEW OF THE SOLID PRINCIPLES***

• SOLID is a mnemonic acronym for five principles

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

***DESIGN PROBLEM***

*It consists of a detailed study of the problem. The input data, output data and the description of the problem must be identified.*

***SINGLE RESPONSIBILITY PRINCIPLE***

*The Single Responsibility Principle is the first of the five that make up SOLID.*

*The Single Responsibility principle tells us that an object must do only one thing. It is very common, if we do not pay attention to this, that we end up having classes that have several logical responsibilities at the same time.*

***OPEN/CLOSED PRINCIPLE***

This principle tells us that a software entity should be open to extension but closed to modification.

• A class is open if it is still available for extension

• A class is closed if it is available for use by other class, and therefore should not be modified

***INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES***

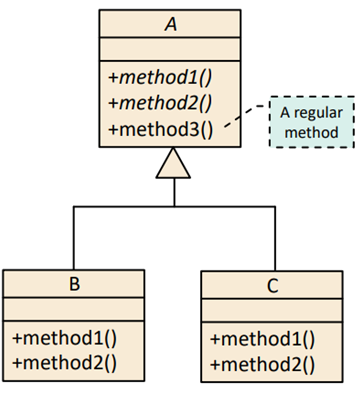
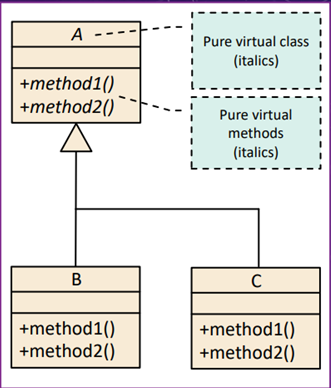
• Review: Inheritance allows a specialization (a derived class) to re-use the generalization’s (a base class’s)

• An interface is like a base class, but only allows for method declarations

• In UML, the names of abstract and pure virtual classes and methods are written in italics

• An abstract and a pure virtual class (C++) may include data members and some method implementations

• The modern Open/Closed Principle encourages developers to



***OPEN/CLOSED PRINCIPLE***

•Inheritance

•Aggregation

• Parameterization

• Following the Open/Closed Principle can help developers

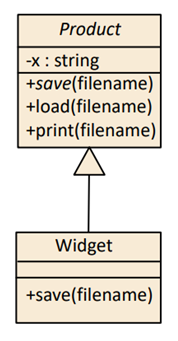
***LISKOV SUBSTITUTION PRINCIPLE***

El principio de sustitución de Liskov nos dice que si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa sigue siendo válido.

Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento del padre.

***FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE***

This principle comes to disprove the preconceived idea that classes are a direct way of modeling reality, and that care must be taken with that modelling.



***LISKOV SUBSTITUTION PRINCIPLE***

• Following the Liskov Substitution Principle can help developers

• Increase Reuse

• Increase Extensibility

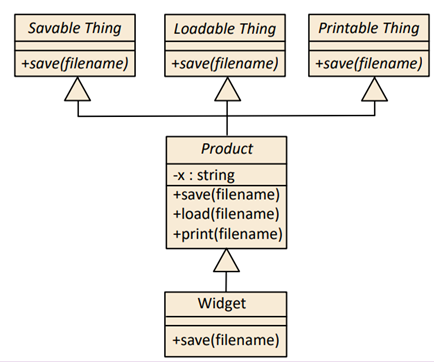
• Increase Maintainability

***INTERFACE SEGREGATION PRINCIPLE***

The principle of interface segregation means that no class should depend on methods that it does not use.

Therefore, when creating interfaces that define behaviors, it is important to make sure that all classes that implement those interfaces will need and be able to add behaviors to all methods. Otherwise, it is better to have several smaller interfaces.

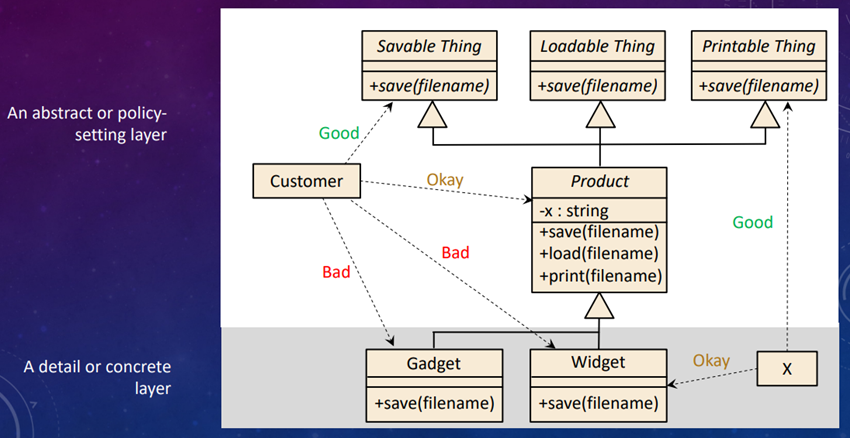
***INTERFACE SEGREGATION PRINCIPLE***

**

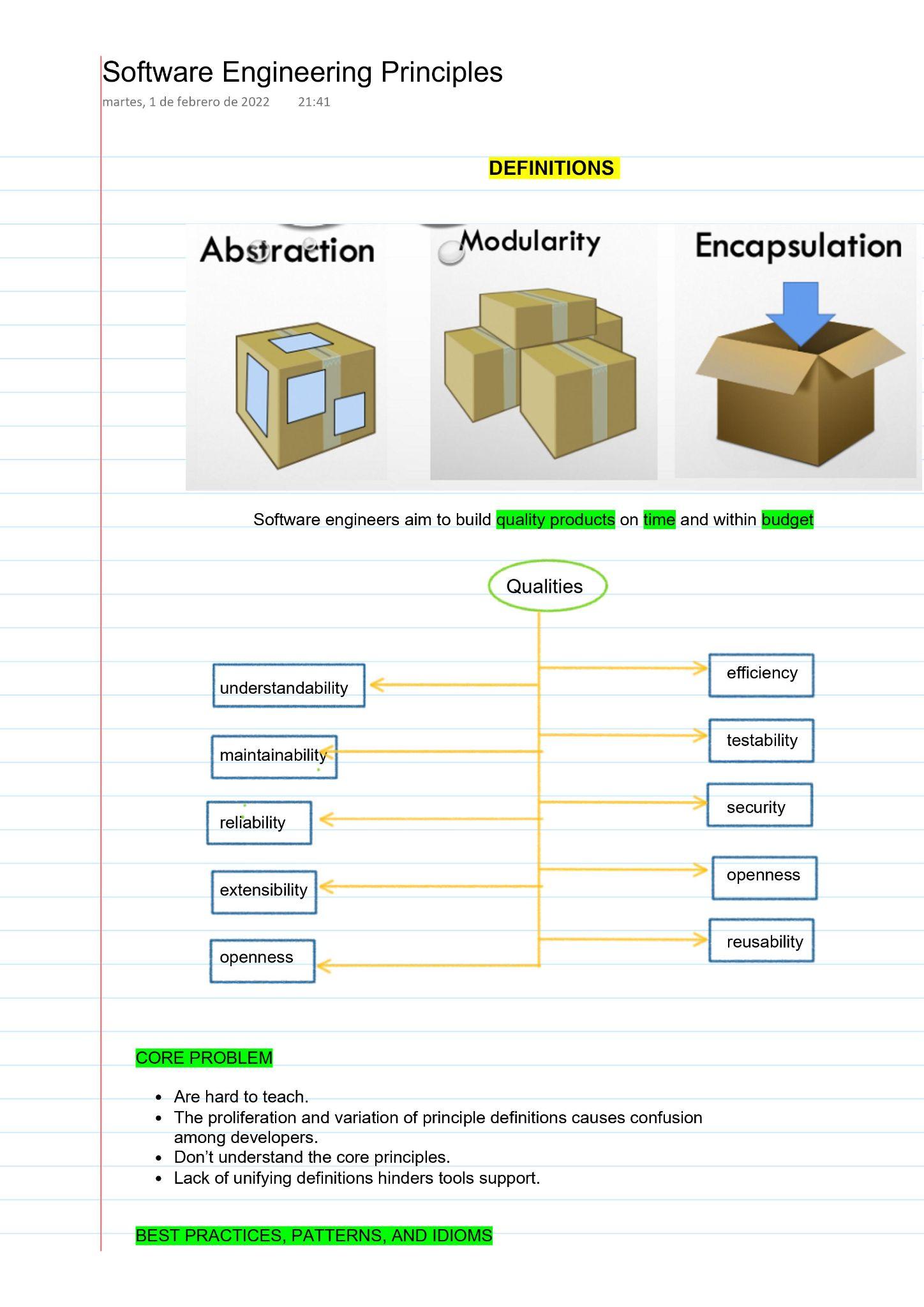
**DEPENDENCY INVERSION PRINCIPLE**

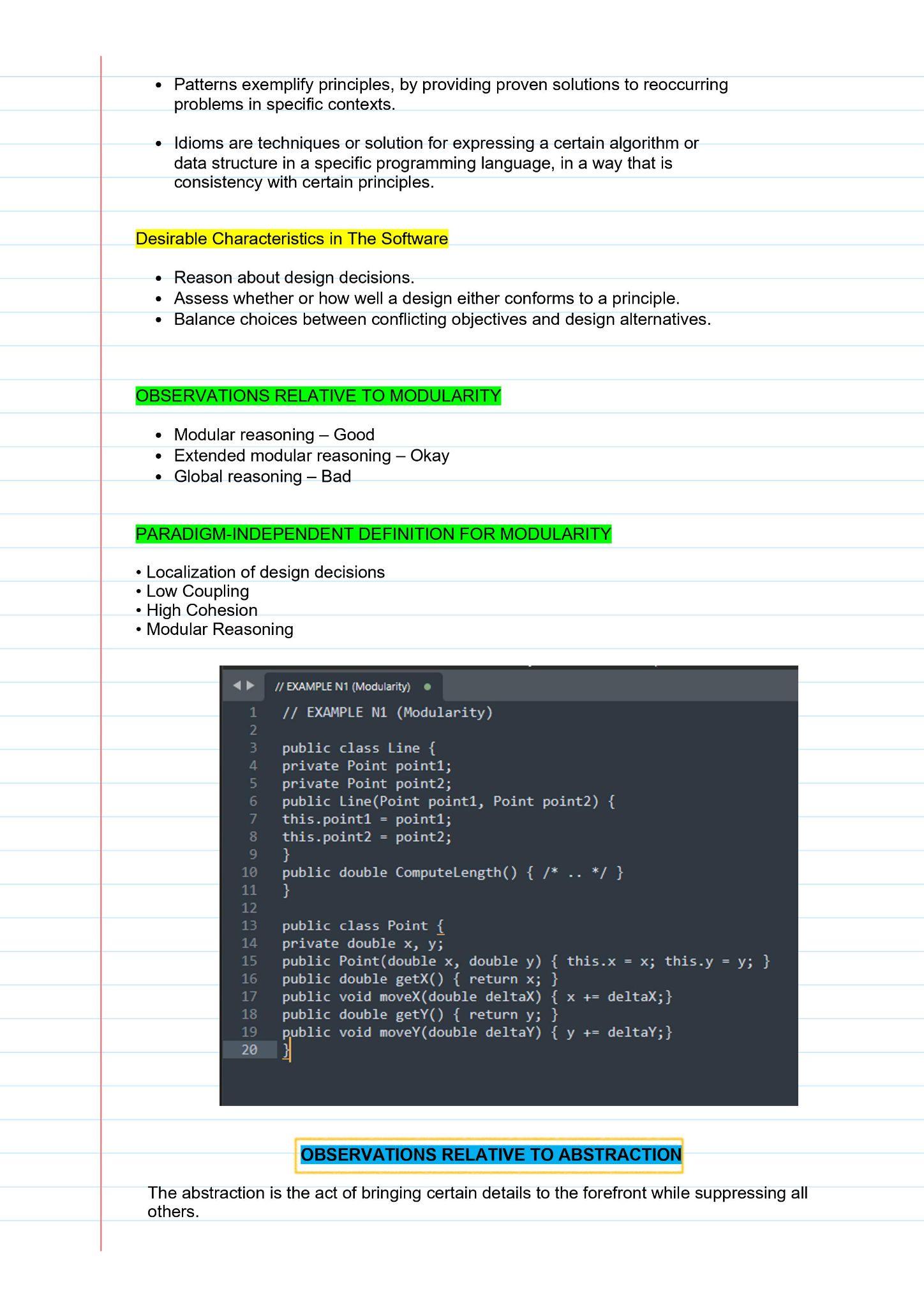
Interfaces help us decouple modules from each other. This is so because if we have an interface that explains the behavior that the module expects to communicate with other modules, we can always create a class that implements it so that it meets the conditions.

