Fundamental Software Design Principles for Quality Coding

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

About Me

I have more than 24 years of experience in software development, being involved in various projects covering desktop, client/server, web, cloud, and mobile applications using mainly Microsoft tools & technologies, but also Apple and open-source.

I'm a Certified Scrum Master and a certified APMG Agile Project Manager. Since 2003, I'm holding a Ph.D. degree in Industrial Engineering.

You can find more about me on LinkedIn: https://www.linkedin.com/in/eduardghergu/

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Fundamental Software Design Principles

1. The importance of Software Design Principles
2. Common Principles: what, how, and code samples
3. Summary
4. Q & A

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Why do we need SDP ?!

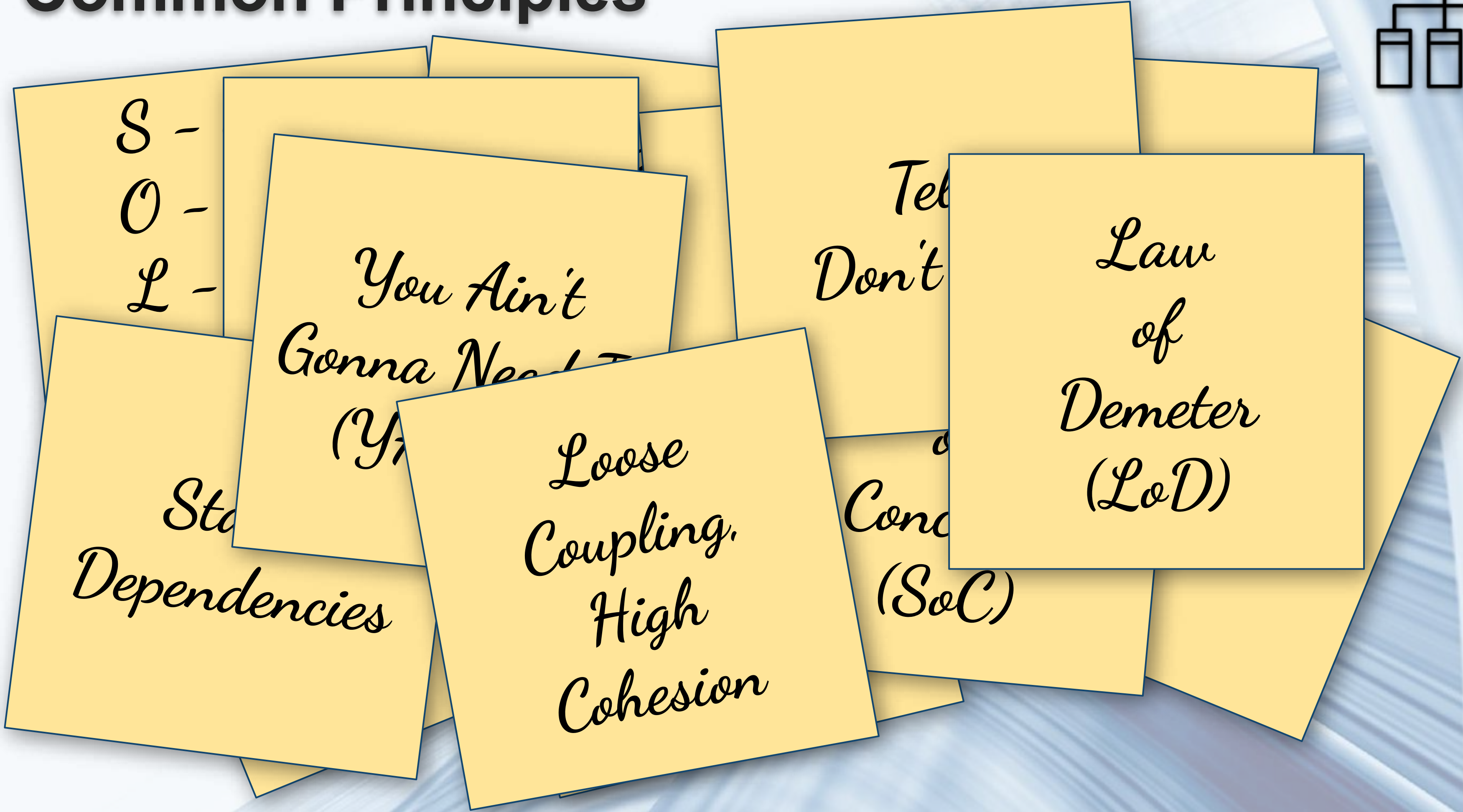
The Software Design Principles are a set of guidelines that help developers in creating good system designs.

