Telerik Academy

SOLID and Other Principles

SOLID, DRY, YAGNI, KISS

High-Quality Code

Telerik Software Academy

http://academy.telerik.com

Telerik Academy

Table of Contents

- SOLID Principles
  - SRP – Single Responsibility Principle
  - OCP – Open/Closed Principle
  - LSP – Liskov Substitution Principle
  - ISP – Interface Segregation Principle
  - DIP – Dependency Inversion Principle
- DRY – Don't Repeat Yourself
- YAGNI – You Aren't Gonna Need It
- KISS – Keep It Simple, Stupid

Telerik Academy

Single Responsibility

SINGLE RESPONSIBILITY PRINCIPLE

Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.

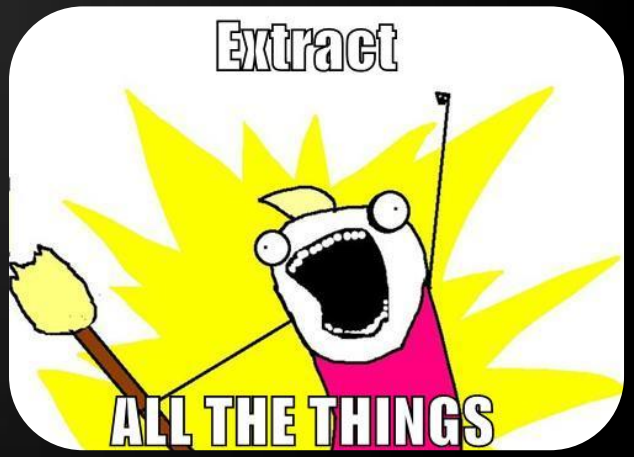
SRP

"The Single Responsibility Principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class."

Wikipedia

"There should never be more than one reason for a class to change."

Robert C. "Uncle Bob" Martin



SRP

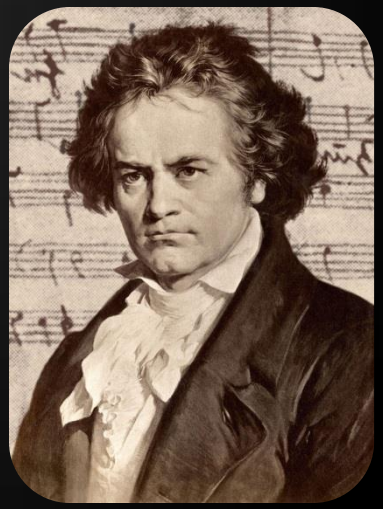
- Cohesion
  - Relation of responsibilities
  - Focused on one task
- Coupling
  - Dependency on other modules
  - Relationship between modules
- Ideal - low coupling / strong cohesion

SRP

- Responsibilities
  - "A reason to change"
  - Mapped to project requirements
  - More requirements – more possible changes
  - More responsibilities – more changes in code
  - Multiple responsibilities in one class – coupling
  - More coupling – more errors on change

SRP

- Classic violations
  - Objects that can print/draw themselves
  - Objects that can save/restore themselves
- Classic solution
  - Separate printer
  - Separate saver (or memento)



SRP

- Solution
  - Multiple small interfaces (ISP)
  - Many small classes
  - Distinct responsibilities
- Result
  - Flexible design
  - Lower coupling
  - Higher cohesion

Telerik Academy

Single Responsibility

Live Demo

Telerik Academy

Open/Closed Principle

OCP

"The Open / Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

Wikipedia

- Open to Extension
  - New behavior can be added in the future
- Closed to Modification
  - Changes to source or binary code are not required

OCP

- Change behavior without changing code?!
  - Rely on abstractions, not implementations
  - Do not limit the variety of implementations
- In .NET
  - Interfaces
  - Abstract Classes
- In procedural code
  - Use parameters

OCP

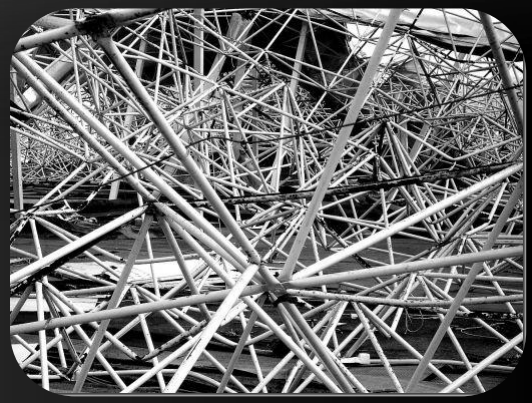
- Classic violations
  - Each change requires re-testing (possible bugs)
  - Cascading changes through modules
  - Logic depends on conditional statements
- Classic solution
  - New classes (nothing depends on them yet)
  - New classes (no legacy coupling)

OCP

- Three approaches to achieve OCP
  - Parameters
    - Pass delegates / callbacks
  - Inheritance / Template Method pattern
    - Child types override behavior of a base class
  - Composition / Strategy pattern
    - Client code depends on abstraction
    - "Plug in" model

OCP

- When to apply OCP?
  - Experience tell you
  - "Fool me once, shame on you"
    - Don't apply OCP at first
    - If module changes once, accept it
    - If it changes a second time, refactor for OCP



- OCP add complexity to design (*TANSTAAFL)*
- No design can be closed against all changes

