The SOLID principles

Programmation Java

At first the objects concepts are not easy to grasp. It is easier to reason with linear and concrete way that the abstract and with inheritance and polymorphism concepts. This requires a step back and a capacity for abstraction that is not totally intuitive and more difficult accessible.La object-oriented design offers unique mechanisms of representation of our environment combining data and behavior, declining generic concepts into multiple specialized .In addition this ability to represent things, object-oriented design also offers tolerance mechanisms to powerful change. This tolerance is a key factor in software development, maintenance, often requiring increased efforts over time and the evolution of how to characterize logiciel.Mais intolerance change? Here are the symptoms: rigidity, "Every change causes a cascade of changes in dependent modules. "A simple procedure at first appears to be a nightmare, like a ball of yarn that has no fin.La fragility" Trend software to break in several places at each change. ". The difference with the rigidity lies in the fact that interventions on the code can affect the modules having no relation with the modified code. Any intervention is subject to a possible change of behavior in another module.L'immobilisme: "Software Inability to be reused by other projects or parts of itself. "-" It is almost impossible to reuse the interesting parts of the software. "The effort and the risks are too importants.La viscosity:" It is easier to make a workaround rather than respect the design that has been thought. "When we work on existing code, we often have several opportunities implementations. We will find solutions conserving, improving the code and other design that will "break" this design. When the latter category is simpler to implement, we believe that the code is visqueux.L'opacité is the readability and simplicity of understanding of the code. The most common situation is a code that does not express its primary function. Thus, a code is more difficult to read and understand, and we consider that it is opaque.Dans the Agile Software Development book Pinciples, Patterns and Practices Robert C. Martin condensed, in 2002, five fundamental principles design, responding to this scalability problem, under the acronym SOLID: - Single Open responsibility principle- close principle- Liskov principle- Interface segregation principle-

Dependency inversion principleSingle responsibilty Principle "shoulds A class-have one reason to change. "As the name suggests, this principle means that a class should have one and only one responsibility. Why you say? If a class has more than one responsibility, the latter will find themselves linked. Changes to liability will impact the other, increasing the rigidity and fragility of code.A number of techniques can help us apply the principle of SRP. Among them we find the CRC cards - Class Responsibility Collaboration.Open Closed Principle "Classes, methods shoulds be open for extension, closed for modification purpose. "A class, a method, a module must be extended, support different implementations (Open for extension) without it having to be changed (closed for modification) .The conditional instantiation in a constructor are good examples of non-compliance with this principle . A new implementation will impact the addition of a condition in the méthode.Liskov Substitution Principle "Subtypes must be substitutable for Their basic kinds. "The subclasses must be substituted in their base class without altering the behavior of its users. In other words, a user of the base class must be able to continue to function properly if a class derived from the main class is provided to him place.Cela means, among others, that we should not raise unexpected exception (as UnsupportedOperationException for example), or change the values ??of attributes of the main class of an improper manner, not respecting the contract defined by méthode.Les for violation of the principle of Liskov are not so frequent in reality and we design in General models that do not violate this principle. However, this can occur by precipitation or lack of knowledge of the implementation details of the base classes and its detection is mostly difficult. I encourage you to remain vigilant when deciding to extend a classe.Interface Segregation Principle "Clients shoulds not be forced to depend on methods That They Do not use. "The aim of this principle is to use interfaces to define contracts, sets of features to meet a functional need, rather than simply provide abstraction to our classes. This results in a reduction of the coupling, customers only dependent on the services they Interface systematic utilisent.L'utilisation type IMaClasse taking public methods of the MyClass class is therefore not a good practice because it links our contracts to their implementation, making delicate reuse and refactorings to venir.Une caution though: one through this principle can be multiplied interfaces. Taking this idea to the extreme, we can imagine an interface with a client method. Of course, experience, pragmatism and common sense are our best allies in this domaine.Dependency Inversion Principle "High level modules shoulds not depend on low level modules. Both shoulds depend on abstractions. "" Abstractions shoulds not depend on details. Details shoulds depend on abstractions. "Let's look at the important concept of this principle: Inversion. The principle of DIP states that high-level modules should not depend on low-level modules. But why? To answer this question, consider the definition in reverse: the high level modules depend on low-level modules. Typically the top level modules contain heart - business - applications. When these modules are dependent on lower level modules, changes in the "low level" modules can affect the "high level" modules and "force" to apply changes.