There should be more than one reason for a class to change.

Cohesion: how strongly related and focused are the various responsibilities of a module.

Coupling: the degree to which each program module relies on each one of the other modules.

**What is responsibility?**

* A reason to change
* A difference in usage scenarios from the client perspective
* Multiple small interface (follow ISP) can help to achieve SRP

Following SRP leads to lower coupling and higher cohesion.

Many small classes with distinct responsibilities result in more flexible design

**Open close principle**

Open close principle states that software entities(classes , modules, functions, etc) should be open for extension, but closed for modification.

**Open to extension**

New behavior can be added in the future

**Closed to modification**

Changes to source or binary code are not required.

Change behavior without changing code?

No limit to variety of implementations of each abstraction.

**The problem**

* Adding new rules required changes to the application every time
* Each change can introduce bugs and requires re-testing, etc.
* We want to avoid introducing changes that cascade through many modules in our application.
* Writing new classes is less likely to introduce problems.

**Three approaches to achieve OCP**

* Parameters (procedural programming)
  + Allow client to control behavior specific via parameter
  + Combined with delegates/lambda, can be very powerful approach.
* Inheritance/template method pattern
  + Child types override behavior of a base class (or interface)
* Composition / strategy pattern
  + Client code depends on abstraction
  + Provide a plugin model
  + Implementations utilize inheritance ; client utilize composition.

**Liskov substitution principle**

States that subtypes must be substitutable for their base types.

Child classes must not:

* Remove base class behavior
* Violate base class invariants

Fixing substitutability problem by adding if-then or switch statement quickly became a maintenance nightmare.

**ISP Tips**

* Keep interfaces small, cohesive, and focused.
* Whenever possible, let the client define the interface.
* Whenever possible, package the interface with the client.

1. Don’t force client code to depend on things it doesn’t need
2. Keep interfaces lean and focused.
3. Refactor large interfaces so they inherit smaller interface

**Dependency inversion**

High level modules should not depend on low level modules. Both should depend on abstraction.

Abstraction should not depend on details. Details should depend on abstraction.

What are dependencies?

* Framework
* Third party libraries
* Database
* File system
* Email
* Webs service
* System resource (clock)
* Configurations
* New keyword
* Static methods
* Thread. Sleep
* Random
* Static methods are used for convenience or as façade layers.
* Call instantiation / call stack logic is scattered through all modules.
  + Violation of single responsibility principle.
* Class constructors should require any dependencies the class needed.
* Classes whose constructors make this clear have explicit dependencies
* Classes that do not have implicit, hidden dependencies.

**Dependency injection**

* Dependency injection is a technique that is used to allow calling code to inject the dependencies a class needed when it is instantiated.
* Hollywood principle
* Tree primary techniques
  + Constructor injection
  + Property injection
  + Parameter injection
* Other methods exist as well

**Constructor injection (strategy pattern)**

* Dependencies are passes in via constructor
* Pros
  + Classes self-document what they need to perform their work
  + Works well with or without container
  + Classes are always in a valid state once constructed.
* Cons
  + Constructors can have many parameters/dependencies (design smell)
  + Some features (e.g. serialization) may require a default constructor
  + Some methods in the class may not require things other methods require (design smell)

**Property injection**

* Dependencies are passes in via a property
  + Also knows as “Setter injection”
* Pros
  + Dependency can be changed at any time during object life time
  + Very flexible
* Cons
  + Objects may be in an invalid state between construction and setting of dependencies via setter
  + Less intuitive

**Property injection**

* Dependencies are passed in via a method parameter
* Pros
  + Most granular
  + Very flexible
  + Requires no changes to rest of class
* Cons
  + Break method signature
  + Can result in many parameter (design smell)
* Consider if only one method has the dependency otherwise prefer constructor injection

**Refactoring:**

* Extract dependencies into interfaces
* Inject implementations of interfaces into order
* Reduce orders responsibilities (SRP principle)

Inversion of control container

**Layer/tired application design**

* Separate logical and sometimes physical layers
  + For instance
  + User interface
  + Business logic layer
  + Data access layer
* Support encapsulation and abstraction
  + Work at the abstraction level appropriate
  + Each level only knows about one level deep(ideally)
* Provide units of reuse
  + Lowest levels generally almost reusable

Dependency

Business logic layer

User interface

Common

Data Access Layer

Services

Database

**Inverted architecture**

WCF (Web Svcs)

IO

Tests

UI

Data access

Business logic (service)

Object model (core , domain objects)

Services

Database

**Don’t repeat yourself**

**Repetition is the root of all software evil.**

**Once and only once.**

**Duplication is evil.**

Analysis:

* Magic string/values
* Duplicate logic in multiple locations
* Repeated if-then logic
* Conditional instead of polymorphism
* Repeated execution pattern
* Lots of duplicate, probably copy-pasted, code
* Only manual testes
* Static methods everywhere

Use of flags over objects is anti-pattern

Static methods are bad because:

* Tightly coupled
* Difficult to test
* Difficult to change behavior
* Cannot use object-oriented design techniques
  + Inheritance
  + polymorphism