C# Developer. Professional

SOLID

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Герасименко Антон

Немного о себе:

- Работаю в компании “Интелком лайн” fullstack lead dev

- Код: в основном на dotnet, с 2010, typescript

Контакты:

Правила вебинара

Активно
участвуем

Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения

Индивидуально

Время, необходимое
на активность

Пишем в чат

Говорим голосом

Документ

Ответьте себе или
задайте вопрос

Маршрут вебинара

- Знакомство
- Зачем нужны и как появились
- Примеры
- Каждый принцип. Теория
- Примеры
- Рефлексия

Цели вебинара

К концу занятия вы сможете

1. Назвать и объяснить принципы SOLID
2. Превращать плохой код в хороший

Смысл

Зачем вам это уметь

1. Чтобы ваш код не вызывал головной боли при прочтении
2. Чтобы проект не пришел через полгода в состояние “проще все написать с нуля”

1. Готовы ли кодить сегодня на занятии?
2. Какие принципы SOLID Вы уже знаете?

Ставим “+”, если да
“-”, если нет для каждого вопроса

Принципы, подходы

? Какие принципы и подходы вы знаете

? Какие принципы и подходы вы знаете

GRASP

KISS

YAGNI

DRY

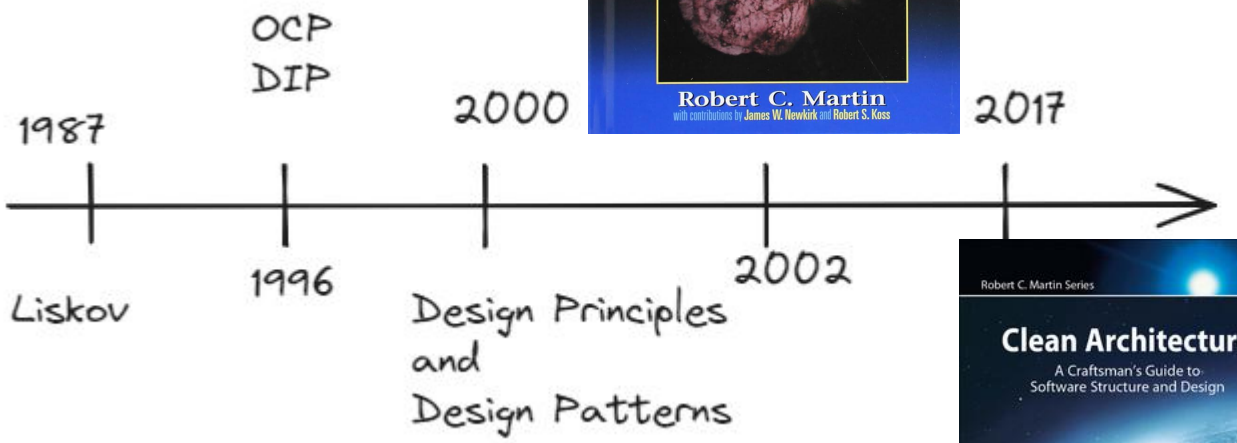
...