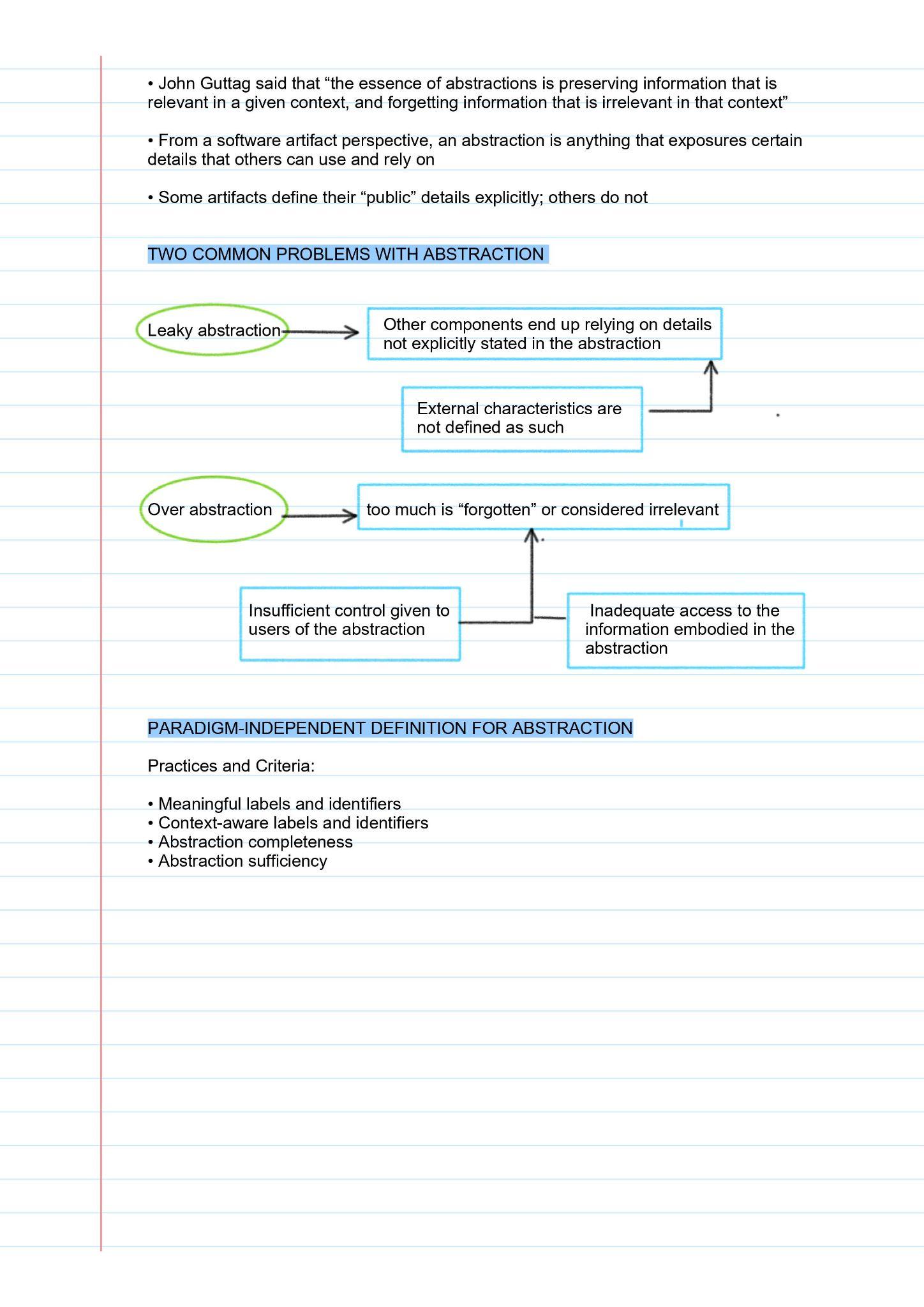
**DEPENDENCY INVERSION PRINCIPLE**

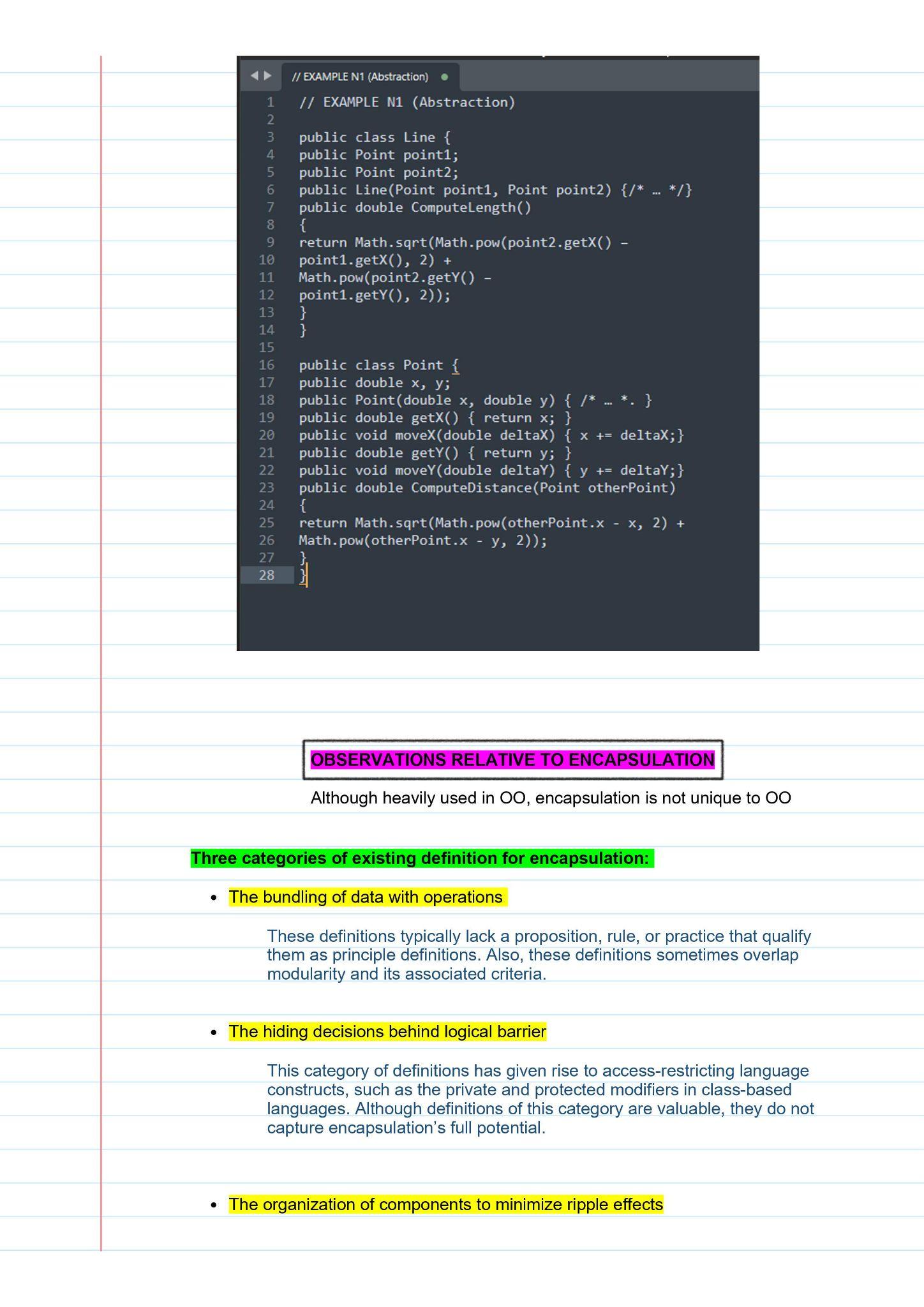


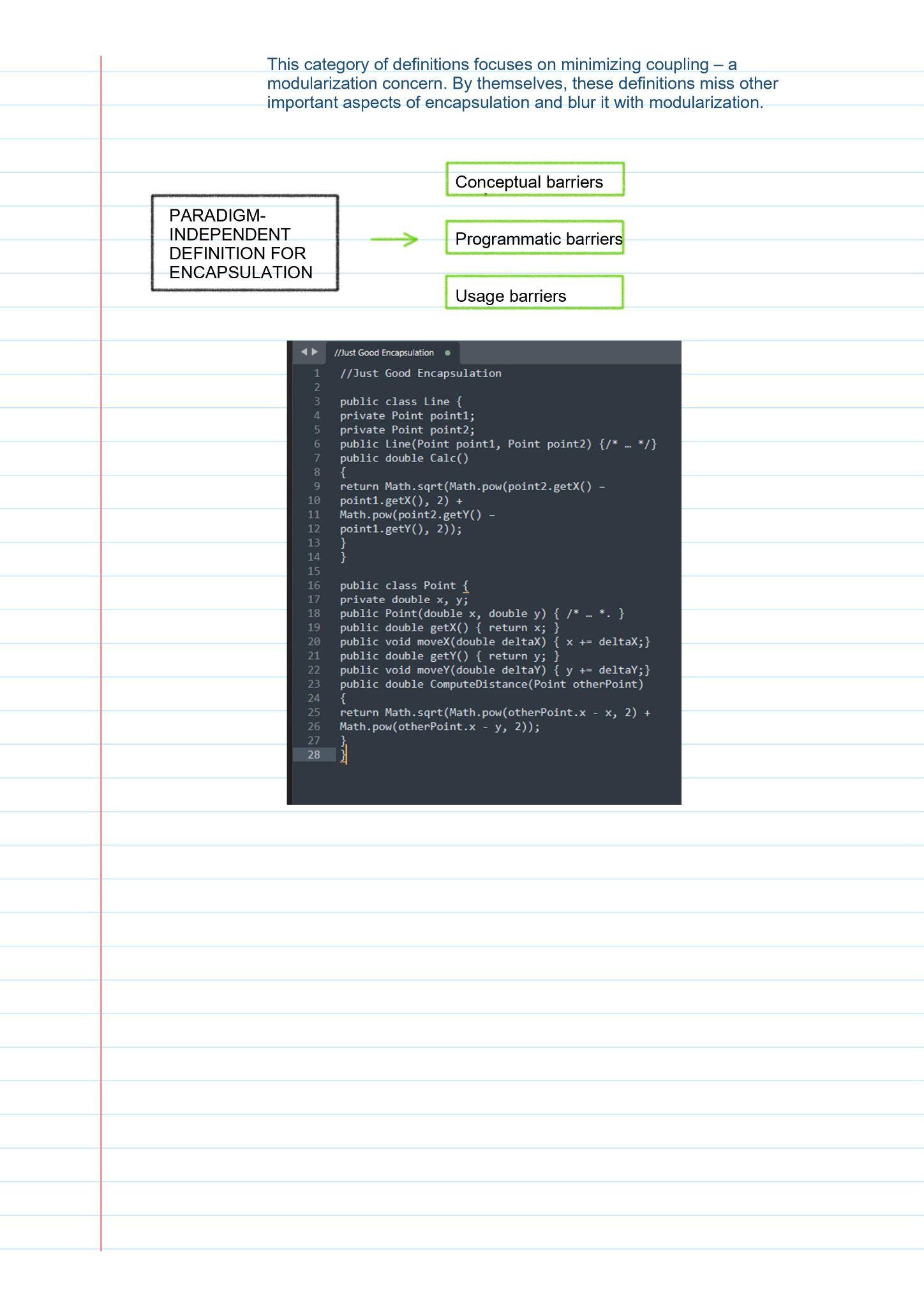
26 MAISINCHO PAUCAR RICHAR ALEXANDER











27 MALDONADO BASTIDAS MATEO STEFANO

**SOLID PRINCIPLES**

**SOLID** is a mnemonic acronym for five principles: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle.

**DESIGN PROBLEM:** Contain N x M square rooms, where N and M are the user-specified width and height of the maze**,** each room has at least one door to an adjoining room the program must be able to print the mazes using ASCII character or draw them in an image

