

Cost of

architecture

Cost of project

Cost of

changes

**Always code as if the guy maintaining your code would be a violent psychopath and he knows where you live.**

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

Cohesion(сцепление) - how strongly related and focused the various responsibilities of a module are(high)

Coupling(связанность) - the degree to which each program module relies on each one of the other modules(loose)

**SRP(разделяй и влавствую – римская империя)**





*We’ve got it!*

*Grand Swiss Army Knife*

1. Есть сломалась часть – что-то заменить простым способом – проблематично
2. У меня швейцарский нож, но другу нужен штопор, но в то же время нож я отдать не могу – я режу что-то. Нож я отдать не могу

Часто получаем один класс, который делает много чего, чего делать не должен.

***A class should have only one reason to change!***



Класс решает только одну проблему – резать. У него высокий cohesion.

Что дает? – меньше вероятность что-то сломать, чем если мы работаем с одним мега-классом. Класс, который сконцентрирован на решении одной проблемы легче поддерживать, читать, тестировать, он более надежный. **Не доходить до фанатизма!**

more readable, that is easier to understand

· less error prone

· more robust

· better testable

· better maintainable and extendable





Пример – с интегралами. Либо логин вью.

**OCP**

***Software entities should be open for extension, but closed for modification***

You should be able to add new features and extend a class without changing its internal behaviour. You can always add new behaviour to a class in the future. At the same time you should not have to recompile your application just to make room for new things. The main goal of the principle is to avoid breaking changes in an existing class as it can introduce bugs and errors in other parts of your application.

The key to success is identifying the areas in your domain that are likely to change and programming to abstractions. Separate out behaviour into abstractions: interfaces and abstract classes. There’s then no limit to the variety of implementations that the dependent class can accept.

**Three Approaches to Achieve OCP**

* Parameters
  + Allow client to control behavior specifics via parameter
  + Combined with lambda can be powerful approach
* Inheritance/Template Method
* Composition/Strategy Pattern
  + Client code depends on abstraction
  + Providers “plug in” model
  + Implementation utilize inheritance; Clients utilize composition

**When do we apply OCP**

* Experience will tell you
* Otherwise – “Fool me one, shame on you; fool me twice shame on me”
  + Don’t apply OCP first
  + If module changed once, accept it
  + If it changes a second time, refactor to achieve OCP
* Remember TANSTAAFL
  + There Ain’t No Such Thing As A Free lunch
  + OCP adds complexity to design
  + No design can be closed against all changes

Пример – пойнт оф сейл(дискаунт)

**Liskov Substitution Principle(подстановки)**



***Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it***





pillow



mattress

cushion

***Child classes must not remove:***

* They must not remove any base class behaviour
* They must not violate base [class invariants](http://en.wikipedia.org/wiki/Class_invariants), i.e. the rules and constraints of a class, in order to preserve its integrity

Если в базовом классе определена скорость света, то потомок не должен пытаться ее переопределить.

The first point means the following: if a base class defines two abstract methods then a derived class must give meaningful implementations of both. If a derived class implements a method with ‘throw new NotImplementedException’ then it means that the derived class is not fully substitutable for its base class. It is a sign that the base class is ‘NOT-REALLY-A’ base class type. In that case you’ll probably need to reconsider your class hierarchy.

Пример – репозиторий сейв(Entity). Изменение требований – для виджета еще что-то. Как вариант – репозиторий под каждый entity.

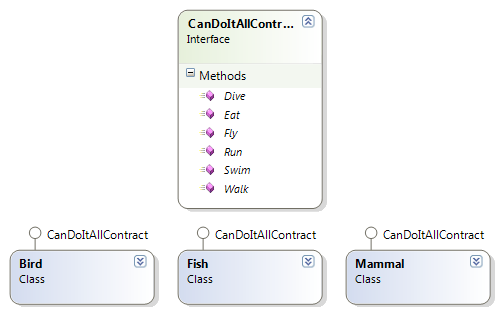
ExistingFIle, NewFile, REadOnlyFIle – Load, Save. Классы наследники переопределяют, меняют поведение базового. Если заменить на интерфейс, мы хотя бы знаем, что может менять имплементация.

