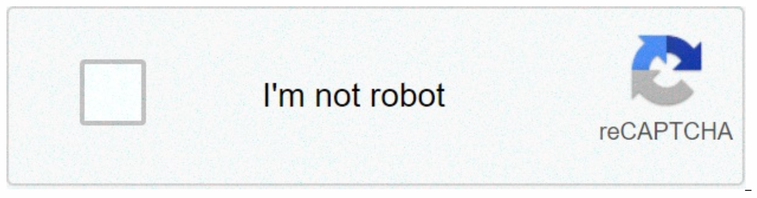

Continue

Strategy pattern java uml

Bject-oriented programming is a rotating programming paradigm, using objects and the Classes declaration to provide simple and reusable design to our program. According to Wikipedia:Object-oriented programming (OOP) is a programming paradigm based on the concept of objects that can contain data in the form of fields, commonly known as attributes; and code, usually in the form of procedures known as methods. Wow!! beautiful but not all OOP, which is important for the structure of classes and relationships. Complex systems such as brains, cities, anthills, buildings are full of patterns. It is built with a well-structured architecture to achieve a long-lasting situation. And software development is not left out. Designing a large application requires a complex and complex connection and collaboration of objects and data. OOP provides the design to do this, but as I said before you need a pattern to achieve a long-lasting one. Problems may arise in our application of OOP design, which can lead to decay. Therefore, these issues are catalogued overtime and elegant solutions for each are defined by experienced early Software Developers. These solutions are Patterns.To as design and date, there are 24 design patterns as described in the original book, Design Patterns: Reusable Object Oriented Software Elements. Each of these patterns provides a set of solutions to a specific problem. In this article, we will look at the Strategy Pattern to understand how it works and how and when it will be applied during software development. Tip: Keep your code DRY with reusable components. Use Bit to share and explore components, upload them to your projects, and sync them. Take a look. UI components that make it reusable with the BitStrategy Pattern: The Basic IdeaStrategy Design Pattern is a type of behavioral design pattern that encapsulates a family of algorithms and selects one of the pools for use during uptime. Algorithms can be modified, meaning they can be substituted for each other. The strategy pattern is a behavioral design pattern that allows you to select an algorithm at run time — the key idea of WikipediaThe is to create objects that represent various strategies. These objects create a strategy pool that the context object can select to change its behavior based on its strategy. These objects (strategies) perform the same operation, have the same (single) job, and form the same interface strategy. Take, for example, the sorting algorithms we have. Sorting algorithms have a unique set of rules that they follow to effectively sort a series of numbers. We have several names in bubble sortlinear SearchHeap SortMerge SortSelection Sorting. Next, in our program, we need different sorting algorithms at a time during execution. Using SP, algorithms and from the pool it's more like an add-in like PlugnPlay or Device Drivers in Windows. All add-ons must be a signature or rule. For example, a Device Driver can be anything, Battery Drive, Disk Drive, Keyboard Driver ... They must implement:Each driver must implement the above functions, DriverEntry is used by the operating system when installing a driver, DriverUnload when the drive is out of memory, AddDriver to add drivers to the driver list. Your operating system driver doesn't need to know what it's doing, it knows that since you call it all a driver, it will assume that all this is available and call them at the required time. If we add sorting algorithms to a class, we find ourselves writing conditional expressions to select an algorithm. Most importantly, all strategies must have the same signature. If you are using an OO-Language, make sure that you inherit strategies from a public interface, and if you are using a non-OO-Language method, such as JavaScript, make sure that strategies are a common method that should be called by content. Quite flat, there is an interface that must comply with all sorting algorithms. The Sort Program receives a Sort Strategy as my money in its runSort and calls the sorting method. Any concrete implementation of the SortIng Strategy should apply the sorting method. As you can see, SP supports SOLID principles and pushes us to comply. D at SOLID says we should be connected to abstractions, not conkresions. That's what happened with the RunSort method. Also, if we had taken an alternative to subclassify our sorting algorithms, which said that entities should be open in reach, not extension, we would eventually encounter code that is difficult to understand and maintain, because we will have many classes related to the difference in the algorithms they carry. I, we have a special interface for the implementation of concrete strategy. You will have to run sorting to sort any sorting algorithm, because it is not just fake specific to the job :). S, all classes that implement the strategy have only one job in the ranking. L, all subclasses of concrete strategies can be substituted for their superclasses. So we really see, you can choose algorithms at run time using SP, and this helps us create expandable frameworks. In the above way, the Build context class depends on strategy. During execution or runtime, strategies with different strategy types are passed to the Context class. The strategy provides the template that strategies must follow for implementation. In the UML class diagram above, an abstraction of the concrete class depends on the Strategy interface. It doesn't apply the algorithm directly. From runStraegy in his own method the context strategy concretion doAlgorithm calls him passed. The context class is method-independent and does not know how to implement the doAlgorithm method and you don't need to. you don't need to. By contract design virtue, you must implement the doAlgorithm method of the class that implements the strategy interface. There are three main assets in the strategy design pattern: Context, Strategy, and ConcreteStrategy.The Context is the body that creates the concrete strategies in which they play their roles. A strategy is a template that defines how all startups should be configured. ConcreteStrategy Strategy template (interface) application. ExamplesSteve Fenton sample car wash program, car wash driver know that you can run different classes of washing and cleaning as paraconnected, higher level of washing than more money. Let's The Car Wash offers: Basic Wheel and Body WashExecutive Wheel and Body wash, and body cleansing is just normal soaping and rinsing for the body and brushing for the car. Manager cleaning goes beyond that, polishing the body and wheel to make it look shiny and then drying. Cleaning depends on the level paid by the driver. Level 1 gives basic cleaning for both the body and the wheels:See now, some pattern appears. Under many circumstances, we re-use the same class, classes are related, but they differ in behavior. Besides, our code is messy and heavy. Most importantly, this program fails the Closed Policy in S.O.L.I.D policies, which state that modules must be open for extension, not modification. For each new level of washing, another conditional is added, this change' s. Using the strategy pattern, any responsibility for our water level knowledge will have to relieve our CarWash program. To do this, we have to separate the cleaning movements. First, we create an interface that all actions must implement:Then all cleaning strategies:Then, I tap the CarWash program:Now, we move on to CarWashProgram any cleaning strategy we want. Another example: If we have an authentication strategy, we have an app, so we want to make it secure to add authentication. There are different auth schemes and strategies: we can try to implement something like this: conditional same old long chain. Also, auth for a specific route in our program. If you want, we will find ourselves with the same thing. If we implement the strategy design model here, we create an interface that all auth strategies must implement:AuthStrategy defines the template that all strategies must build on. Any concrete auth strategy authentication style should apply ni to the auth method to provide us. We have tangible strategies for Auth0, Basic and OpenID. Next, we need to tap our AuthProgram class:Now you see, the authentication method does not carry the long key case. The usage method adjusts the authentication strategy to be used, and the authentication method calls only the auth method. AuthStrategy's Pattern: Problems This SolvesStrategic Pattern prevents constant wiring of all algorithms into the program. This makes our program complex and much more fake and difficult to refactor/protect and understand. This lets them include algorithms that they do not use our program. Let's say we have a Printer class that prints in different flavors and style. If we include all the styles and flavors of printing in the printer class:OR As you can see, we end up with a fake class that is hard to read, maintain, and have too many conditionals. However, with Strategy Pattern, we separate print styles into different tasks. Therefore, instead of many conditionals, each condition is moved to a separate strategy class. The printer class doesn't need to know how to apply different print styles. Strategy Pattern and SOLID Principles Strategy Pattern is used on compositional heredity. More abstraction program is recommended for these consements. You see that the Strategy Pattern is compatible with SOLID policies. As an example, there is a DoorProgram with different styles of locking mechanism to lock doors. As different locking mechanisms change between the lower classes of the door. We may want to apply the door locking mechanism to the door class as follows: It seems ok, but the behavior of the doors is different. Each has its own locking and opening mechanism. That's different behavior. When we create different types of doors:And when we try to implement the opening/locking mechanism, you see that we need to call the main method before implementing its own open/lock mechanism. By making the door such an interface:You see that open/lock behavior should be declared in every class or model or Door.Quite, but there are many drawers that will arise as our application grows. The door model must have an open/lock mech. Does a door need to open/close? No, the door never even closes. That's why we see that our door models will have to open/lock. Then, the interface doesn't draw a line between using the interface as a model or as an open/lock mech. Note: In SOLID, a class in S must have a responsibility. A Glass Door is also a wooden door, a Metal Door, a Ceramic Door (do they exist?) Another class must be responsible for handling the on/off mechanism. Using SP, we classas the related, in this case lock/open mech. Then at run time, switch the lock/open mechanism door model to use. The door model can choose from a pool of lock/open strategies that lock/open the mech. Each open/lock strategy is defined in a class inherited from the base interface. SP supports this because it is better to code an interface to adapt highly. Then, our Door models have a subclass of each Door class. We have a Door Adapter whose job it is to open doors. For. We've created objects for several Door models and identified lock/open strategies. The glass door will be locked/opened with retinal scanning and the metal door has a keypad to enter the secret password. What we have achieved here is the separation of concerns, the separation of relevant behavior. Every Door does not know the models and the month of concern to implement a specific locking/opening strategy was delegated for another entity. We programmed it into an interface required by SP, as it makes it easier to change strategy during runtime. This may not hold for a long time but a better approach to courtesy of Strategy Pattern.A Door can be many lock/open strategies courtesy and you can use it during locking and opening one or all. Whatever you do, keep the Strategy Model in mind. JavaScript Strategy