Telerik Academy

Open / Closed

Live Demo

Telerik Academy

Liskov Substitution

LSP

> *"The Liskov Substitution Principle states that Subtypes must be substitutable for their base types."*
>
> Agile Principles, Patterns, and Practices in C#

- Substitutability – child classes must not
  - Remove base class behavior
  - Violate base class invariants

LSP

- Normal OOP inheritance
  - IS-A relationship
- Liskov Substitution inheritance
  - IS-SUBSTITUTABLE-FOR

LSP

- The problem
  - Polymorphism break
  - Client code expectations
  - "Fixing" by adding if-then – nightmare (OCP)
- Classic violations
  - Type checking for different methods
  - Not implemented overridden methods
  - Virtual methods in constructor

LSP

- Solutions
  - "Tell, Don't Ask"
    - Don’t ask for types
    - Tell the object what to do
  - Refactoring to base class
    - Common functionality
    - Introduce third class

Telerik Academy

Liskov Substitution

Live Demo

Telerik Academy

Interface Segregation

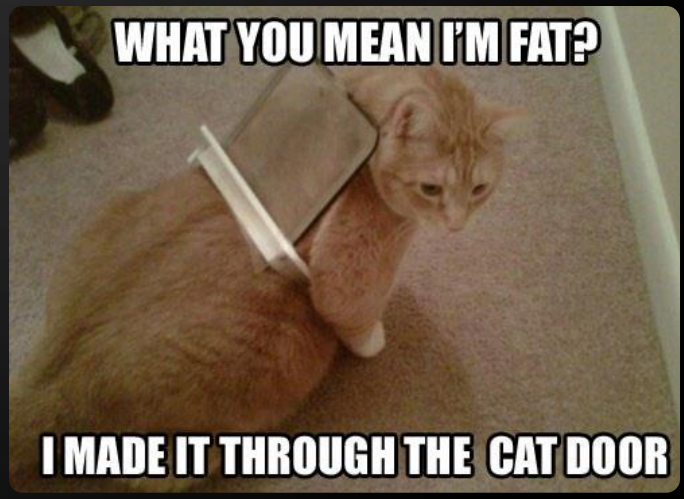
ISP

"The Interface Segregation Principle states that Clients should not be forced to depend on methods they do not use."
Agile Principles, Patterns, and Practices in C#

- Prefer small, cohesive interfaces
- Divide "fat" interfaces into smaller ones

ISP

- Interface is:
  - The interface type
  - All public members of a class
- Having "fat" interfaces leads to:
  - Classes having methods they do not need
  - Increasing coupling
  - Reduced flexibility
  - Reduced maintainability



ISP

- Classic violations
  - Unimplemented methods (also in LSP)
  - Use of only small portion of a class
- When to fix?
  - Once there is pain! Do not fix, if is not broken!
  - If the "fat" interface is yours, separate it to smaller ones
  - If the "fat" interface is not yours, use "Adapter" pattern

ISP

- Solutions
  - Small interfaces
  - Cohesive interfaces
  - Focused interfaces
  - Let the client define interfaces
  - Package interfaces with their implementation

Telerik Academy

Interface Segregation

Live Demo

Telerik Academy

Dependency Inversion

Dependency Inversion Principle

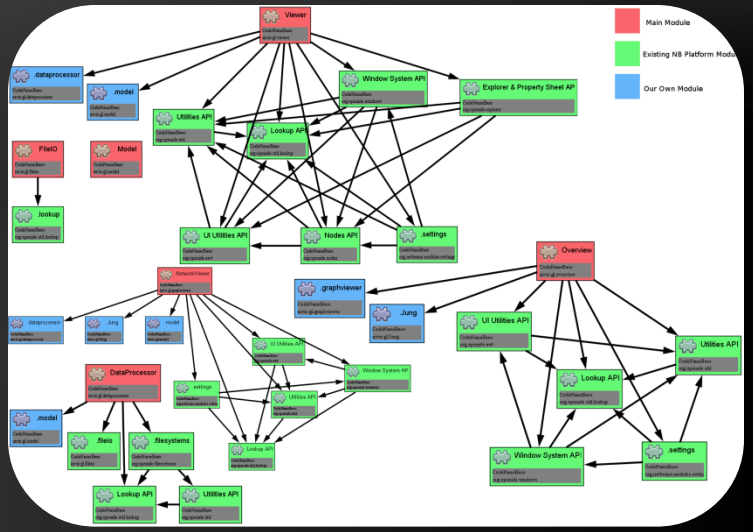
Would you solder a lamp directly
to the electrical wiring in a wall?

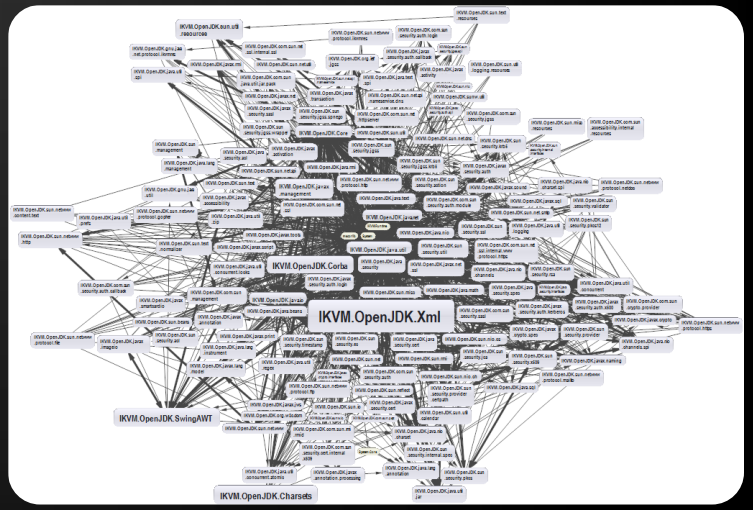
DIP

"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."