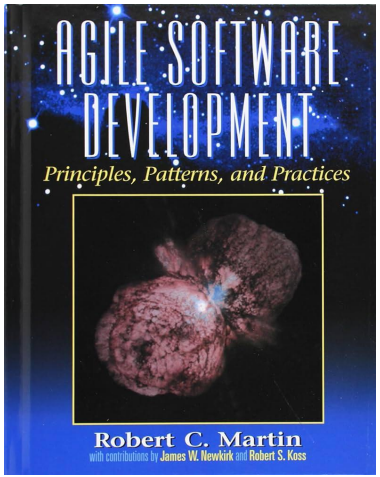
? Какие принципы и подходы вы знаете

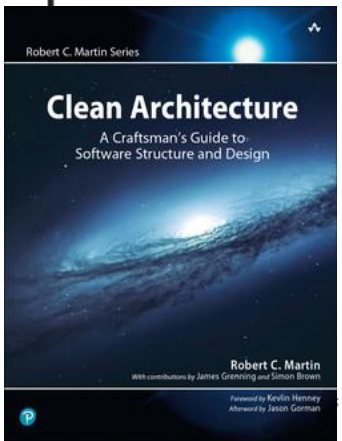
GRASP
KISS
YAGNI
DRY
…

SOLID

SOLID история







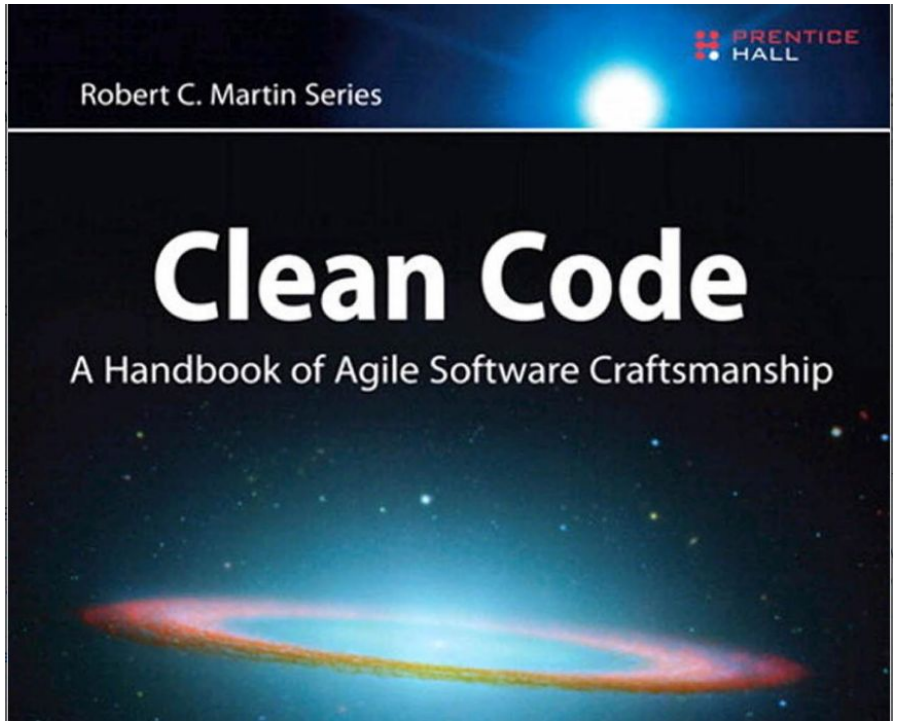
http://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

SOLID. Что это?

Как расшифровать SOLID

1) **S**ingle responsibility — принцип единственной ответственности
2) **O**pen-closed — принцип открытости / закрытости
3) **L**iskov substitution — принцип подстановки Барбары Лисков
4) **I**nterface segregation — принцип разделения интерфейса
5) **D**ependency inversion — принцип инверсии зависимостей

Кто и зачем их придумал



1. Симптомы плохого кода

1) Он замедляет работу
2) Он плохо читаемый, непонятный
3) Fragility. Изменения в одном месте ломают что-то в куче других мест
4) Rigid. Любое небольшое изменение влечет за собой массу изменений повсюду
5) Невозможность переиспользовать подходящий код, так как он делает ещё много чего неподходящего и приходится писать заново

2. Coupling. Dependency

Codependipity

Single responsibility

Принцип единственной ответственности

«Каждый программный модуль должен иметь только одну причину для изменения».