Only 20 to 40% of the development time will be spent writing code; the rest will be spent reading code and maintaining the system.

As a result, creating a good system design is critical. A good system should have a good code base that is easy to read, understand, maintain (add/modify features, fix bugs), and extend in the future. This will cut down on development time and resources while also increasing our happiness.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Common Principles



Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

About sample source code

Github public repository with code samples Before and After applying the principles:

https://github.com/AbstractSoft/design_principles

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

SOLID Principles [3]

- S - Single Responsibility Principle (SRP)
- O - Open-Close Principle (OCP)
- L - Liskov Substitution Principle (LSP)
- I - Interface Segregation Principle (ISP)
- D - Dependency Inversion Principle (DIP)

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Single Responsibility Principle (SRP) - 1

Every module, class or function in a computer program should have responsibility over a single part of that program's functionality

Also, they should encapsulate that part and their services should be narrowly aligned with that responsibility[1]

SRP is closely related to the concepts of Coupling (low) and Cohesion (high)

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Single Responsibility Principle (SRP) - 2

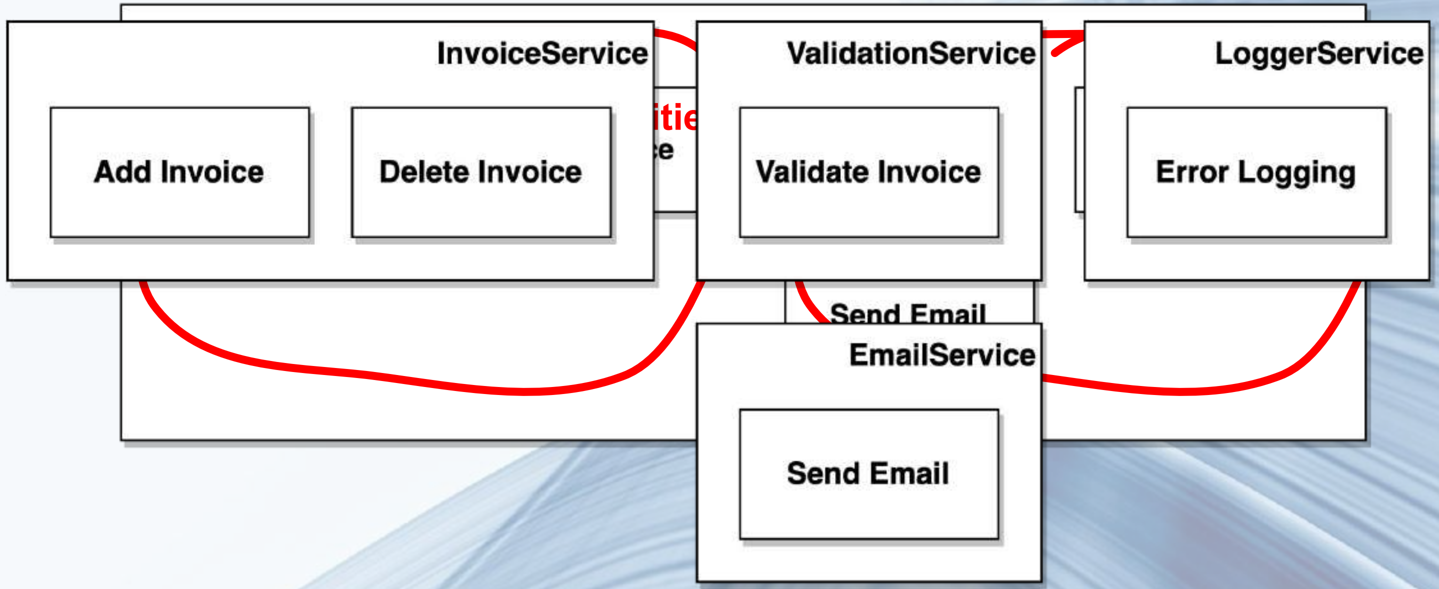
SRP does not necessarily mean that your class should only have one method or property, but rather that the functionality should be related to a single responsibility.