**SINGLE RESPONSIBILITY PRINCIPLE:** This principle is very closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose

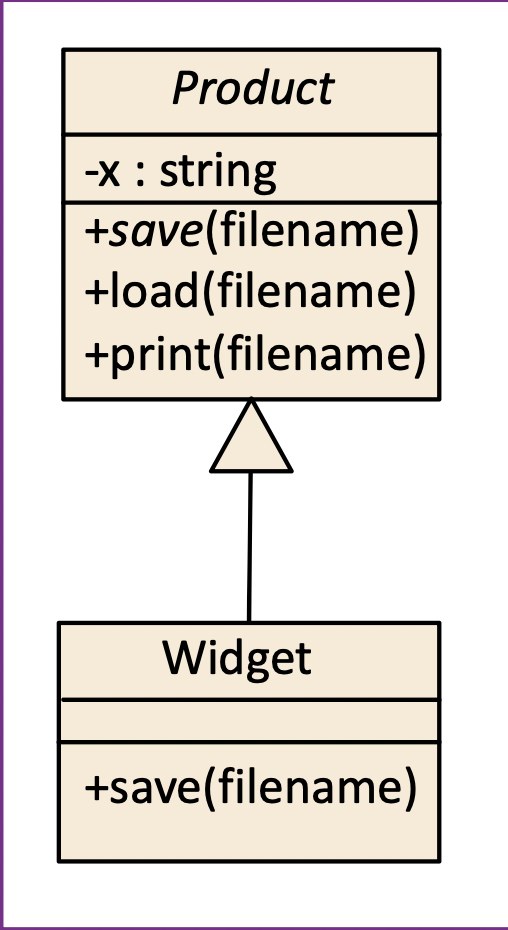
**OPEN/CLOSED PRINCIPLE:** A class is open if it is still available for extension a class is closed if it is available for use by other class, and therefore should not be modified

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES:** An abstract and a pure virtual class(C++)may include data members and some method implementations, the modern Open/Closed Principle encourages developers to use interfaces, abstract classes, and pure virtual classes to declare public data members

**OPEN/CLOSED PRINCIPLE:** Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes, Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically, this technique has been embodied in something called the strategy pattern – more on this later, use a generic to capture a template solution and instantiate it with the specific data types

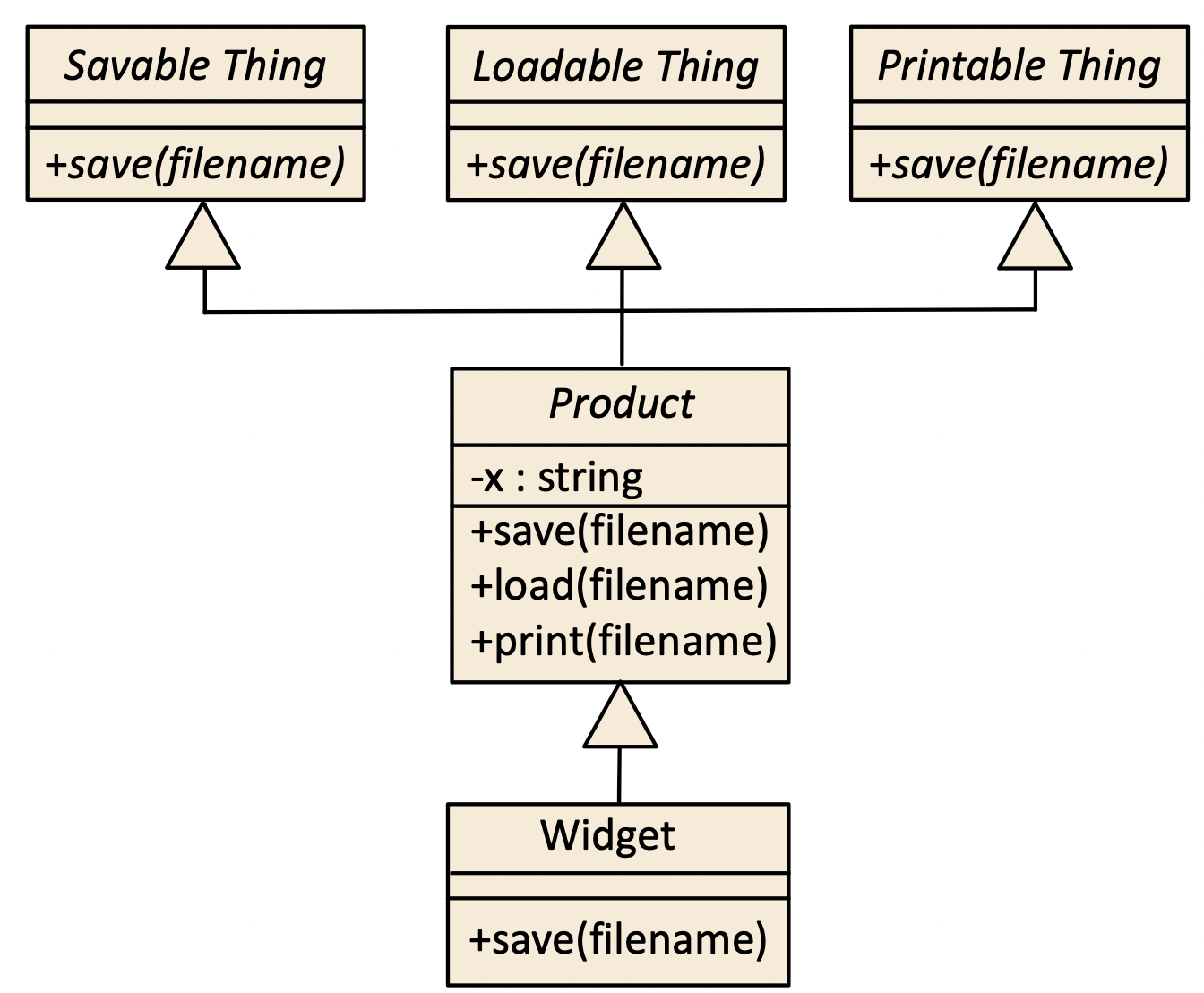
**LISKOV SUBSTITUTION PRINCIPLE:** if S is a specialization of T, then an S object can be used wherever a T object is required, Strong behavioral subtyping – an S object must be able to do everything any T object can do

Let Product be a base class, with one virtual method ,called save, whose intent is to save an object to a file widget, of some Product, ensure tha the implementation of save in Widget adheres to the purpose of save in Product



**INTERFACE SEGREGATION PRINCIPLE:** an interface represents public methods of a component, an interface doesn’t have to declare all the possible public methods of a

component: a component can have many interfaces, Following the Interface Segregation Principle, when used with other principles, can help Developers reduce complexity by increasing Cohesion and reducing Coupling

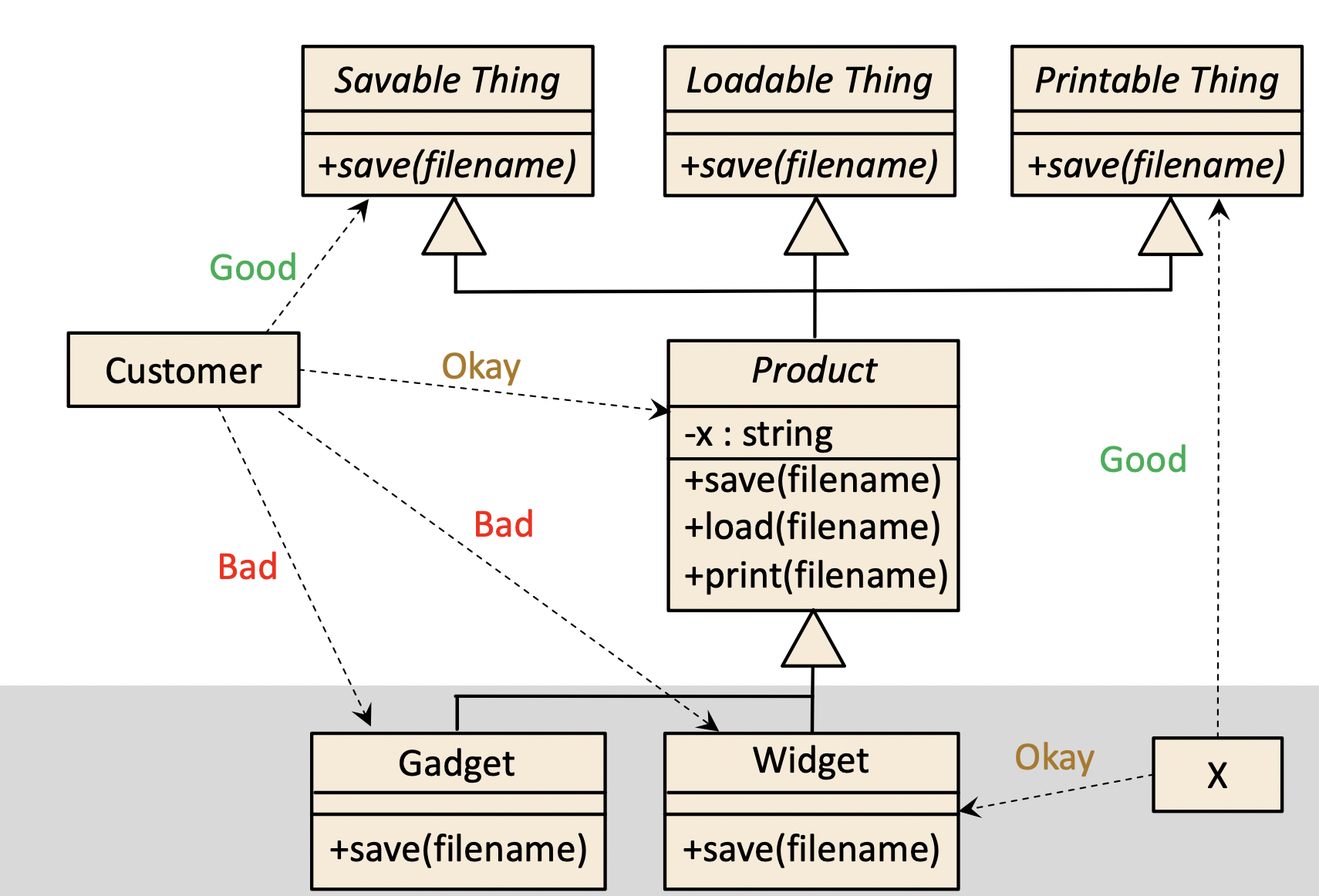


**DEPENDENCY INVERSION PRINCIPLE:** How to apply the Dependency Inversion Principle

• High-level modules should not depend on low-level modules

• Both low – level and high-level modules should depend on abstractions • “Program to the abstraction”

• Following the Dependency Inversion Principle helps Developers



**COMMON PARADIGMS**

Today, many languages and development environments support multi-paradigm software development, Object orientation (OO), Aspect orientation (AO), Functional programming (FP), Logic programming (LP), Genetic programming (GP) • Structured program (SP)

**PROBLEM BACKGROUND:** Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees, however, the concepts and definitions of these principles differ across paradigms.

**CONTRIBUTIONS OF THIS INITIAL PAPER:** Clarify the purpose of software-engineering (SE) principles, in general, and distinguish them from “best practices”, idioms, and patterns propose a template for documenting principles that allows a principle’s definition to go beyond just communicating the underlying concepts, provides a basis for assessing adherence to the principle and a foundation for teaching the principle to programmers

**SOFTWARE ENGINEERING PRINCIPLES:** a principle is a foundational concept (truth, proposition, rule, etc.) that leads to and supports reasoning about desirable characteristics, such as maintainability, efficiency, openness, reusability, etc.