Single responsibility

```
public class UserService
{
    public void Register(string email, string password)
    {
        if (!ValidateEmail(email))
            throw new ValidationException("Email is not an email");
            var user = new User(email, password);

            SendEmail(new MailMessage("mysite@nowhere.com", email) { Subject="HEllo foo" });
    }
    public virtual bool ValidateEmail(string email)
    {
      return email.Contains("@");
    }
    public bool SendEmail(MailMessage message)
    {
      _smtpClient.Send(message);
    }
}
```

Open/Closed Principle

Принцип открытости / закрытости

«Модули должны быть открыты для расширения, но закрыты для модификаций».

```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
```

```
public class AreaCalculator {
    public double TotalArea(Rectangle[] arrRectangles)
    {
        double area;
        foreach(var objRectangle in arrRectangles)
        {
            area += objRectangle.Height * objRectangle.Width;
        }
        return area;
    }
}
```

```
public class Circle{
   public double Radius {get;set;}
}
public class AreaCalculator
{
   public double TotalArea(object[] arrObjects)
   {
      double area = 0;
      Rectangle objRectangle;
      Circle objCircle;
      foreach(var obj in arrObjects)
      {
         if(obj is Rectangle)
         {
            area += obj.Height * obj.Width;
         }
         else
         {
            objCircle = (Circle)obj;
            area += objCircle.Radius * objCircle.Radius * Math.PI;
         }
      }
      return area;
   }
}
```

```
public abstract class Shape
{
    public abstract double Area();
}
```

```
public class AreaCalculator
{
   public double TotalArea(Shape[] arrShapes)
   {
      double area=0;
      foreach(var objShape in arrShapes)
      {
         area += objShape.Area();
      }
      return area;
   }
}
```

```
public class Rectangle: Shape
{
   public double Height {get;set;}
   public double Width {get;set;}
   public override double Area()
   {
      return Height * Width;
   }
}
public class Circle: Shape
{
   public double Radius {get;set;}
   public override double Area()
   {
      return Radius * Radus * Math.PI;
   }
}
```

The Liskov Substitution Principle

Принцип подстановки Барбары Лисков

Производные классы должны быть доступны через интерфейс базового класса, при этом пользователю не обязательно знать разницу.

Принцип подстановки Барбары Лисков

Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты производных классов, не зная об этом.

Принцип подстановки Барбары Лисков

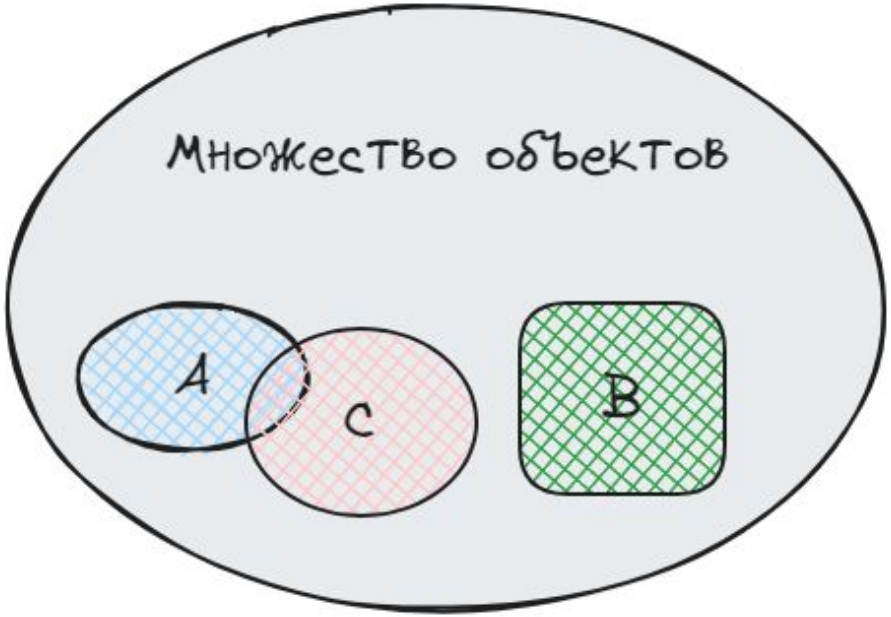
Предусловия в подклассе не могут быть усилены.
Постусловия в подклассе не могут быть ослаблены.