With the help of SRP the classes become smaller and cleaner and thus easier to maintain.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Single Responsibility Principle (SRP) - 3

Invoice Service Sample



Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Open-Close Principle (OCP) - 1

OCP states that “software entities such as modules, classes, functions, etc. should be open for extension, but closed for modification“.

In simple words, one module/class should be developed in such a way that it should allow its behavior to be extended without altering its source code.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Open-Close Principle (OCP) - 2

How to apply OCP

- add the new functionalities by creating new derived classes which should be inherited from the original base class.
- allow the client to access the original class with an abstract interface.

So, instead of changing the existing functionality, create new derived classes and leave the original class implementation as it is.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Open-Close Principle (OCP) - 3

Problems of Not following OCP

- If you allow a class or function to add new logic, you must test the entire functionality of the application, including both new and existing functionality.

- It is also required to inform the QA team about the future changes so that they can prepare for regression testing as well as new feature testing.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Open-Close Principle (OCP) - 4

Problems of Not following OCP

- If you are not following OCP, then it is possible to break the SRP as long as the class or module is going to have multiple responsibilities.

- If you are implementing all the functionalities in a single class, then the maintenance of the class becomes very difficult (testing included).

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Liskov Substitution Principle (LSP)

The LSP is a Substitutability principle in OOP. It states that, if S is a subtype of T, then objects of type T should be replaced with the objects of type S.

So, if we can successfully replace the object/instance of a parent class with an object/instance of the child class, without affecting the behavior of the base class instance, then LSP is followed.

For example, a father is a teacher whereas his son is a doctor. In this case, the son can’t simply replace his father even though both belong to the same family.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Interface Segregation Principle (ISP)

The ISP states that “Clients should not be forced to implement any methods they don’t use. Rather than one fat interface, numerous little interfaces are preferred based on groups of methods with each interface serving one submodule“.

The above definition can be split into two parts:

1. No class should be forced to implement any interface method(s) that it does not use.

2. Rather than creating large interfaces, create multiple smaller interfaces with the goal of allowing clients to focus on the methods that are relevant to them.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Dependency Inversion Principle (DIP) - 1

DIP states that high-level modules/classes should not depend on low-level modules/classes.

Both should depend upon abstractions.

Secondly, abstractions should not depend on details. Details should depend upon abstractions.

Always try to keep the High-level module and Low-level module as loosely coupled as possible.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Dependency Inversion Principle (DIP) - 2

When a class knows about the design and implementation of another class, it raises the risk that if we do any changes to one class will break the other class.

So we must keep these high-level and low-level modules/classes loosely coupled as much as possible.

To do that, we need to make both of them dependent on abstractions instead of knowing each other.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Boy Scout Rule - Clean Code - 1 [4]

It’s not enough to write the code well. The code has to be kept clean over time. So we must take an active role in preventing this degradation.

The Boy Scouts of America have a simple rule that we can apply to our profession: Leave the campground cleaner than you found it.

The cleanup: Apply Clean Code Prescriptions

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Boy Scout Rule - Clean Code - 2 [4]

Clean Code Prescriptions

- **Change names for the better**
- **Break up one function that’s a too large**
- **Eliminate one small bit of duplication**
- **Clean up one composite if statement**
- **Remove any comments (incl. code)**

… and others that are covered next

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Don't Repeat Yourself (DRY) - 1 [5]

The idea behind DRY design principle is an easy one: a piece of code, or even logic, should only appear once in an deployment unit (app, lib, etc.)

The code that is common to at least two different parts of your system should be refactored out into a single location so that both parts call upon in

Our motto should be the following:

Repetition is the root of all software evil

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Don't Repeat Yourself (DRY) - 2 [5]

Repetition does not only refer to writing the same piece of code or logic twice in two different places. It also refers to repetition in your processes – testing, debugging, deployment etc.