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION:** Ensure that the private implementation details of a component are insulated so they cannot be accessed or modified by other components. Doing so will lead to better testability, maintainability, and reliability. It will also help with a clear separation of concerns and avoid accidental coupling.

**NON-REDUNDANCY AND COMPLIMENTARY – CRITERION:** Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different, we show satisfaction of the second criteria, namely that developers and choose to follow each principle independent, with an example consisting of four functional-identical code snippets a simple program snippet with good Modularity, Abstraction, Encapsulation

28 MANTUANO FERNANDEZ LEONEL FERNANDO

**INTRODUCTION TO THE SOLID PRINCIPLES**

They help the quality of the software in view of the developers.

**SINGLE RESPONSIBILITY PRINCIPLE**

The responsibilities of any component must have a single purpose, to maintain a single reason for change.

**OPEN/CLOSED PRINCIPLE**

Software entities can only have extensions, in other words they cannot be modified.

Implementations have pure method declarations.

Public methods can be grouped into abstractions, each with a unique purpose.

Can be encapsulated to vary secondary objects, each component decreasing complexity.

It has ways of roaming, aggregation and Parameterization.

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

Allows the declaration of methods with interface constraints.

Objects cannot be instantiated.

**LISKOV SUBSTITUTION PRINCIPLE**

An object will always need another object. Both objects may be able to do what the other can do.

The principle can increase reusability, extensibility, and maintainability.

**DEPENDENCY INVERSION PRINCIPLE**

Order in layers, the abstraction does not depend on the details but on the details of the abstractions.

**DEPENDENCY INVERSION PRINCIPLE**

High and low level modules depend on abstractions but do not depend on low level, increasing

reusability and maintainability.

**SOFTWARE ENGINEERING GOALS**

To produce quality products with a low budget.

**PROBLEM BACKGROUND**

Three main paradigms are postulated: modularity, abstraction and encapsulation.

**CORE PROBLEM**

Programmers lack basic knowledge of multi-paradigm software development.

**SOFTWARE ENGINEERING PRINCIPLES**

The definition of the principle aids in the completion of the features to be realized.

The practice of idioms express algorithms in a coherent way that exemplify the patterns and principles of solution.

**OBSERVATIONS RELATIVE TO MODULARITY**

It consists of building modules by grouping abstractions with almost no dependencies, it facilitates the process of loosely coupled units.

It uses part of the encapsulation by hiding the components from the users, for an efficient development it uses coupling and cohesion.

For FP it depends on parameters

LPs consist of rules and facts

Each design decision must be localized in time.

**OBSERVATIONS RELATIVE TO ABSTRACTION**

The important thing is to keep the important details in the foreground, and leave out the useless ones. Some abstraction artifacts show the public details.

For a developer creating a software abstraction leads to extensive and tedious analysis.

The most eye-catching problems are: Leaky, excessive abstraction and inefficient control of details.

**PARADIGM-INDEPENDENT DEFINITION FOR**

**ABSTRACTION**

The declaration of a component may be part of the source code and may become dependent on it, its practices and criteria consist of meaningful tags and identifiers.

**OBSERVATIONS RELATIVE TO ENCAPSULATION**

There are three types of definition:

Grouping: lacking proposition, and confused with modularity.

Hiding: Linguistic construction

Data organization: Minimizes coupling.

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION**

Details are kept private to prevent access or modification, achieving greater reliability and avoiding accidental coupling.

29 MORALES CAICEDO ANTHONY JAVIER

**Solid Principles**

The word SOLID is an acronym for the following principles:

- Single Responsibility Principle

- Open/closed principle

- Liskov substitution principle

- Interface separation principle

- Dependency Inversion Principle

According to many these previous principles are the first five, although this has not been confirmed. In any case, following these principles guarantees software quality.

**Single Responsibility Principle (main ideas):**

- Each class is responsible for one part of the system (in its functionality).

- The responsibility of the class must be encapsulated by the class.

- The properties of the class are related to that responsibility.

This whole principle is familiar with the principle of cohesion, in which it is specified that any component is related to the purpose. In addition, it relates to the localization of design decisions and encapsulation.

**Open/closed principle (main ideas):**

- Software entities are open for extension, but closed to modification.

**Original definitions:**

An open class is available for extension.

A class is closed if it is to be used by another class.

**Method declarations:**

An interface is like a base class, but only allows method declarations.

It does not allow data members to be declared.

**Ways to achieve the open/closed principle:**

- Inheritance: move public methods to their own abstractions, public methods of a class can be grouped into multiple abstractions.

Java does not support multiple inheritance, so a class can have multiple base classes.

- Aggregation: Encapsulate behaviors in objects and allow them to change dynamically.

This technique has been incorporated into the strategy pattern.

- Parameterization: Capture a template solution with specific data types.

**Liskov substitution principle:**

The product of a base class with a method called knows, which is used to save the object to the file.

Do not reload the object from a file.

Make sure that Widget.save() accomplishes everything it was programmed to do.

Liskov's substitution principle serves to: increase reusability, increase extensibility, increase maintainability.

**Interface separation principle:**

Fundamental concepts: an interface is a window into the functionality of a component, the interface does not have to declare the possible methods of a component, a component can have many interfaces.

**Dependency inversion principle:**

Core ideas: organize the system into packages, abstractions should not depend on details, implementation details depend on abstractions.

**Software Engineering Goals**

· supports reasoning about the desirable characteristics of a software system

· creates software with certain desirable characteristics

· an aspect of software design that imparts desirable characteristics.

**Best practices, patterns and idioms**

- These are procedures or techniques that help developers adhere to principles, without having to consider the principles or details of a situation at a theoretical level.

- The patterns exemplify the principles, providing proven solutions to recurring problems in specific contexts.

- Idioms are techniques that express a particular algorithm or data structure in a programming language.

**Observation relative to modulary**

- Good modularity should minimize the domino effect when software changes occur in expected (and some unexpected) ways.

- There are two concepts that can help achieve this desirable characteristic:

- Coupling: the degree to which components depend on each other.

- Cohesion: the degree to which the properties of a component relate to the component's primary responsibility.

**Paradigm-independent definition for modulary**

Counterparts:

- Localization of design decisions and high cohesion can lead to many components which is good for testability, extensibility and reusability, but can hinder readability.

- By itself, modularity does not guarantee desirable features.

- However, the lack of modularity will compromise desirable features

- Adherence to or violation of modularity principles often affects multiple components.

**Observations relative to abstraction**

- Creating good software abstractions is difficult

- Software abstraction requires developers to examine large and

of details, and then determine the most salient and distinctive concepts.

- Abbott et al. describe an abstraction as the “reification and conceptualization of a distinction.”

**Paradigm-independent definition for encapsulation**

PARADIGM INDEPENDENT DEFINITION FOR ABSTRACTION

Practices and criteria:

- Meaningful labels and identifiers

- Context-aware labels and identifiers

- Completeness of abstraction

- Abstraction Sufficiency OBSERVATIONS RELATING TO ENCAPSULATION

**NON-REDUNDANCY AND COMPLEMENTARITY**

**NATURE OF MAE PRINCIPLES**

**CRITERION #1**

- Modularity is concerned with decomposition into components, whereas abstraction and encapsulation are concerned with individual components.

- Therefore, modularity cannot be subsumed by either of the other two.

- Abstraction and encapsulation can be considered dual to each other.

**CRITERION #2**

- We demonstrate that the second criterion is met, i.e., that developers choose to follow each principle independently.

30 PALACIOS CANDO DIEGO SEBASTIAN

**INTRODUCTION TO THE SOLID PRINCIPLES**

**OPEN/CLOSED PRINCIPLE**

Software entities (e.g., classes, generics) should be open for extension but closed to modification

A class is open if it is still available for extension

A class is closed if it is available for use by other class, and therefore should not be modified

A system of classes is open for extension and closed for modification, if

Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes (in Java)

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

Review: Inheritance allows a specialization (a derived class) to re-use the generalization’s (a base class’s):

Data members

Method declarations

Method definitions (i.e., their implementations)

**OPEN/CLOSED PRINCIPLE**

Ways to achieve the open/closed principle

**Inheritance**

Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual clases

**Aggregation**

Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

**Parmeterization**

Use a generic to capture a template solution and instantiate it with the specific data types

Following the Open/Closed Principle can help developers

Reduce complexity by reducing coupling (dependencies among components)

**DEPENDENCY INVERSION PRINCIPLE**

How to apply the Dependency Inversion Principle

Abstractions should not depend on details

High-level modules should not depend on low-level modules

Both low-level and high-level modules should depend on abstractions

“Program to the abstraction”

Following the Dependency Inversion Principle helps Developers

Increase Reusability

Increase Maintainability

**SOFTWARE ENGINEERING GOALS**

Software engineers aim to build quality products on time and within budget Some Desirable Qualities:

understandability

testability

maintainability

efficiency

eliability

security

extensibility

openness

interoperability • reusability

**PROBLEM BACKGROUND**

Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees However, the concepts and definitions of these principles differ across

paradigms.In some cases, there are conflicting definitions within the same paradigm • There are also many other proposed principles that overlap and break up the ideas differently.

**CONTRIBUTIONS OF THIS INITIAL PAPER**

The purpose of this paper is NOT to reinvent the concepts of modularity, abstraction, or encapsulation.Instead, it is to stimulate discussion about the unification of existing ideas.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level. Patterns exemplify principles, by providing proven solutions to recurring problems in specific contexts. Idioms are techniques or solution for expressing a certain algorithm or

data structure in a specific programming language, in a way that is consistent with certain principles.

**OBSERVATIONS RELATIVE TO MODULARITY**

Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways.Two concepts that can help achieve this desirable characteristic:

Coupling: the degree to which components depend on each other

Cohesion: the degree to which the properties of a component relate to the component’s primary responsibility

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

Practices and Criteria:

Localization of design decisions

Low Coupling

High Cohesion

Modular Reasoning

31 PAUCAR LEMA ALEX JAVIER

**The solid principles 1.14.40 AM**

SOLID is a mnemonic acronym:

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Following these principles can help ensure quality software, primarily from a developer's perspective.

**SINGLE RESPONSIBILITY PRINCIPLE:**

- Every class should be responsible for a single part of the system’s functionality.

- “A class should have only one reason to change”, Robert C. Martin

This principle is vey closely related to the more general principle of Cohesion, which says that the responsibilities of any component should be tightly aligned and focused on a single purpose.

This principle is also related to the principles of: localization of design decision and encapsulation.

Following this principle can help: reduce complexity, even though the number of classes might increase.

**OPEN/CLOSED PRINCIPLE:**

Software entities should be open for extension but closed to modification.

A class is open if it is still available for extension

A class is closed if it is available for use by other class.

A system of classes is open for extension and closed for modification, if:

- Public methos are declared using interfaces, or abstract classes

- Users

- Concrete classes inherit the public method declarations from the interfaces, abstract classes, or pure virtual classes.

**INTERFACES. ABSTRACT CLASSES, PURE VIRTUAL CLASSES:**

Inheritance allows a specialization to re-use the generalization’s:

- Data members

- Method declarations

- Methods definitions

An interface is like a base class, but only allows for method declarations:

- It does not allow data members to be declared

- It has no methods implementation

- Java supports interfaces

In UML, the names are written in italics

An abstract and pure virtual class may include data members and some method implementations:

- There is at least one method declaration without an implementation

The modern Open/Closed Principle encourages developers to:

- Use interfaces, abstract classes, and pure virtual classes to declare public data members

Ways to achieve the open/closed principle:

- Inheritance:

o Move public methods into their own abstractions.

o The public methos of one class can be grouped into multiple abstractions

o Each abstraction should focus on a single purpose

o Java does not support multiple inheritance, so class can have multiple base classes.

- Aggregation

o Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

o This technique has been embodied in something called the strategy pattern

- Parameterization

o Use a generic to capture a template solution and instantiate it with the specific data types.