Agile Principles, Patterns, and Practices in C#



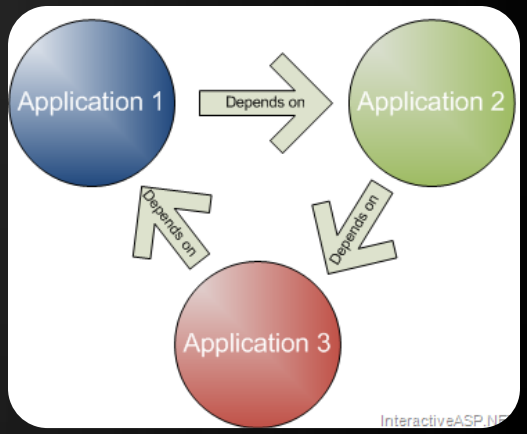


DIP

- Dependency is:
  - Framework
  - Third Party Libraries
  - Database
  - File System
  - Email
  - Web Services
  - System Resources (Clock)
  - Configuration
  - The new Keyword
  - Static methods
  - Thread.Sleep
  - Random

DIP

- Traditional Programming
  - High level modules use lower lever modules
  - UI depends on Business Layer
  - Business layer depends on
    - Infrastructure
    - Database
    - Utilities
  - Static methods (Façade for example)
  - Classes instantiated everywhere



DIP

- How it should be
  - Classes should declare what they need
  - Constructors should require dependencies
  - Hidden dependencies should be shown
  - Dependencies should be abstractions
- How to do it
  - Dependency Injection
  - The Hollywood principle
    "Don't call us, we'll call you!"

DIP

- Constructor injection
  - Dependencies – through constructors
  - Pros
    - Classes self document requirements
    - Works well without container
    - Always valid state
  - Cons
    - Many parameters
    - Some methods may not need everything

DIP

- Property injection
  - Dependencies – through setters
  - Pros
    - Can be changed anytime
    - Very flexible
  - Cons
    - Possible invalid state
    - Less intuitive

DIP

- Parameter injection
  - Dependencies – through method parameter
  - Pros
    - No change in rest of the class
    - Very flexible
  - Cons
    - Many parameters
    - Breaks method signature

DIP

- Classic violations
  - Using of the new keyword
  - Using static methods/properties
- How to fix?
  - Default constructor
  - Main method/starting point
  - Inversion of Control container

DIP

- IoC containers
  - Responsible for object instantiation
  - Initiated at application start-up
  - Interfaces are registered into the container
  - Dependencies on interfaces are resolved
  - Examples – StructureMap, Ninject and more