Repetition in code or logic is often solved by abstractions or some common service classes

Repetition in your process is tackled by automation.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Don't Repeat Yourself (DRY) - 3 [5]

Common violations:

- “Magic” values (strings, numbers) -> replace with constants with meaningful names
- Repeated code (the same lines of code in different places) -> refactor to methods/classes, maybe apply Strategy Design Pattern
- Repeated logic (some parts of the code are doing almost the same thing, but are not identical) -> refactor to methods/classes, maybe apply Strategy Design Pattern

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Encapsulation - 1 [2]

Encapsulation refers to the idea that objects should manage their own behavior and state, so that their collaborators need not concern themselves with the object's inner workings.

Object-oriented programming languages provide built-in support to control the visibility of class-level structures, and developers should use these constructs to differentiate between objects' public and non-public interfaces.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Encapsulation - 2 [2]

Failure to properly apply the principle of encapsulation to object-oriented designs leads to many related code smells and design problems, such as violating Don't Repeat Yourself (DRY), Tell, Don't Ask, and Flags Over Objects anti-pattern, to name a few.

Objects should ideally be kept in valid states, controlling how their state is modified to avoid being placed into states that do not make sense.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Encapsulation - 3 [2]

Sample:

Initial:

```
public class Product
{
  public double Volume;
}
```

Step 1:

```
public class Product
{
   public double Volume { get; set; }
}
```

Step 2:

```
public class Product
{
  private double volume;

  public double Volume {
    get { return volume; }
    set
    {
      if (value < 0)
      {
        throw new ArgumentOutOfRangeException("Volume must be 
non-negative.");
      }
      volume = value;
    }
  }
}
```

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Encapsulation - 4 [2]

Sample:

Step 3:

```
public class Volume
{
  private Volume(double amount, string unitOfMeasure)
  {
    if(amount < 0) throw new ArgumentOutOfRangeException("Volume amount must be non-negative.");
    Amount = amount;
    UnitOfMeasure = unitOfMeasure;
  }
  public static Volume InLiters(double amount)
  {
    return new Volume(amount, "Liters");
  }
  // include other static factory methods here for other units
  public double Amount { get; private set; }
  public string UnitOfMeasure { get; private set; }
  // perhaps include methods here to convert from one unit to another
}
```

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Principle Of Least Astonishment (PoLA) [9]

It's all about predictable behavior - set expectations and then deliver on them.

PoLA is applicable to a wide range of design activities - and not just in computing (though that is often where the most astonishing things happen).

It is potentially astonishing for someone to have a class that tries to do everything - or needing two classes to do a single thing. It is also likewise astonishing to find two methods that do apparently the same thing.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Don't Call Us, We'll Call You (Hollywood) - 1

It's closely related to the Dependency Inversion Principle, and illustrates a different way of writing software from the more traditional form of programming in which one's own code dictates the flow of control.

When following this principle, code is written to respond to external events, such as from an existing framework.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Don't Call Us, We'll Call You (Hollywood) - 2

Plug-in and extension models for applications frequently must take advantage of the Hollywood Principle in their design, exposing events and similar "hooks" for plug-in modules to use, allowing the overall program to call into the plug-in whenever needed, while avoiding the having the plug-in code take over control of the execution of the entire application.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Keep It Simple (KISS) - 1 [2]

When building software, an incremental approach that keeps things as simple as possible for as long as possible tends to yield working software with fewer defects, faster.

One way to minimize the amount of bugs in an application is to maximize the number of lines of code that aren't written, and avoiding needless complexity is a sure way to help achieve this goal.

It's also important to remember, when debating whether some complexity might be worthwhile, that many times You Aren't Gonna Need It. If we write our software such

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Keep It Simple (KISS) - 2 [2]

It's also important to remember, when debating whether some complexity might be worthwhile, that many times You Aren't Gonna Need It (YAGNI).

If we write our software such that it is flexible, we can add new functionality later when it's needed. To this end, simple software tends to be more malleable than complex software.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Persistence Ignorance (PI) - 1 [2]

This principle holds that classes modeling the business domain in a software application should not be impacted by how they might be persisted, and should not be tainted by concerns related to how the objects' state is saved and later retrieved.