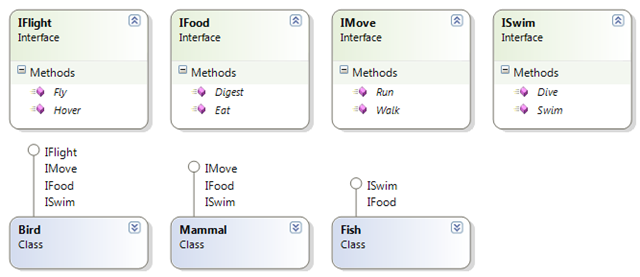
Главное – выбирать правильные абстракции, чтобы клиент класса не задумывался о конкретной абстракции, а просто пользовался ей.

* Tell, Don’t Ask
  + Don’t interrogate objects to their internals
  + Tell the object what you want it to do
* Consider refactoring to a new base class
  + Given two classes that share a lot of behavior but are not substitutable…
  + Create new base class that both can derive from
  + Ensure substitutability is retained between each class and new base

**Interface Segregation Principle**

***Clients should not be forced to depend on methods that they do not use***





Изменение в одном большом интерфейсе влечет к тому, что все реализацию меняются - они вынуждены либо добавить, либо изменинть либо еще что-то. Сложно определять, чем пользуются, а чем нет – какие методы определены а какие нет.

Пример. IPointOfSaveWidget – Sum, Products. I want to show new Widget SumToPayWidget – нельзя наследовать, потому что продукты нам не нужны – вынуждаем реализовывывать.

Разделить интерфейс на SumToShow, ProductShow, подумать о другой иерархии. Очень близко с SRP. Мы вынуждаем клиентов нарушать SRP – разделяем, чтобы они отвечали за свой кусочек логики.

**When do we fix ISP**

* Once there is a pain
  + If there is no pain, there’s no problem to address
* If you find yourself depending on ‘fat’ interface you own
  + Create a smaller interface with just what you need
  + Have a ‘fat’ interface implement your new interface
  + Reference the new interface in your code
* If you find ‘fat’ interfaces are problematic but you don’t own them
  + Create a smaller interface with just what you need
  + Implement this interface using an Adapter that implements the full intrface

**Dependency Inversion**

***High-level classes should not depend on low-level classes. Both should depend on abstractions.***



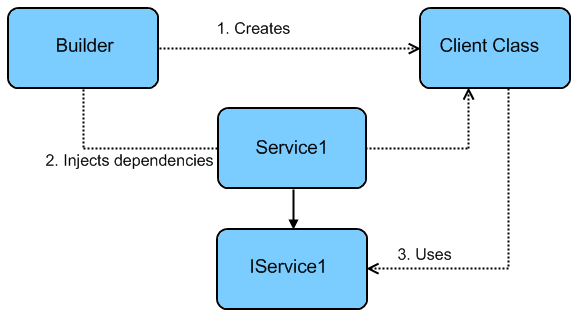
* Dependency Inversion Principle
  + Principle used in architecting software
* Inversion of Control
  + Specific pattern used to invert interfaces, flow and dependencies
* Dependency Injection
  + Implementation of IoC to invert dependencies
* Inversion of Control Container
  + Framework to do dependency injection

**Stable Dependencies**

* The class or module already exists.
* You expect that new versions won’t contain breaking changes.
* The types in question contain deterministic algorithms.
* You never expect to have to replace the class or module with another.

**VOLATILE DEPENDENCIES**

* The dependency doesn’t yet exist, but is still in development.
* The dependency isn’t installed on all machines in the development organization. The most common symptom is disabled TESTABILITY.
* The dependency contains nondeterministic behavior.



GRASP, DRY, KISS