Following the Open/Closed Principle can help developers

- Reduce complexity

- Increase extensibility.

**LISKOV SUBSTITUTION PRINCIPLE:**

Use a generic to capture a template solution and instantiate it with the specific data types

**FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE:**

When implementing a specialization, Widget, of some Product, ensure that:

- The widget implementation adheres to the purpose of product specialization

Following the Liskov Substitution Principle can help developers

- Increase reuse

- Increase Extensibility

- Increase Maintainability

**INTERFACE SEGREGATION PRINCIPLE:**

An interface is a” window” or “portal” into the functionality of a component.

A component can have many interfaces.

Java does support interfaces directly.

An interface represents public methods of a component.

The public methods of a component can be grouped by purpose or responsibility

No client should be forced to depend on methods that it does not use.

**INTERFACE SEGREGATION PRINCIPLE:**

Following the Interface Segregation Principle, when used with other principles, can help Developers:

- Reduce complexity by increasing cohesion and reducing coupling

- Increase extensibility

- Increase reuse

- Increase maintainability

**DEPENDENCY INVERSION PRINCIPLE:**

Organize the system into layers

Components from the abstract layers should not depend on components from the detail layers; instead, they should depend on abstractions that the detailed components implement

Abstractions should not depend on details

Implementation details should depend on abstractions

How to apply the Dependency Inversion Principle:

- Abstractions should not depend on details

- High-level modules should not depend on low-level modules

- Both low-level and high-level modules should depend on abstractions

Following the Dependency Inversion Principle helps Developers:

- Increase Reusability

- Increase Maintainability

32 QUINGA GUAYASAMIN LEANDRO ALEXANDER

*THE SOLID PRINCIPLES*

**OVERVIEW OF THE SOLID PRINCIPLES**

SOLID is a mnemonic acronym for five principles

1. Single Responsibility Principle
2. Open/Closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

**1.SINGLE RESPONSIBILITY PRINCIPLE**

* Every class should be responsible for a single part of the system’s functionality
* A class’s responsibility should be entirely encapsulated by the class
* A class’s properties should be narrowly aligned with that responsibility
* “A class should have only one reason to change.”, Robert C. Martin

**2.OPEN/CLOSED PRINCIPLE**

* Software entities (e.g., classes, generics) should be open for extension but closed to

modification

* A class is open if it is still available for extension
* A class is closed if it is available for use by other class, and therefore should not be

modified

**3.LISKOV SUBSTITUTION PRINCIPLE**

* if S is a specialization of T, then an S object can be used wherever a T object is required.

**4.INTERFACE SEGREGATION PRINCIPLE**

* An interface is a ”window” or “portal” into the functionality of a component
* An interface represents public methods of a component
* An interface doesn’t have to declare all of the possible public methods of a

component; a component can have many interfaces

* Java does support interfaces directly
* No client (user of a component) should be forced to depend on methods that it

does not use

* The public methods of a component can be grouped by purpose or responsibility

as captured and declared in interfaces, or abstract classes

**5.DEPENDENCY INVERSION PRINCIPLE**

* Organize the system into layers: some layers, like reusable libraries or

frameworks will be more abstract or policy-setting layer, others will be

detail oriented

* Components from the abstract layers should not depend on components

from the detail layers; instead, they should depend on abstractions that

the detailed components implement

* Abstractions should not depend on details

Implementation details should depend on abstractions

*SOFTWARE ENGINEERING PRINCIPLES*

* In other words, a principle is a foundational concept (truth,

proposition, rule, etc.) that leads to and supports reasoning about

desirable characteristics, such as maintainability, efficiency,

openness, reusability, etc.

* If some concept, P, is an effective principle for achieving a set of

desirable characteristics Q, then the degree to which a software

engineer adheres to P should predicate the degree to which Q is

present in the software artifacts.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

* Patterns exemplify principles, by providing proven solutions to reoccurring

problems in specific contexts.

* Idioms are techniques or solution for expressing a certain algorithm or

data structure in a specific programming language, in a way that is

consistency with certain principles.

**PRINCIPLES VS.BEST PRACTICES, PATTERNS, AND IDIOMS VS. DESIRABLE CHARACTERISTICS**

* Principles are not desirable characteristics, but adherence to a principle

should lead to desirable characteristics in the software, e.g.

* Abstraction à maintainability, reuse
* Principles should should give developers ways to
* Reason about design decisions
* Assess whether or how well a design either conforms to a principle
* Balance choices between conflicting objectives and design alternatives.

**OBSERVATIONS RELATIVE TO MODULARITY**

* Design decisions need to be “hidden” from the users of the

component in which they are placed – this is actually

encapsulation

**OBSERVATIONS RELATIVE TO ABSTRACTION**

* From a process perspective, abstraction is the act of bringing certain

details to the forefront while suppressing all others.

* From a software artifact perspective, an abstraction is anything that

exposures certain details that others can use and rely on

* Software abstraction requires developers to sift through large and

diverse collections of details, and then determine the most salient

and distinguishing concepts

**OBSERVATIONS RELATIVE TO ENCAPSULATION**

Three categories of existing definition for encapsulation:

* The bundling of data with operations
* The hiding decisions behind logical barriers
* The organization of components to minimize ripple effect

33 RUANO PONCE ALEXANDER JAVIER

**Solid Principles**

***SOLID is a mnemonic acronym for five principles:***

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

***SINGLE RESPONSIBILITY PRINCIPLE***

• Core ideas:

• “A class should have only one reason to change.”, Robert C. Martin

• This principle is vey closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose

• Localization of design decisions

• Encapsulation

• Following this principle can help

• Increase Reuse and Maintainability

• Reduce Complexity, even though the number of classes might increase

***OPEN/CLOSED PRINCIPLE***

• Ways to achieve the open/closed principle

• Inheritance

• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes

• Aggregation

• Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

• Parameterization

• Reduce complexity by reducing coupling (dependencies among components)

• Increase extensibility

***LISKOV SUBSTITUTION PRINCIPLE***

• Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file

• When implementing a specialization, Widget, of some Product, ensure that

• The implementation of save in Widget adheres to the purpose of save in Product

• don’t have it do some unrelated thing, like re-load the object from a file instead

• Widget.save() doesn’t rely on stronger assumptions than Product.save()

• Programmatically implement any special conditions that Widget.save() required and handle exceptions appropriately

• Ensure that Widget.save() accomplishes, as minimum, all that Product.save() is supposed to accomplish

• If Product.save() is supposed to save the x attribute to a file, then Widget.save() must do at least this much.

• Following the Liskov Substitution Principle can help developers

• Increase Reuse

• Increase Extensibility

• Increase Maintainability

***INTERFACE SEGREGATION PRINCIPLE***

• An interface is a ”window” or “portal” into the functionality of a component

• An interface represents public methods of a component

• An interface doesn’t have to declare all of the possible public methods of a component; a component can have many interfaces

• Java does support interfaces directly

• No client (user of a component) should be forced to depend on methods that it does not use • The public methods of a component can be grouped by purpose or responsibility as captured and declared in interfaces, or abstract classes.

***DEPENDENCY INVERSION PRINCIPLE***

• How to apply the Dependency Inversion Principle

• Abstractions should not depend on details

• High-level modules should not depend on low-level modules

• Both low-level and high-level modules should depend on abstractions

• “Program to the abstraction”

• Following the Dependency Inversion Principle helps Developers

• Increase Reusability

• Increase Maintainability

34 SALTOS TACO PAUL ALEXANDER

**INTRODUCTION TO THE SOLID PRINCIPLES**

There are five specific aspects of object-oriented programming that each **SOLID** principle addresses, with each letter representing a principle. Fortunately, this acronym makes the five principles relatively easy to memorize:

* **S**ingle Responsibility Principle
* **O**pen/Closed Principle
* **L**iskov Substitution Principle
* **I**nterface Segregation Principle
* **D**ependency Inversion Principle

None of these principles is truly exclusive. On the contrary, it could be argued that they are mutually inclusive. Some of them represent multiple strategies and each of them plays a role in achieving a single goal. Or, in other cases, they are by-products: Proper adherence to one of these SOLID practices can naturally beget another.

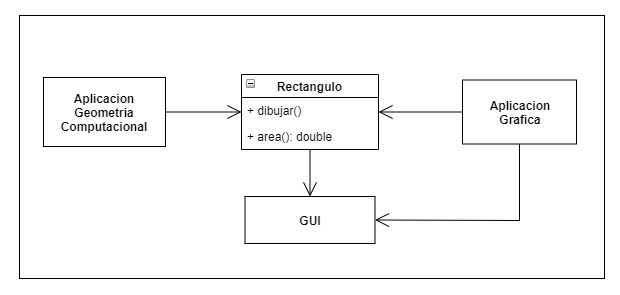
**SOFTWARE ENGINEERING PRINCIPLES PAPER**

This document presents some important general principles, which are central to

develop software successfully, and that deal with both the software engineering process and the Final product. The right process will help develop the desired product, but also the product desired will affect the choice of process to use. A traditional software engineering problem is put the emphasis on the process or the product to the exclusion of the other, however, both are important.

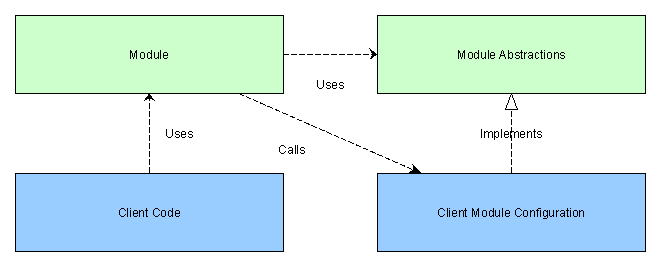
**Single Responsibility Principle**

It is true when our class only does one thing. It tells us that a module has only one reason to change. For software engineering, it states that each module or class should have responsibility for only one part of the functionality provided by the software, and this responsibility must be fully encapsulated by the class.



**Open/Closed Principle**

This principle tells us that a software entity should be open to extension but closed to modification.This helps us keep adding functionality with the assurance that it won't affect existing code. New functionality will mean adding new classes and methods, but in general it shouldn't mean modifying what has already been written. It is usually solved using polymorphism. Instead of forcing the main class to know how to perform an operation, it delegates the operation to the objects it uses, so it doesn't need to know explicitly how to perform it. These objects will have a common interface that they will implement specifically according to your requirements.

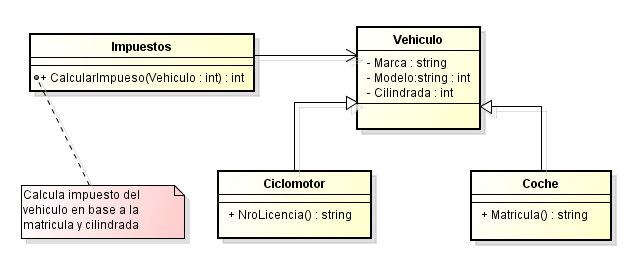


**Liskov Substitution Principle**

It is a principle of object-oriented programming. and can be defined as: Every class that inherits from another can be used as its parent without needing to know the differences between them.

If somewhere in our code we are using a class, and this class is extended, we have to be able to use any of the child classes and the program is still valid.

This forces us to make sure that when we extend a class we are not altering the behavior of the parent.