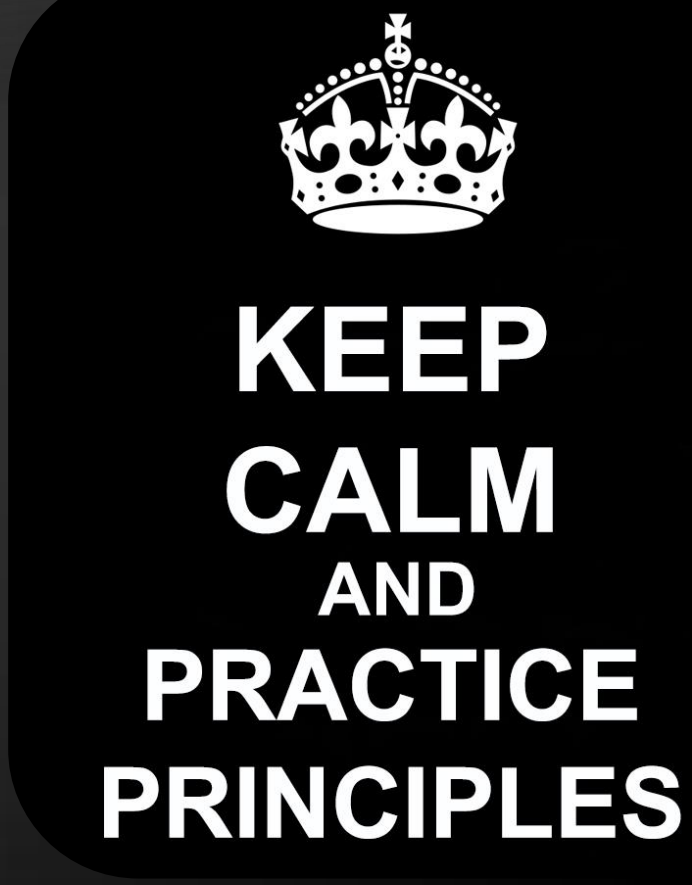
Telerik Academy

Dependency Inversion

Live Demo

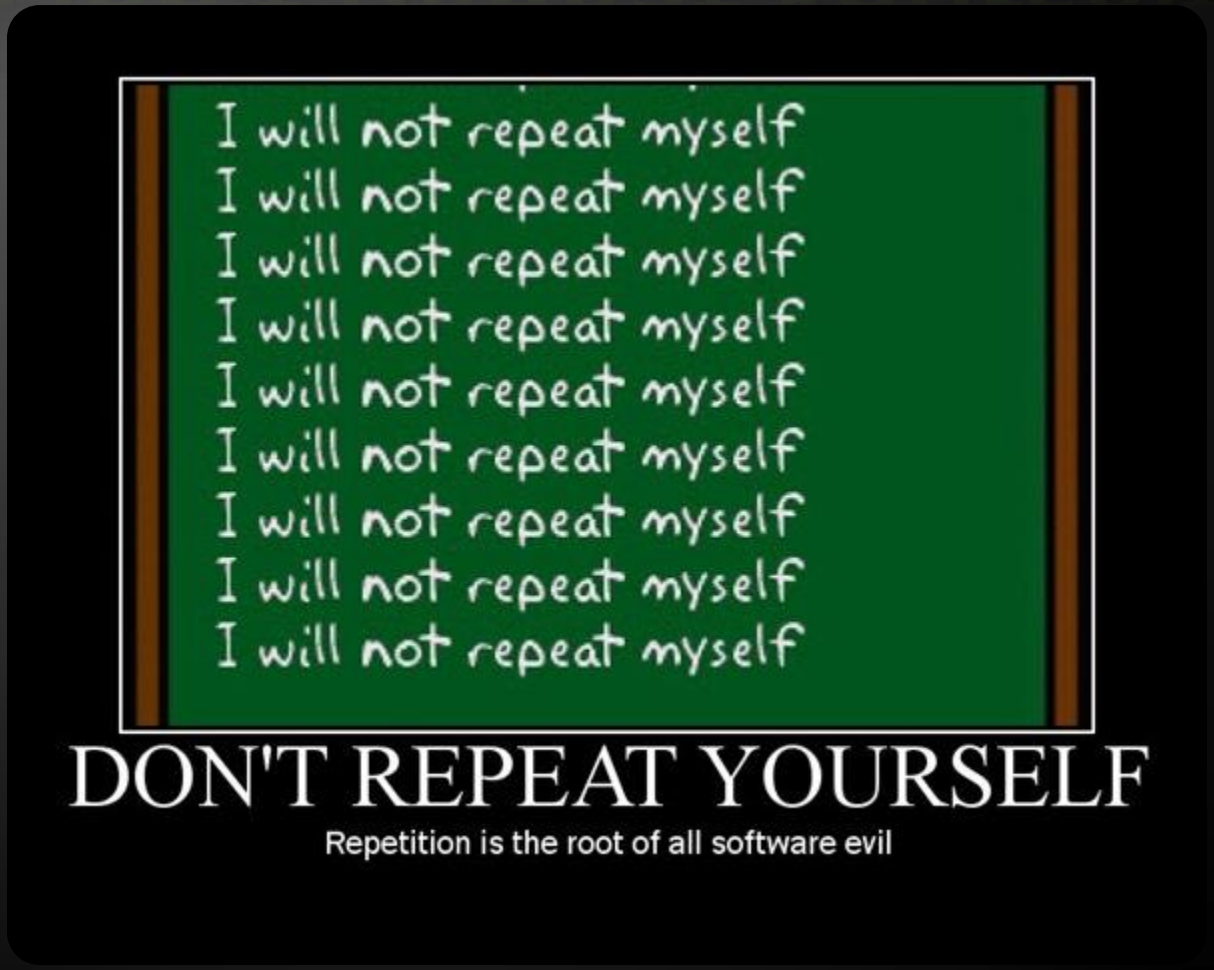
Telerik Academy

Other Principles



Telerik Academy

Don't Repeat Yourself



DRY

> *"Every piece of knowledge must have a single, unambiguous representation in the system."*
>
> The Pragmatic Programmer

> *"Repetition in logic calls for abstraction. Repetition in process calls for automation."*
>
> 97 Things Every Programmer Should Know

- Variations include:
  - Once and Only Once
  - Duplication Is Evil (DIE)

DRY

- Classic violations
  - Magic Strings/Values
  - Duplicate logic in multiple locations
  - Repeated if-then logic
  - Conditionals instead of polymorphism
  - Repeated Execution Patterns
  - Lots of duplicate, probably copy-pasted, code
  - Only manual tests
  - Static methods everywhere

Telerik Academy

Don't Repeat Yourself

Live Demo

Telerik Academy

You Ain't Gonna Need It

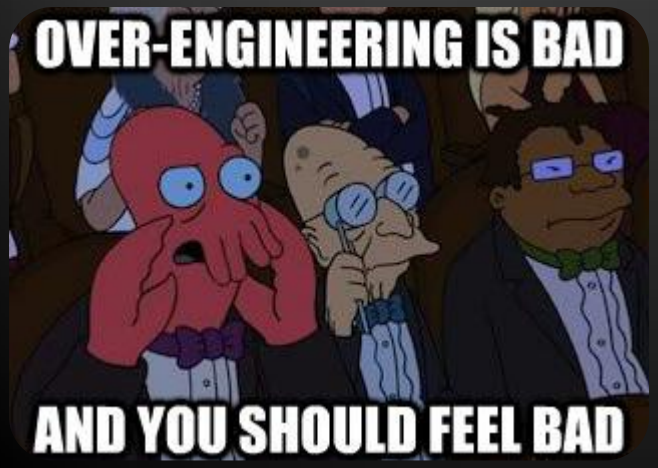
YAGNI

"A programmer should not add functionality until deemed necessary."