Some common violations of PI include domain objects that must inherit from a particular base class, or which must expose certain properties.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Persistence Ignorance (PI) - 2 [2]

Sometimes, the persistence knowledge takes the form of attributes that must be applied to the class, or support for only certain types of collections or property visibility levels (see ORMs).

There are degrees of persistence ignorance, with the highest degree being described as Plain Old CLR Objects (POCOs) in .NET, and Plain Old Java Objects (POJOs) in the Java world.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Separation of Concerns (SoC) - 1 [2]

A key principle of software development and architecture is the notion of SoC. At a low level, this principle is closely related to SRP.

The general idea is that one should avoid co-locating different concerns within the design or code.

For instance, mixing the business objects of an application with the logic for displaying the information, authentication, and logging would certainly violate the SoC principle.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Separation of Concerns (SoC) - 2 [2]

The design would be more maintainable, less tightly coupled, and less likely to violate DRY principle if the logic for determining which items needed formatted were located in a single location (with other business logic), and were exposed to the user interface code responsible for formatting simply as a property.

At an architectural level, separation of concerns is a key component of building layered applications. In a traditional N-tier application structure, layers might include data access, business logic, and user interface.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Separation of Concerns (SoC) - 3 [2]

In addition to separating logic across programming layers, one can also separate concerns along application feature sets. Applications may be written to allow functionality to be added or removed in a modular fashion (plugins).

SoC tends to be a natural consequence of following DRY principle, since of necessity abstractions must be built to encapsulate concepts that would otherwise be repeated throughout the application. As long as these abstractions are logically grouped and organized, then SoC should be achieved.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Stable Dependencies [2]

It states that "The dependencies between software packages should be in the direction of the stability of the packages. That is, a given package should depend only on more stable packages."

Whenever a package changes, all packages that depend on it must be validated to ensure they work as expected after the change.

"Writing software that fully meets its specifications is like walking on water. For each, the former is easy if the latter is frozen and near impossible if fluid." - Anonymous

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Tell, Don't Ask (TDA) - 1 [2]

According to this principle, it is preferable to issue a command to an object in order for it to perform some operation or logic, rather than querying its state and then taking action as a result. It is related to both the Flags Over Objects and the Anemic Domain Model antipatterns.

TDA violations are easy to spot in code that queries or uses multiple properties of an object to perform an action. This is especially problematic when the same action is performed multiple times (violating DRY), but it can also indicate a design flaw even if it only occurs once in the current codebase.

Tell, Don't Ask (TDA) - 2 [2]

Sample:

```
// Violates TDA
public class CpuMonitor
{
    public int Value { get; set; }
}

public class Client
{
    public void AlertService(List<CpuMonitor> cpuMonitors)
    {
        foreach (var cpuMonitor in cpuMonitors)
        {
            if (cpuMonitor.Value > 90)
            {
                // alert
            }
        }
    }
}
```

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Tell, Don't Ask (TDA) - 3 [2]

Sample:

```
// Refactored
public class CpuMonitor
{
    private readonly int alertThreshold;

    public CpuMonitor(int alertThreshold)
    {
        this.alertThreshold = alertThreshold;
    }

    public int Value { get; set; }

    public bool ExceedsThreshold
    {
        get { return Value >= alertThreshold; }
    }
}
```

```
public class Client
{
    public void AlertService(List<CpuMonitor> cpuMonitors)
    {
        foreach (var cpuMonitor in cpuMonitors)
        {
            if (cpuMonitor.ExceedsThreshold)
            {
                // alert
            }
        }
    }
}
```

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Tell, Don't Ask (TDA) - 4 [2]

Sample:

```
// Refactored Further
public class CpuMonitor
{
    private readonly int alertThreshold;
    private readonly Action<CpuMonitor> alertAction;

    public CpuMonitor(int alertThreshold,
           Action<CpuMonitor> alertAction)
    {
        this.alertThreshold = alertThreshold;
        this.alertAction = alertAction;
    }

    public int Value { get; set; }

    public bool ExceedsThreshold
    {
           get { return Value >= alertThreshold; }
    }

    public void Sample()
    {
        if (ExceedsThreshold)
        {
            alertAction(this);
        }
    }
}
```

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Tell, Don't Ask (TDA) - 5 [2]

Sample:

```
// Refactored Further

public class Client
{
    public void AlertService(List<CpuMonitor> cpuMonitors)
    {
        foreach (var cpuMonitor in cpuMonitors)
        {
            cpuMonitor.Sample();
        }
    }
}
```

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

You Ain't Gonna Need It (YAGNI) - 1 [2]

YAGNI emerged as one of the key principles of Extreme Programming. Put another way, the principle states: "Always implement things when you actually need them, never when you just foresee that you may need them."