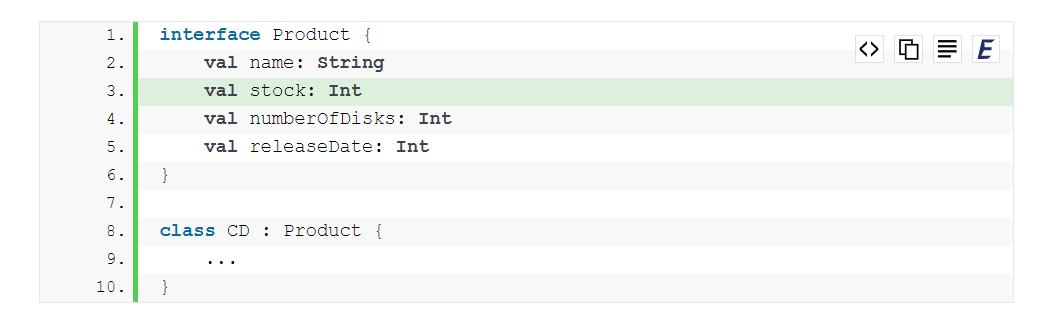
****

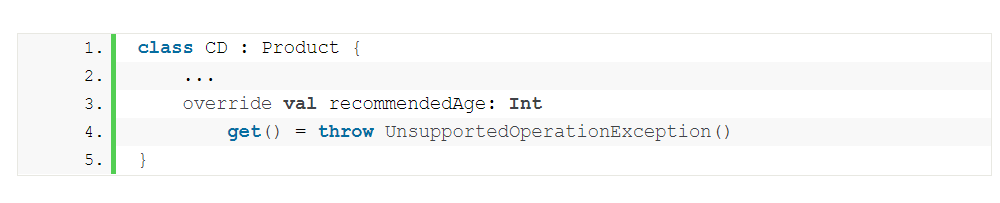
**Interface Segregation Principle**

It helps us not to force any class to implement methods it doesn't use. This will avoid problems that can lead to unexpected errors and unwanted dependencies. It also helps us reuse code more intelligently.

That no class should depend on methods it doesn't use. Therefore, when creating interfaces that define behaviors, it is important to make sure that all classes that implement those interfaces will need and be able to add behaviors to all methods.

Otherwise, it's better to have several smaller interfaces. The module that describes the interface doesn't have to know anything about our code, yet we can work with it just fine.

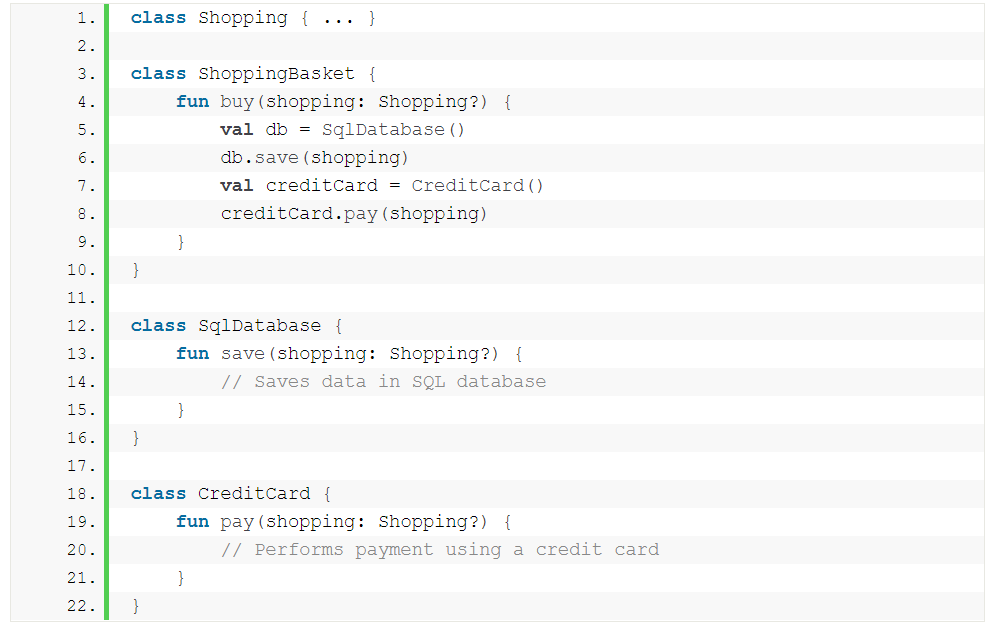


****

**Dependency Inversion Principle**

It forces us to organize our code in a very different way than we are used to, and against what logic initially dictates, but in the long run it compensates for the flexibility it gives to the architecture of our application.

In object-oriented design, the dependency inversion principle is a specific way of decoupling software modules. By following this principle, the conventional dependency relationships established from high-level policy-setting modules to high-level policy-setting dependency modules. low-level modules are inverted, making high-level modules independent of the low-level module's implementation details.

****

35 SANCHEZ MISHQUERO JOSE FRANCISCO

**SOFTWARE ENGINEERING PRINCIPLES PAPER**

Software engineers aim to build quality products on time and within budget

**CORE PRINCIPLES**

· Modularity

· Abstraction

· Encapsulation

COMMON PARADIGMS

· Object orientation

· Aspect orientation

· Functional programming

· Logic programming

· Genetic programming

· Structured program

• LISKOV SUBSTITUTION PRINCIPLE:

Use of specialization, Strong behavior subtyping: both objects can do the same thing, Using conditions for methods, Conservation of supertype invariants, History restriction

Following this principle is achieved:

-A Product be a base class using the "save an object to a file" method.

-Implementation of specializations: Widget of some product fulfills a single purpose or a single function within the system.

-Implementation of any special condition that requires Widget and handling exceptions appropriately

This principle helps increase reusability, extensibility, and maintainability.

• INTERFACE SEGREGATION PRINCIPLE:

An interface is a functional "window" that represents the public methods of a component, without having to declare them. A component can have many interfaces

This principle achieves:

-The user may or may not depend on methods that are not used

-Group the public methods of a component by purpose or responsibility and declared in interfaces, or abstract classes

This principle helps reduce complexity, increases cohesion, coupling, extensibility, reusability, and maintainability.

• DEPENDENCY INVERSION PRINCIPLE:

Organize the system in layers, Abstractions should not depend on details, Implementation details must depend on abstractions

This principle achieves:

Abstractions should not depend on details

High-level modules must not depend on low-level modules, both must depend on abstractions

“Program to abstraction”

This principle helps increase reusability and maintainability.

36 SHUGULI REINOSO ALAN JESITH

**INTRODUCTION TO THE SOLID PRINCIPLES**

These principles were called S.O.L.I.D. for its acronym in English:

* **S: Single responsibility principle**
* **O: Open/closed principle**
* **L: Liskov substitution principle**
* **I: Interface segregation principle**
* **D: Dependency inversion principle**

Applying these principles will make your work much easier, both your own and that of others (it is very likely that your code will end up being read by many other developers throughout its life cycle). Some of the advantages of applying it are:

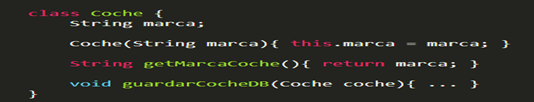
* **Easier and faster code maintenance**
* **It allows adding new functionalities more easily**
* **Promotes greater code reusability and quality, as well as encapsulation**

We are going to see each of these principles in detail, along with basic examples, which, despite not being applicable in the real world, I hope will provide enough clarity for you to be able to understand and apply these principles in your developments.

**S: Single Responsibility Principle**

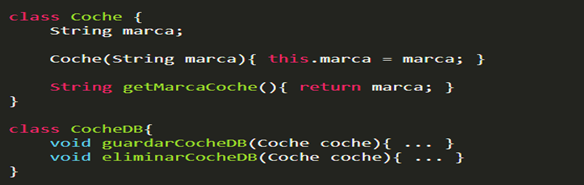
As its name suggests, it establishes that a class, component or microservice must be responsible for only one thing (the much acclaimed term "decoupled" in English). If, on the contrary, a class has several responsibilities, this implies that the change in one responsibility will cause the modification in another responsibility.

Consider this example:



As we can see, the Car class allows both access to the properties of the class and operations, so the class already has more than one responsibility.

To avoid this, we need to separate the responsibilities of the class, so we can create another class that takes care of the operations.

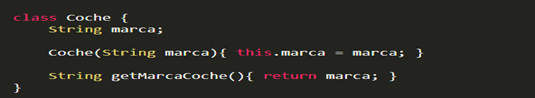


Our program will be much more cohesive and encapsulated by applying this principle.

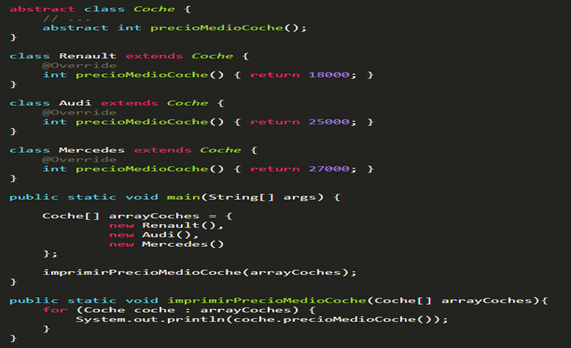
**O: Open/closed principle**

It states that software entities (classes, modules, and functions) should be open for extension, but closed for modification.

If we continue with the Car class:



To comply with this principle, we could do the following:



Each car extends the abstract class Car and implements the abstract method priceAverageCar().

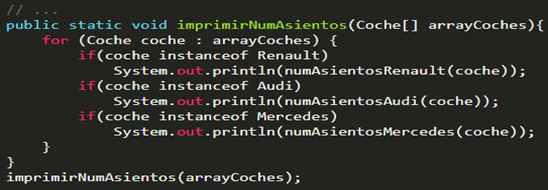
Thus, each car has its own implementation of the method priceAverageCar(), so the method printPriceAverageCar() iterates through the array of cars and only calls the method priceAverageCar().

Now, if we add a new car, AverageCarPrice() will not have to be modified. We will only have to add the new car to the array, thus fulfilling the open/closed principle.

**L: Liskov substitution principle**

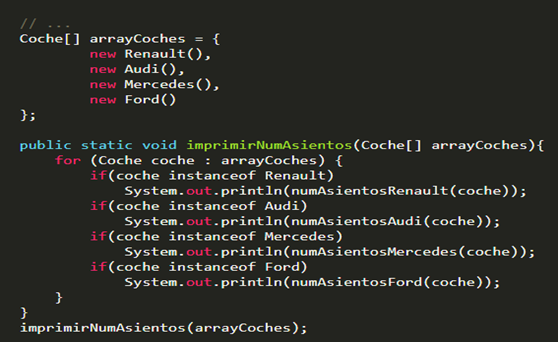
Complying with this principle will confirm that our program has an easy-to-understand class hierarchy and reusable code.

Let's look at an example:



The program must know each type of Car and call its associated numSeats() method.

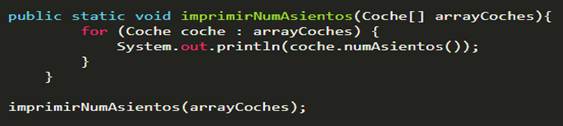
Thus, if we add a new car, the method must be modified to accept it.



For this method to comply with the principle, we will follow these principles:

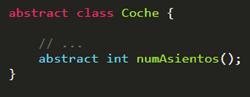
* **If the superclass (Car) has a method that accepts a parameter of the type of the superclass (Car), then its subclass (Renault) should accept either a type of the superclass (Car) or a type of the subclass (Renault) as an argument.**
* **If the superclass returns a type of itself (Car), then its subclass (Renault) should return either a type of the superclass (Car) or a type of the subclass (Renault).**

If we re-implement the above method:

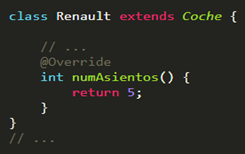


Now the method doesn't care about the type of the class, it just calls the numSeats() method of the superclass. It only knows that the parameter is of type car, either Car or one of its subclasses.

For this, now the Car class must define the new method:



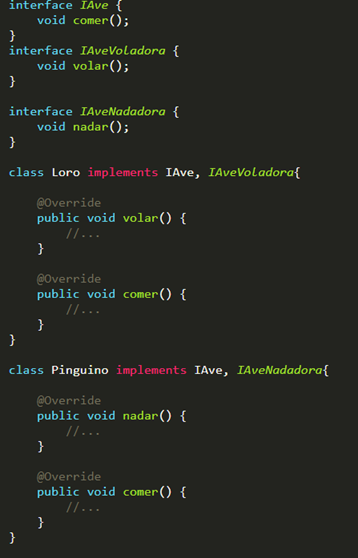
And the subclasses must implement said method:



As we can see, now the printNumSeats() method does not need to know what type of car it is going to perform its logic with, it simply calls the numSeats() method of the Car type, since by contract, a subclass of Car must implement said method.