Wikipedia

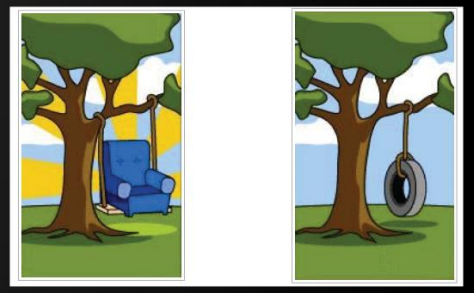
"Always implement things when you actually need them, never when you just foresee that you need them."

Ron Jeffries, XP co-founder



YAGNI

- Disadvantages
  - Time for adding, testing, improving
  - Debugging, documented, supported
  - Difficult for requirements
  - Larger and complicate software
  - May lead to adding even more features
  - May be not know to clients



Telerik Academy

You Ain't Gonna Need It

Live Demo

Telerik Academy

Keep It Simple, Stupid

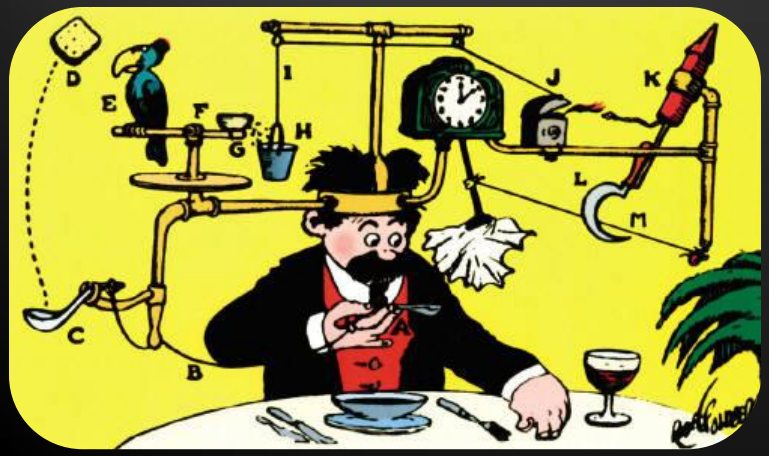
KISS

"Most systems work best if they are kept simple."

U.S. Navy

"Simplicity should be a key goal in design and unnecessary complexity should be avoided."