It maximizes the amount of unnecessary work that is left undone, which is a great way to improve developer productivity and product simplicity.

In some ways, you can think of YAGNI as being similar to Just-In-Time manufacturing.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

You Ain't Gonna Need It (YAGNI) - 2 [2]

As a result, YAGNI is closely related to the KISS principle. The overall design of the system can be kept simpler for longer by deferring adding features and complexity until they are actually required.

And what about that feature you thought you'd need? Usually, it turns out that you didn't need it in the first place, or that when you do, your understanding of how to design it will be superior to if you tried to guess how it would be used earlier in the project.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Law of Demeter (LoD) - 1

The LoD is also known as the Principle of Least Knowledge. Ian Holland proposed it as a software design guideline related to Loose Coupling.

This law mentions the following basic rules

- Don't talk to “Strangers”
- Do not do method chaining communication with other objects. By doing this it increases coupling.
- Only talk to your immediate friends.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Law of Demeter (LoD) - 2

The primary goal of this law is to prevent external objects from gaining access to another object's internals because their state can be changed in an undesirable way.

The below rules should be followed to obey the LoD:

- Apply Aggregation
- Apply Composition
- Use good encapsulation at each level.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Loose Coupling - 1 [8]

Modules should be as independent as possible from other modules, so that changes to a module don’t heavily impact other modules.

By aiming for loose coupling, you can easily make changes to the internals of modules without worrying about their impact on other modules in the system.

It makes it easier to design, write, and test code since our modules are not interdependent on each other. We also get the benefit of easy to reuse and compose-able modules. Problems are also isolated to small, self-contained units of code.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Loose Coupling - 2 [8]

High coupling would mean that your module knows way too much about the inner workings of other modules.

The changes are hard to coordinate and make modules brittle. If Module A knows too much about Module B, changes to the internals of Module B may break functionality in Module A.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

High Cohesion - 1 [8]

Cohesion often refers to how the elements of a module belong together. Related code should be close to each other to make it highly cohesive.

Easy to maintain code usually has high cohesion. The elements within the module are directly related to the functionality that module is meant to provide.

By keeping high cohesion within our code, we end up trying DRY code and reduce duplication of knowledge in our modules.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

High Cohesion - 2 [8]

We can easily design, write, and test our code since the code for a module is all located together and works together.

Low cohesion would mean that the code that makes up some functionality is spread out all over your code-base.

Not only is it hard to discover what code is related to your module, it is difficult to jump between different modules and keep track of all the code in your head.

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Sample source code

Github public repository with code samples Before and After applying the principles:

https://github.com/AbstractSoft/design_principles

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Summary

It is critical to follow good principles consistently in order to write good code. This means applying coding principles whether or not a project is nearing a critical deadline.

The key to writing high-quality software is to avoid tight coupling! The rest will follow…

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Q & A

For any questions or requests:
eduard.ghergu@professional-programmer.com
https://github.com/AbstractSoft

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

Thank you!
See you next time!

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com

References

1. https://en.wikipedia.org/wiki/Single-responsibility_principle
2. https://deviq.com
3. https://dotnettutorials.net
4. Clean Code, Robert C. Martin
5. https://dotnetcodr.com/
6. https://www.geeksforgeeks.org/an-introduction-to-software-development-design-principles/
7. https://www.c-sharpcorner.com/article/the-law-of-demeter/
8. https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6
9. https://softwareengineering.stackexchange.com/questions/187457/what-is-the-principle-of-least-astonishment
10. https://medium.com/@peterlee2068/software-design-principles-every-programmer-should-know-c164a83c6f87

Eduard Ghergu, Eng., PhD.
www.professional-programmer.com