Принцип подстановки Барбары Лисков

Это критерий, который описывает правильное использование полиморфизма и, в частности, наследования.

=> у нас правильная иерархия типов

Принцип подстановки Барбары Лисков



```
public class SqlFile
{
   public string FilePath {get;set;}
   public string FileText {get;set;}
   public string LoadText()
   {
      /* Code to read text from sql file */
   }
   public string SaveText()
   {
      /* Code to save text into sql file */
   }
}
```

```
public class SqlFileManager
{
   public List<SqlFile> lstSqlFiles {get;set}

   public string GetTextFromFiles()
   {
      StringBuilder objStrBuilder = new StringBuilder();
      foreach(var objFile in lstSqlFiles)
      {
         objStrBuilder.Append(objFile.LoadText());
      }
      return objStrBuilder.ToString();
   }
   public void SaveTextIntoFiles()
   {
      foreach(var objFile in lstSqlFiles)
      {
         objFile.SaveText();
      }
   }
}
```

```
public class SqlFileManager
{
   public List<SqlFile? lstSqlFiles {get;set}
   public string GetTextFromFiles()
   {
      StringBuilder objStrBuilder = new StringBuilder();
      foreach(var objFile in lstSqlFiles)
      {
         objStrBuilder.Append(objFile.LoadText());
      }
      return objStrBuilder.ToString();
   }
   public void SaveTextIntoFiles()
   {
      foreach(var objFile in lstSqlFiles)
      {
         //Check whether the current file object is read-only or not.If yes, skip ca
         // SaveText() method to skip the exception.

         if(! objFile is ReadOnlySqlFile)
         objFile.SaveText();
      }
   }
}
```

```
public interface IReadableSqlFile
{
   string LoadText();
}
public interface IWritableSqlFile
{
   void SaveText();
}
```

```
public class SqlFileManager
{
   public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
   {
      StringBuilder objStrBuilder = new StringBuilder();
      foreach(var objFile in aLstReadableFiles)
      {
         objStrBuilder.Append(objFile.LoadText());
      }
      return objStrBuilder.ToString();
   }
   public void SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)
   {
   foreach(var objFile in aLstWritableFiles)
   {
      objFile.SaveText();
   }
   }
}
```

Interface Segregation Principle (ISP)

Принцип разделения интерфейса

Клиенты не должны быть вынуждены реализовывать интерфейсы, которые они не используют.
Вместо одного толстого интерфейса предпочтительнее использовать множество небольших интерфейсов, основанных на группах методов, каждый из которых обслуживает один подмодуль.

```
public Interface ILead
{
   void CreateSubTask();
   void AssginTask();
   void WorkOnTask();
}
public class TeamLead : ILead
{
   public void AssignTask()
   {
      //Code to assign a task.
   }
   public void CreateSubTask()
   {
      //Code to create a sub task
   }
   public void WorkOnTask()
   {
      //Code to implement perform assigned task.
   }
}
```

```
public class Manager: ILead
{
   public void AssignTask()
   {
      //Code to assign a task.
   }
   public void CreateSubTask()
   {
      //Code to create a sub task.
   }
   public void WorkOnTask()
   {
      throw new Exception("Manager can't work on Task");
   }
}
```

```
public interface IProgrammer
{
    void WorkOnTask();
}
```

```
public interface ILead
{
    void AssignTask();
    void CreateSubTask();
}
```