**I: Interface Segregation Principle**

This principle states that clients should not be forced to rely on interfaces they do not use.



Thus, each class implements the interfaces that it really needs to implement its methods. When it comes to adding new features, this will save us a lot of time, and in addition, we comply with the first principle (Single Responsibility).

**D: Dependency Inversion Principle**

It states that the dependencies must be on the abstractions, not on the concretions. That is to say:

* **High level modules should not depend on low level modules. Both should depend on abstractions.**
* **Abstractions should not depend on details. Details should depend on abstractions.**

At some point our program or application will be made up of many modules. When this happens, it is when we must use dependency injection, which will allow us to control the functionalities from a specific place instead of having them scattered throughout the program. Also, this isolation will allow us to test much more easily.



Thus, both the high-level module and the low-level module depend on abstractions, thus fulfilling the principle of dependency inversion. In addition, this will force us to comply with the Liskov principle, since the types derived from Connection (DatabaseService and APIService) are substitutable for their abstraction (Interface Connection).

**SOFTWARE ENGINEERING GOALS**

The goals of software engineering are straightforward and easy to understand – but they aren’t always easy to meet. This is because there are so many different ways to approach software engineering and so many outcomes that are possible. While we do have best practices and there are standards in place, every software engineer has a different approach and sometimes they don’t always mesh well with other members of an IT team.

* **Software engineers aim to build quality products on time and within budget**
* **Some Desirable Qualities:**
  + understandability
  + testability
  + maintainability
  + efficiency
  + reliability
  + security
  + extensibility
  + openness
  + interoperability
  + reusability

**Software Engineering Principles and Best Practices Examples**

Software engineering best practices ensure that software developed by a delivery team, a contractor, or an individual developer is high quality and efficient. If applications do not meet these best practices, it puts you at risk for outages, security hazards, and failing systems adherence to software engineering best practices help to prevent that.

**Iterative Development**

An important development methodology best practice in software engineering is iterative development. Iterative development ensures that software flaws or risks are resolved before there has been a lot of time and effort put into the software. This development approach enables continuous testing and continuous integration which creates the opportunity for early feedback so that changes can be made swiftly.

**Service-Based Architecture and Microservices**

Service-Based Architecture and Microservices are some of the most critical best practices in software engineering today. Service-based architecture is a software design best practice where services are provided to other components by application components, through communication protocols. A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently. It has four properties:

* **Logically represents a business activity with a specified outcome.**
* **Self-contained.**
* **Black box for its consumers.**
* **May consist of additional underlying services.**

**Software Modelling**

Using visual modeling tools helps to improve the ability to manage software, rationalize, and maintain that software. It will also help to keep information among the team easy to understand and constant – which is especially important in agile development when multiple teams are working laterally to develop the same software or updating the same application portfolio.

**Software Testing**

Software Testing is another critical element of software engineering best practices and principles. A team wants to verify that all software developed is high quality and meets the requirements set forward in the planning stage.

Remember that changing software later in development is much, much costlier. Continuous testing from the start of development will help to avoid costly repairs later on or even after deployment. Software engineering practices that do not include testing will eventually fail – but not after costing a lot of time and money from your budget.

**Software Engineering Practices and Theory**

Software engineering theory and practice meld together computer science with artistry and design. It is a fine line to walk – software that is too “pretty” but doesn’t function isn’t effective, but software that isn’t written well can be difficult as well.

Abstraction is a theory in both art and software engineering. Abstraction is the simplification of a description into the bare-bones essentials. In software engineering theory, this means making code easy enough to read and edit. Elements have to be well named so that they are descriptive – typically only using a combination of verbs and nouns. It needs to be easy enough to understand that someone who has never looked at the code before will understand.

It also has to be accurate – you want to reflect the work that is complete or the action carried out by the string of code – nothing more, nothing less. Brevity should also be clear.

Software engineering theory also calls for separation of the code. One needs to treat what the software does and how the software does it independently. This does make it longer but helps with the clarity that is so essential in software engineering that will be maintained for a longer period of time.

**THREE CORE PRINCIPLES**

**Modularity**

Modularity exists in a software system when it is composed of loosely coupled and cohesive components that isolate each significant or modifiable component. design decision on a component and make sure related ideas are as similar as possible. Modularity can improve understandability, testability, maintainability, security and reusability.

**Abstraction**

For each component, there is an explicit and clear declaration of the component's functions. Exposed features and functionality should be no more or less than what other components may need. The abstraction principle can improve understandability, testability, maintainability, and reusability. It can also allow developers to follow modularity more effectively.

**Encapsulation**

As can be seen from the diagrams, the object variables are located in the center or core of the object. The methods surround and hide the core object from other objects in the program. 'To the packaging of the variables of an object with the protection of its methods is called encapsulation. Typically, the encapsulation is used to hide unimportant implementation details from other objects. So the implementation details can change at any time without affecting other parts of the project program.

**Ways to encapsulate**

1. **Standard (Default)**
2. **Open: Makes the class member accessible from outside the Class and anywhere in the Class program.**
3. **Protected: It is only accessible from the Class and the classes that inherit it (at any level).**
4. **Semi closed: It is only accessible from the inherited class.**
5. **Closed: It is only accessible from the Class.**

In encapsulation there are analyzers that can be semantic and syntactic.

37 SIMBAÑA SIMBAÑA JONATHAN GUSTAVO

**THE SOLID PRINCIPLES**

OVERVIEW OF THE SOLID PRINCIPLES

SOLID is a mnemonic acronym for five principles

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Some argue that these are the ”first five” principles • That claim has not been justified or widely accepted • However, whether they are the “first five” principles is not very important • Following these principles can help ensure quality software, primarily from a developers’ perspective

DESIGN PROBLEM

You’ve been contracted to build a Maze Generator

The program must be able to print the mazes using ASCII character or draw them in an image

SINGLE RESPONSIBILITY PRINCIPLE

This principle is vey closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose

This principle is also related to the principles of

• Localization of design decisions

• Encapsulation

Following this principle can help

• Increase Reuse and Maintainability

• Reduce Complexity, even though the number of classes might increase

OPEN/CLOSED PRINCIPLE

Core Ideas: Software entities (e.g., classes, generics) should be open for extension but closed to modification

Original definitions: A class is open if it is still available for extension , A class is closed if it is available for use by other class, and therefore should not be modified

Revised definitions:

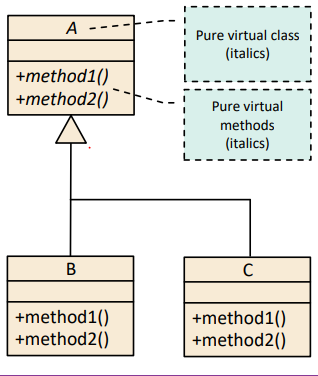
A system of classes is open for extension and closed for modification, if

Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes (in Java)

Implementations or concrete classes inherit the public method declarations from the interfaces, abstract classes, or pure virtual classes

Users

INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES



OPEN/CLOSED PRINCIPLE

Ways to achieve the open/closed principle

Inheritance

Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes

The public methods of one class can be grouped into multiple abstractions

Each abstraction should focus on a single purpose, as per the Single Responsibility Principle

Have concrete classes inherit from these abstraction

Java does not support multiple inheritance, so a class can have multiple base classes

38 TAPIA ALBAN ANDREA JULIANNA

The solid principles are:

· Single responsibility principle: Each class should be responsible for its part in the system functionality. The class `s responsibility should be encapsulated by the class; its properties should be aligned with the responsibility. This principle it`s related with cohesion, that means that responsibilities are aligned and focused on a unique purpose.

· Open/closed principle: Software should be open for extension but closed to modification. The class extends to other projects without modifications but could be reused. If the programmer does modifications, the principle is broken.

· Liskov substitution principle: Objects of a superclass must be able to be replaced with objects of its subclass without interrupting the application.

· Interface segregation principle: No client should be forced to depend on methods that it doesn´t use. The public methods can be reunited by purpose or responsibility.

· Dependency inversion principle: Organize the system into layers, some reusable libraries will be abstract or detail oriented. The components from the reusable layers do not depend on detail layers. Furthermore, they depend on the abstraction that the detailed components implement. Abstractions should not depend on details and implementation should depend on abstractions.

39 TAYO RUIZ SEBASTIAN ALEJANDRO

**Introduction to the SOLID Principles**

SOLID is an acronym for:

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

**SINGLE RESPONSIBILITY PRINCIPLE**

Every class should be working on only one thing of the program, and they have to be encapsulated

**OPEN/CLOSED PRINCIPLE**

Software entities should be opened for extensions but closed to modification, this means that a class is opened if it is available for extensions, if it’s closed if it's available for use of other classes.

We have the abstract classes as an example of an Open/Closed Principle, also we can achieve the open/close principle with: Inheritance, Aggregation and parametrization.

**LISKOV SUBSTITUTION PRINCIPLE**

For example, if s is a specialization of T, then an S object can be used wherever a T object is required.

Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file, this handles exceptions appropriately

Following the Liskov Substitution Principle can help developers:

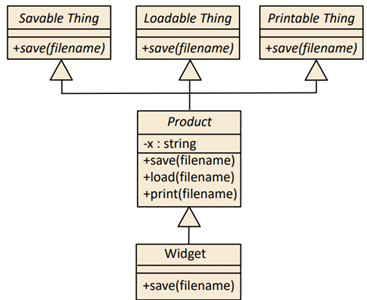
• Increase Reuse

• Increase Extensibility

• Increase Maintainability

**INTERFACE SEGREGATION PRINCIPLE**

One class can be implemented to more classes



This method reduces complexity by increasing Cohesion, increasing extensibility, increasing reuse and increasing maintainability.

**DEPENDENCY INVERSION PRINCIPLE**

The Inversion Principle organizes the system into layers, like reusable libraries. Components from the abstract layers should not depend on components from the detail layers.

How to apply the Dependency Inversion Principle

• Abstraction should not depend on details

• Both low-level and high-level modules should depend on abstractions

The dependency Inversion Principle Increases reusability and maintainability

**Software Engineering Goals**

Software engineers aim to build quality products on time and within budget

For this reason, we have the Three core Principles: Modularity, Abstraction, Encapsulation.

**Problem background**

Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees. In some cases, there are conflicting definitions within the same paradigm.

**Core Problem**

There are no general, unifying definitions, especially for multiparadigm software development.

The principles are hard to teach and Programmers often don’t understand the core principles, and therefore don’t benefit from their guidance, especially in multi-paradigm software development.

In other words, a principle is a foundational concept (truth, proposition, rule, etc.) that leads to and supports reasoning about desirable characteristics, such as maintainability, efficiency, openness, reusability, etc.

**BEST PRACTICES, PATTERNS, AND IDIOMS**

Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level.

**OBSERVATIONS RELATIVE TO MODULARITY**

Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways. The other four SOLID principles overlap to some degree with the basic idea of Low Coupling are actually specific best practices for achieving Low Coupling in certain contexts.

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

For FP, the components are primarily functions, but could also include other artifacts like build scripts. By definition, a pure function in FP only depends on values that are passed in as input parameters.

Every interesting or potentially changeable design decision needs to be localized. This is done by defining a predicate and set of rules for each design decision.

**OBSERVATIONS RELATIVE TO ABSTRACTION**

Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts. For each component, there is an explicit and clear declaration of the component’s accessible features or functionality. Depending on the paradigm and programming language.

Leaky abstraction – other components end up relying on details not explicitly stated in the abstraction.

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION**

Ensure that the private implementation details of a component are insulated so they cannot be accessed or modified by other components. Doing so will lead to better testability, maintainability, and reliability.

Practices and Criteria:

• Conceptual barriers

• Programmatic barriers

• Usage barriers

**NON-REDUNDANCY AND COMPLIMENTARY – CRITERION**

• Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different.

• We show satisfaction of the second criteria, namely that developers choose to follow each principle independent, with an example consisting of four functional-identical code snippets.