Wikipedia



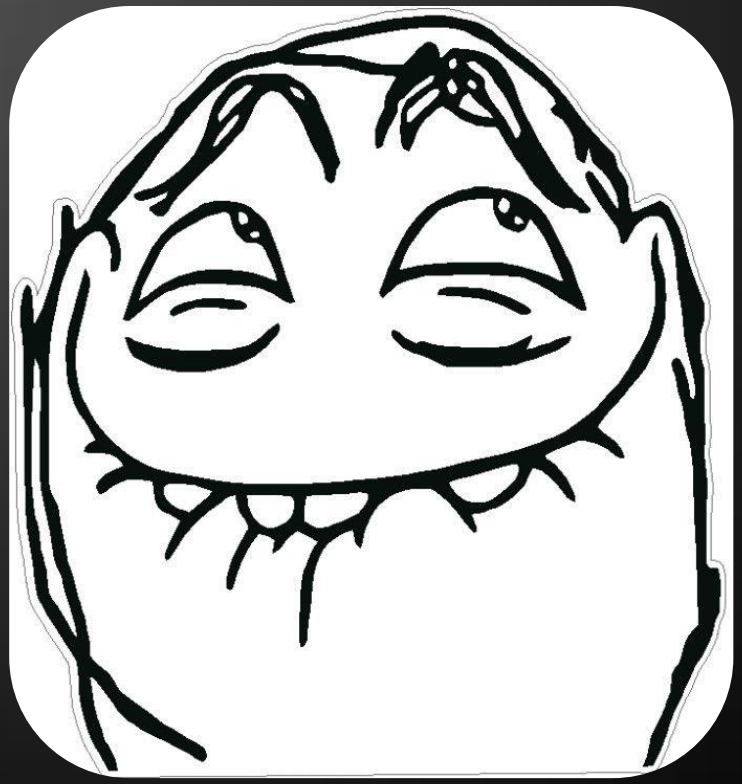
Telerik Academy

Keep It Simple Stupid

Live Demo

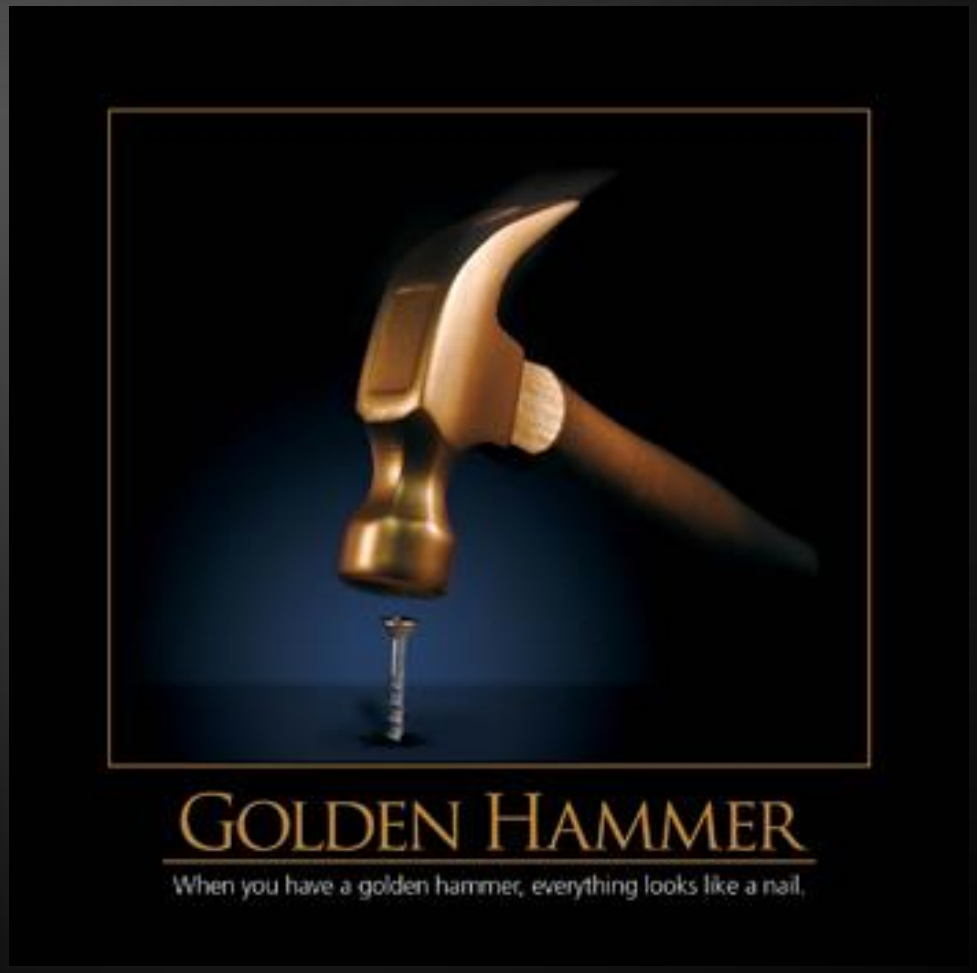
Telerik Academy

Even More Principles



Telerik Academy

Golden Hammer



Telerik Academy

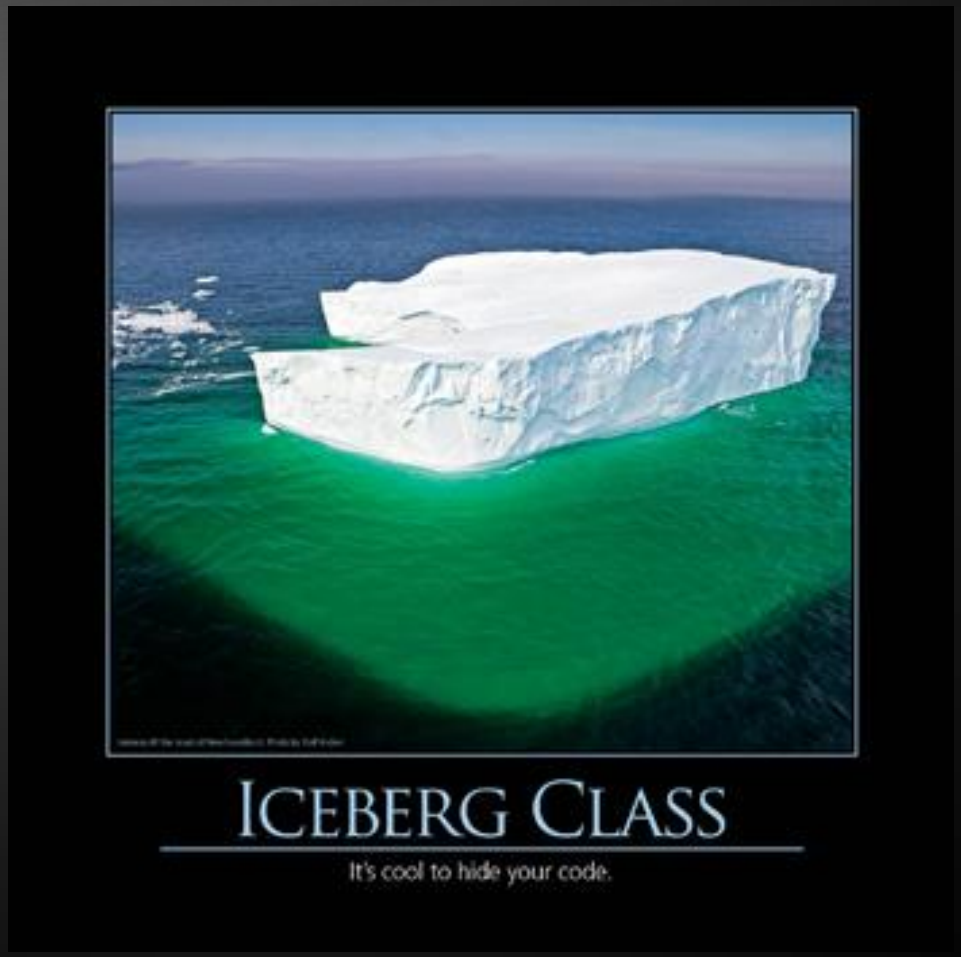
Feature Creep

Telerik Academy

Duck Tape Coder

Telerik Academy