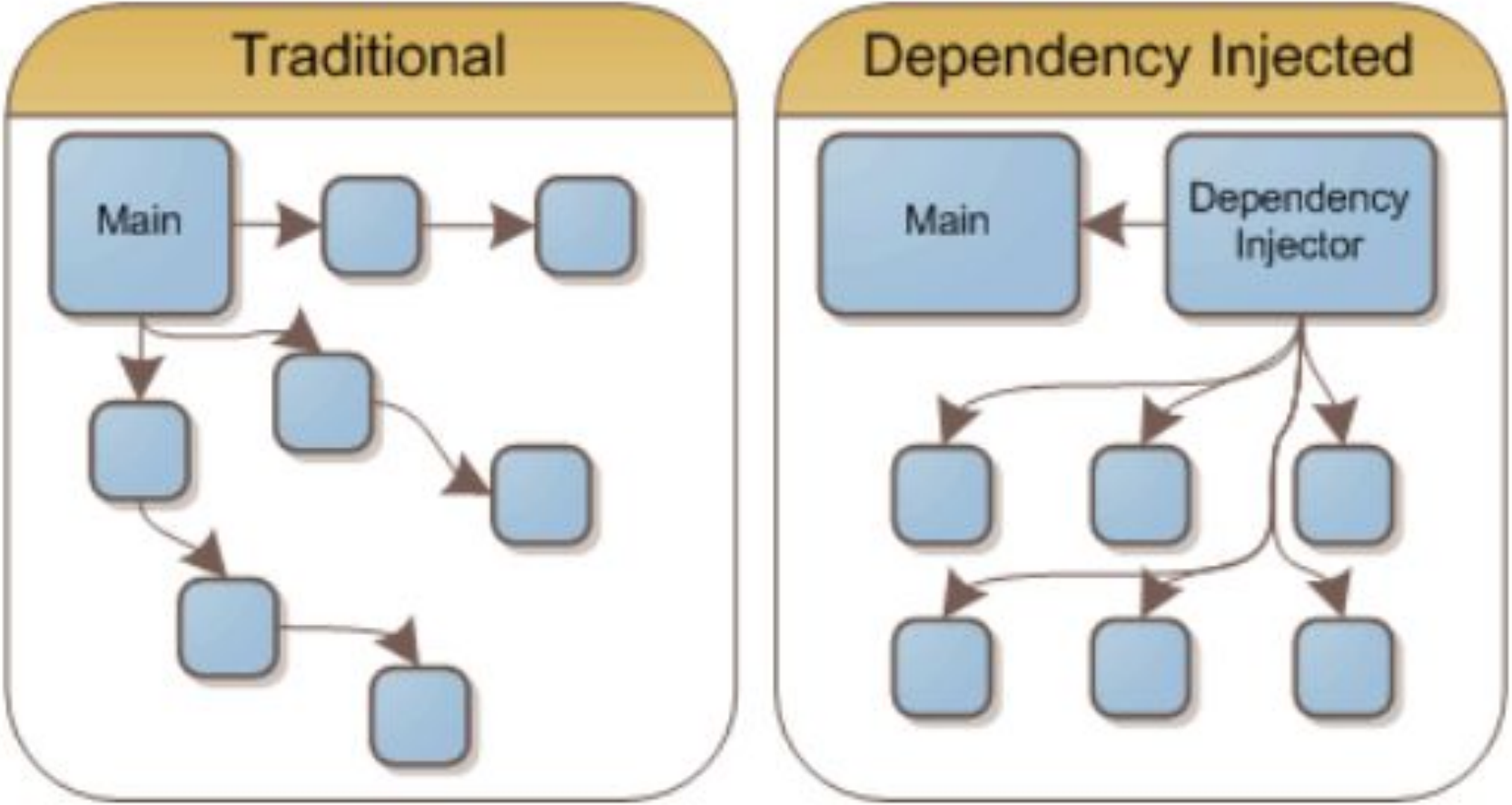
```
public class TeamLead: IProgrammer, ILead
{
    public void AssignTask()
    {
        //Code to assign a Task
    }
    public void CreateSubTask()
    {
        //Code to create a sub task from a task.
    }
    public void WorkOnTask()
    {
        //code to implement to work on the Task.
    }
}
```

```
public class Programmer: IProgrammer
{
    public void WorkOnTask()
    {
        //code to implement to work on the Task.
    }
}
public class Manager: ILead
{
    public void AssignTask()
    {
        //Code to assign a Task
    }
    public void CreateSubTask()
    {
    //Code to create a sub taks from a task.
    }
}
```

Dependency Inversion Principle

Принцип инверсии зависимостей

Высокоуровневые компоненты не должны зависеть от низкоуровневых компонент. И те, и те должны зависеть от абстракций.
Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.