It’s very important to clarify the purpose of Software Engineering principles, so we don’t make mistakes applying the concepts when we are coding.

40 TECA TELLO CAMILA MILENA

**INTRODUCTION TO THE SOLID PRINCIPLES**

Following these principles can help ensure quality software

**• SINGLE RESPONSIBILITY PRINCIPLE:**

The class’s responsibility must be encapsulated and every class must be responsible for only one part of the functioning of the system. This principle is related to the principle of cohesion (must focus on a single purpose) and is related to encapsulation.

This principle helps to reduce complexity, apply reusability and maintainability.

**• OPEN/CLOSED PRINCIPLE:**

A system of classes is open if it is available for extension and closed if it is available for use by another class, that is, it is closed for modification.

**This principle is achieved:**

-Inheritance, Aggregation, Parameterization, Encapsulation

-Public methods in your interfaces or abstract classes

-Group public methods into multiple abstractions

-Each abstraction must focus on a single purpose.

-Concrete classes inherit from these abstractions

-Technique "strategy pattern"

This principle helps reduce complexity and increase extensibility.

**• LISKOV SUBSTITUTION PRINCIPLE:**

Use of specialization, Strong behavior subtyping: both objects can do the same thing, Using conditions for methods, Conservation of supertype invariants, History restriction

**Following this principle is achieved:**

-A Product be a base class using the "save an object to a file" method.

-Implementation of specializations: Widget of some product fulfills a single purpose or a single function within the system.

-Implementation of any special condition that requires Widget and handling exceptions appropriately

This principle helps increase reusability, extensibility, and maintainability.

**• INTERFACE SEGREGATION PRINCIPLE:**

An interface is a functional "window" that represents the public methods of a component, without having to declare them. A component can have many interfaces

**This principle achieves:**

-The user may or may not depend on methods that are not used

-Group the public methods of a component by purpose or responsibility and declared in interfaces, or abstract classes

This principle helps reduce complexity, increases cohesion, coupling, extensibility, reusability, and maintainability.

**• DEPENDENCY INVERSION PRINCIPLE:**

Organize the system in layers, Abstractions should not depend on details, Implementation details must depend on abstractions

**This principle achieves:**

Abstractions should not depend on details

High-level modules must not depend on low-level modules, both must depend on abstractions

“Program to abstraction”

This principle helps increase reusability and maintainability.

**UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION**

Modularity, abstraction and encapsulation have value in all common software development paradigms, these paradigms are:

Object orientation (OO), Aspect orientation (AO), Functional programming (FP), Logic programming (LP), Genetic programming (GP), Structured program (SP)

**Purpose of principles:**

The principles will be a rule that must be followed to build quality products on time and within budget

**Use of the best practices, patterns, and idioms:**

They are a support to the developers helping them to follow the principles.

-Best practices: they are techniques that help to fulfill the principles, without considering the theoretical details.

-Patterns: exemplify the principles

-The idioms: they are techniques or solutions to express a certain data structure in a specific programming language, in a coherent way.

There are three core principles:

**• MODULARITY:**

**Observations relative to modularity:**

-Location of design decisions.

-Design decisions must be “hidden” from users: encapsulation

-Minimize the effects of software: use of coupling and cohesion,

**Some concepts of modularization:**

- Grady Booch said that modularization is “To build modules that are cohesive

-Martin Fowler and other experts believe that modularity will prevent common code smells (long method, large class, long parameter list)

-G. Kiczales and others say that a developer should seek to understand it well enough to make changes.

**Paradigm-independent definition for modularity:**

Modularity exists in a software system when it is made up of loosely coupled, cohesive components that isolate every major or modifiable design decision in one component.

Modularity can improve understandability, testability, maintainability, reliability, security, extensibility, and reusability. It can also help during the software development process by delineating loosely coupled units of work.

**Practices and Criteria:**

Localization of design decisions, Low Coupling, High Cohesion, Modular Reasoning

**•ABSTRACTION:**

**Observations relative to abstraction:**

Abstraction is the act of saving important information while others are suppressed. Software abstraction requires a lot of work where developers examine large and various collections in detail, to determine the most outstanding

**Common problems with abstraction:**

-Leaky abstraction and excessive abstraction

-Insufficient control given to users of the abstraction

-Inadequate access to the information contained in the abstraction

**Paradigm-independent definition for abstraction:**

The explicit declaration of each component may or may not be part of the code or documentation, where its functions or features are accessible. Adherence to the principle of abstraction can improve understandability, testability, maintainability, and reusability. It can also allow developers to follow modularity more effectively,

**Practices and Criteria:**

Meaningful labels and identifiers, Context-aware labels and identifiers, Abstraction completeness, Abstraction sufficiency

**•ENCAPSULATION:**

**Observations relative to encapsulation:**

The encapsulation is not unique to OO, exist three categories for encapsulation:

-The bundling of data with operations

-The hiding decisions behind logical barriers

-The organization of components to minimize ripple effect

**Paradigm-independent definition for encapsulation:**

The private implementation of a component must be isolated so that no component can access it.

Encapsulation will lead to better testability, maintainability, and reliability.

**Practices and Criteria:**

Conceptual barriers, Programmatic barriers, Usage barriers

41 TERAN FLORES MELANIE ELIZABETH

**SOLID PRINCIPLES**

1. **S**ingle Responsibility Principle

Every class should be responsible for a single part of the system’s functionality, a class’s responsibility should be entirely encapsulated by the class and the properties should be narrowly aligned with that responsibility. This is closely related to the principle of Cohesion, localization of design decisions and encapsulation.

**2. O**pen/Closed Principle

Software entities should be open for extension but closed to modification. A class is open if it is still available for extension and is closed if it is available for use by other class.

A system of classes is open for extension and closed for modification, if public methods are declared using interfaces, or abstract classes, Implementations or concrete classes inherit the public method declarations from the interfaces, abstract classes, or pure virtual classes

**3. L**iskov Substitution Principle

Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application (objects of subclasses behave in the same way as objects of superclass). This can help developers to Increase Reuse, Increase Extensibility, Increase Maintainability.

**4. I**nterface Segregation Principle

No user of a component should be forced to depend on methods that it does not use

The public methods of a component can be grouped by purpose or responsibility as captured and declared in interfaces, or abstract classes.

**5. D**ependency Inversion Principle

Organize the system into layers, components from the abstract layers should depend on abstractions that the detailed components implement. Abstractions should not depend on details, implementation details should depend on abstractions

**CORE PRINCIPLES**

6. Modularity

Modularity exists in a software system when it is comprised of loosely coupled (the degree to which components depend on each other) and cohesive (the degree to which the properties of a component relate to the component’s primary responsibility) components that isolate each significant or changeable design decision in one component and ensure that related ideas are as close as possible. Modularity can improve understandability, testability, maintainability, reliability, security, extensibility, and reuse. It can also help with collaboration during the software development process by outlining loosely coupled work units

7. Abstraction

For each component, there is an explicit and clear declaration of the component’s accessible features or functionality. The exposed features and functionality should be no more and no less than what other components may need or depend on.

Adherence to the abstraction principle can improve understandability, testability, maintainability, and reusability. It can also allow developers to follow modularity more effectively, because it will bring to light weakness with localization of design decisions, unnecessary coupling, and low cohesion.

8. Encapsulation

Ensure that the private implementation details of a component are insulated so they cannot be accessed or modified by other components. Doing so will lead to better testability, maintainability, and reliability. It will also help with a clear separation of concerns and avoid accidental coupling.

42 VILLEGAS ESTRELLA SALMA ABIGAIL

**INTRODUCTION TO THE SOLID PRINCIPLES**

SOLID is a mnemonic acronym for five principles: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle

• Some argue that these are the” first five” principles

• That claim has not been justified or widely accepted

• However, whether they are the “first five” principles is not very important

• Following these principles can help ensure quality software, primarily from a developers’ perspective

**SINGLE RESPONSIBILITY PRINCIPLE**

Every class should be responsible for a single part of the system’s functionality, a class’s responsibility should be entirely encapsulated by the class, a class’s properties should be narrowly aligned with that responsibility

This principle is very closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose

Also related to the principles of: Localization of design decisions, Encapsulation

**OPEN/CLOSED PRINCIPLE**

Software entities (e.g., classes, generics) should be open for extension but closed to modification

A class is open if it is still available for extension, a class is closed

Ways to achieve the open/closed principle

• Inheritance: Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes. Java does not support multiple inheritance, so a class can have multiple base classes

• Aggregation: Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

• Parameterization: Use a generic to capture a template solution and instantiate it with the specific data types

**INTERFACE SEGREGATION PRINCIPLE**

An interface is a” window” or “portal” into the functionality of a component, an interface represents public methods of a component, an interface doesn’t have to declare all of the possible public methods of a component; a component can have many interfaces, Java does support interfaces directly.

**DEPENDENCY INVERSION PRINCIPLE**

Organize the system into layers: some layers, like reusable libraries or frameworks will be more abstract or policy-setting layer, others will be detail oriented

Components from the abstract layers should not depend on components from the detail layers; instead, they should depend on abstractions that the detailed components implement, abstractions should not depend on details

Implementation details should depend on abstractions

**THREE CORE PRINCIPLES**

**Abstraction**

The principle of abstraction can improve understandability, testability, maintainability, and reusability. It can also allow developers to track modules more effectively, as it will highlight weaknesses with localization of design decisions, unnecessary coupling, and weak linkage.

For each component, have a clear and unambiguous statement explicit about the component's accessible features or functionality. The features and functionality exhibited should be no more or less than what other components may need or depend on.

**Encapsulation**

Ensures that a component's private implementation details are isolated so that they cannot be viewed or modified by other components. This will lead to better testability, maintainability and reliability. It will also help to clearly separate concerns and avoid random pairing.

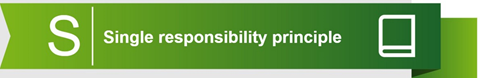
**Modularity**

Appears in a software system when it is loosely coupled (the extent to which components depend on each other) and cohesive (the extent to which properties of a component that are related to the main responsibility of the component) components that separate each important or mutable design decision within a component and ensure that ideas are closely related tight as possible.

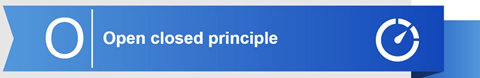
43 ZEAS CLAVIJO JOEL ALEXANDER

**Introduction to the SOLID principles**

To achieve quality software and guarantee its correct operation, the following points can be taken into account that help us to carry out this:

****

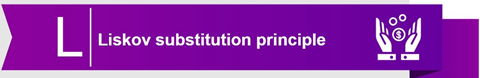
The class must be encapsulated and each class must be responsible for only part of the system operation. This principle is related to the principle of cohesion, which focuses on a single function and is related to encapsulation.

****

A class system is open if it is available for extension and closed if it is available for use by another class, for modification. This principle helps reduce complexity and increase extensibility. Thanks to this you can get:

-Each abstraction should focus on a single purpose.

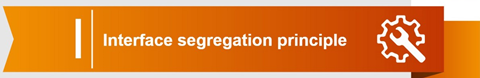
-Concrete classes inherit from these abstractions.

****

It is used in OOP and says that each class that inherits from another can be used as its parent without needing to know the differences between them, resulting in:

-A Product is a base class using the "save an object to a file" method.

-Implementation of specializations: Widget of some product fulfills a single purpose or a single function within the system.



Avoid the implementation of methods to classes that do not need it, to avoid problems that can lead to unexpected errors and unwanted dependencies and thus code can be reused and improvements implemented.

****

This principle helps us understand that high-level modules should not depend on low-level modules, since both should depend on abstractions. The abstractions must not depend on the details, but the details must depend on the abstractions.

**Modularity, abstraction and encapsulation**

These mentioned methods help to improve the functionality, development and understanding of a code, among them it can be highlighted that:

**Modularity** is the ability to ignore the details of the parts in order to focus attention on a higher level of a problem.

**Abstraction** happens when something is wrapped in a protective layer or shielded from anything that might harm it.

**Encapsulation** is the act of packing or protecting data or attributes with methods.