Iceberg Class

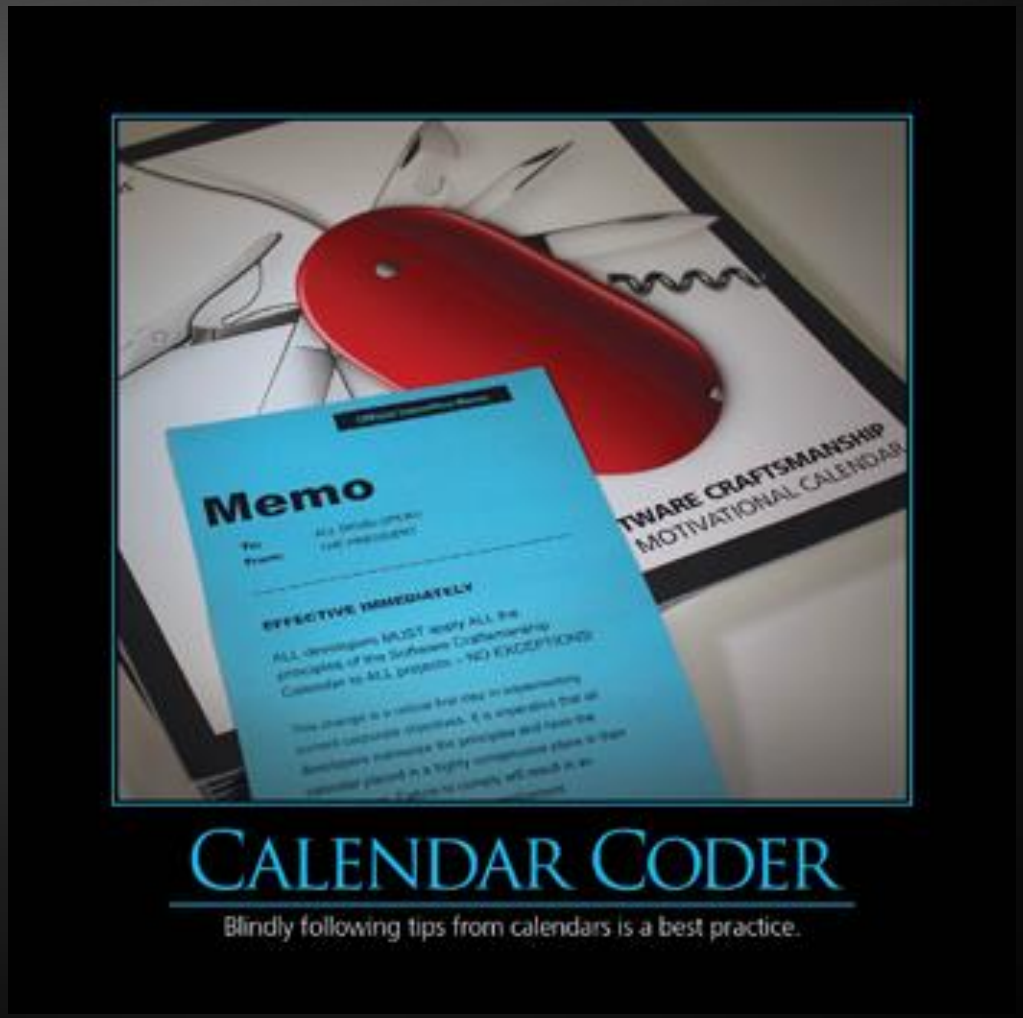


Telerik Academy

Spaghetti Code

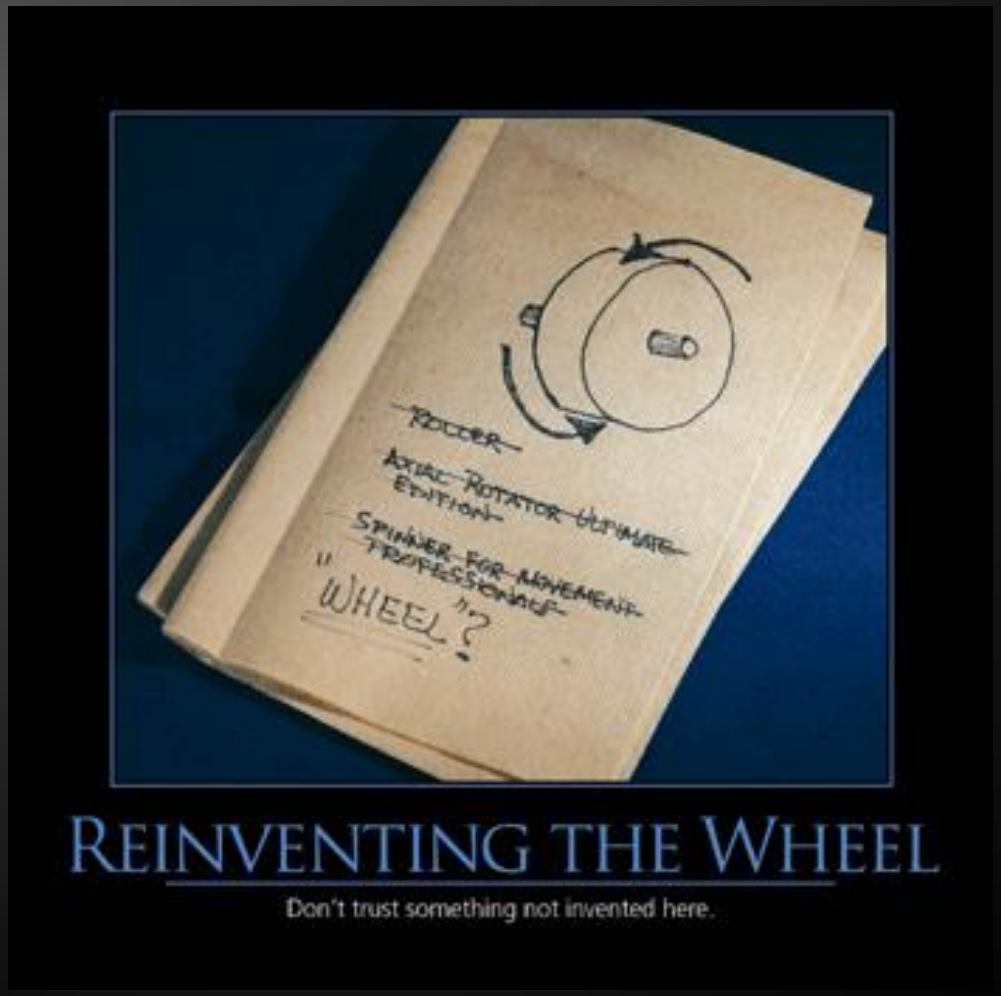
Telerik Academy

Calendar Coder



Telerik Academy

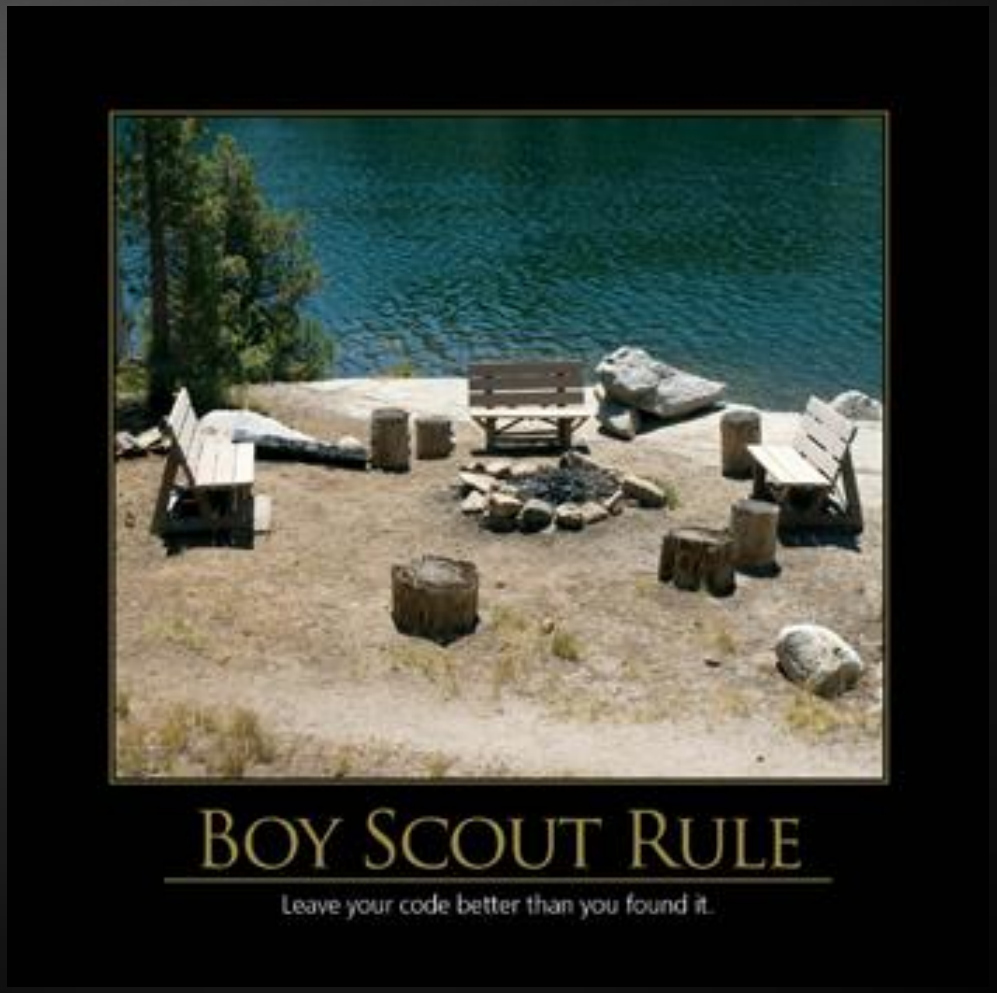
Reinventing The Wheel



Telerik Academy

Copy Paste Programming

Boy Scout Rule



Telerik Academy

SOLID And Other Principles

Questions?

http://academy.telerik.com

Free Trainings @ Telerik Academy

- C# Programming @ Telerik Academy
  - csharpfundamentals.telerik.com

- Telerik Software Academy
  - academy.telerik.com

- Telerik Academy @ Facebook
  - facebook.com/TelerikAcademy

- Telerik Software Academy Forums
  - forums.academy.telerik.com