Traditional
Main
Dependency Injected
Main
Dependency Injector

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Практика

Практика

Сможем ли мы теперь определить, какой принцип или принципы нарушаются в данном классе?

- Пример 1

  Решение Решение

- Пример 2

  Решение Решение Решение

- Пример 3

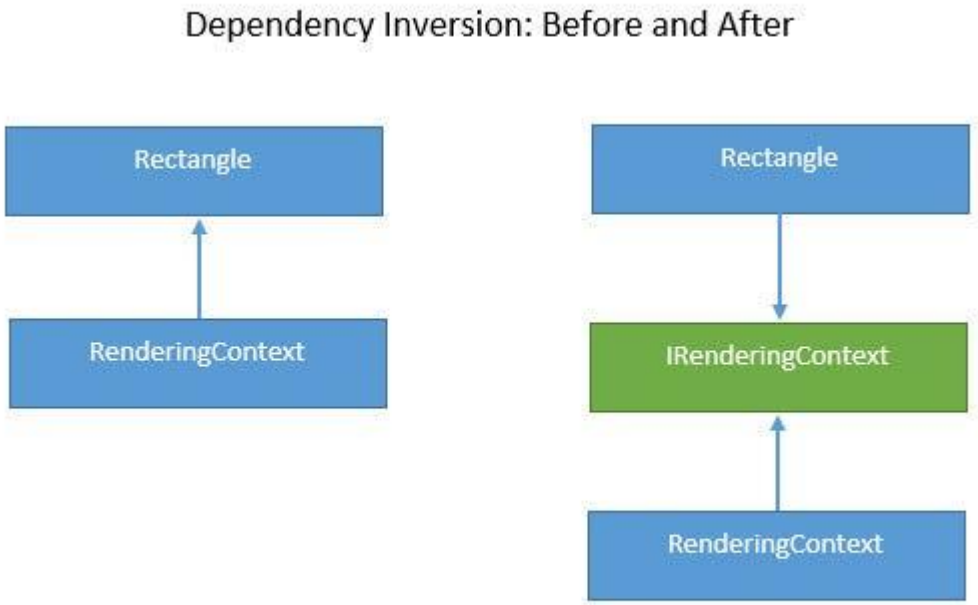
  Решение

- Пример 4

  Решение

Практика

Ответ 4



Рефлексия

Цели вебинара

Проверка достижения целей

1. Назвать и объяснить принципы SOLID
2. Превращать плохой код в хороший

Вопросы для проверки

https://docs.google.com/document/d/119k6TLGTEMexww0CuyV-S3-iqqGiiFrp05SoQwZZf40/edit?usp=sharing

Слайд с тезисами

Подведем итоги

1. Узнали что такое принципы SOLID
2. Выяснили какие проблемы они призваны решать

Материалы

Design Patterns https://refactoring.guru/ru/design-patterns/catalog
Design Principles https://principles.design/examples/
Programming principles
https://en.wikipedia.org/wiki/Category:Programming_principles
SOLID https://github.com/thangchung/clean-code-dotnet#solid
Dependency injection in .NET
https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection
Robert C. Martin’s books - https://en.wikipedia.org/wiki/Robert_C._Martin#Publications

- Блог Александра Бындю: Принцип замещения Лисков
- SOLID book

Рефлексия

С какими впечатлениями уходите с вебинара?

Как будете применять на практике то,
что узнали на вебинаре?