Pattern is based on most of our examples OOP languages. JS is not written statically, it is written dynamically. In other words, the concept of OOP such as interface, polymorphism, encapsulation, delegation does not exist. But in SP, we can assume they're there, we simulate them. Let's use our first example to show how we can implement SP in JS. The first sample was based on sorting algorithms. Now, the interface sortingstrategy has a method sorting that should define all application strategies. The SortProgram class takes a Sort Strategy in the runSort method and calls the sort method. We model our sorting algorithms: there was no interface, but we did it still. There may be a better, more robust way, but for now, that should be enough. The thing here is that in my mind we have a sorting method to do sorting for every sorting strategy you want to implement. Strategy Pattern: When to Use a Strategy Pattern should be used when you start noticing duplicate algorithms, but in different variations. In this way, you need to classially class out algorithms and feed them according to what you want in your program. Then, if you notice duplicate conditional expressions around a related algorithm. When most of your classes have relevant behavior. It'll be time to take them to classes. Advantages of ConcernsSeeing: Related behaviors and algorithms are divided into classes and strategies. Always program interfaces because strategies at run time are switched to the system. Elimination of fake and conditionally infested code. Easy maintenance and refactoring. Selection of algorithms to use. The Strategy Pattern is one of many Design Patterns in the software development pattern. In this post, we saw many examples of how to use SP and then, we saw the benefits and disadvantages. You do not need to apply a design pattern as described. You have to understand thoroughly and know when to apply it. And if you don't understand, don't worry, keep referring to understandings over and over again. You'll get used to it, and eventually. Benefits. Next, in our series, we will be looking into the Template Method Design Pattern so we will continue to follow us :) If you have any questions that you need to add this or something, correctly or remove it, do not hesitate to comment, email or DM me. Thank you for reading it! 📚Reads 📚Res