Object interrelationships

2. Software contractions

-Говорим за т.нар. 'Design by Contract (DbC)' методология, въведена в ООП от Bertrand Meyer през 2008.Тя подменя на по-високо ниво if-then-throw структурния образец за тестване на предусловия.

-.NET framework за първи път от версия 4 имплементира методологията (чрез библиотеката Code Contract – част от framework). Освен тук , тя се поддържа и в JavaScript, Perl, Java, Ruby, Silverlight 4.

Първи пример- с използване на традиционна технология:

```
// The Calculator Class Implementing the
// If-Then-Throw Pattern
public class Calculator
{
   public Int32 Sum(Int32 x, Int32 y)
   {     // Check input values
      if (x <0 || y <0)
         throw new ArgumentException();
      // Perform the operation
      return x + y;
   }

   public Int32 Divide(Int32 x, Int32 y)
   {       // Check input values
      if (x < 0 || y < 0)
         throw new ArgumentException();
      if (y == 0)
         throw new ArgumentException();
 // Perform the operation
      return x / y;
   }
}
```

Подобрения: тест и над изходните данни

```
public class Calculator
{    public Int32 Sum(Int32 x, Int32 y)
   {
      // Check input values
      if (x <0 || y <0)
         throw new ArgumentException();

      // Perform the operation
      Int32 result = x + y;

      // Check output
      if (result <0)
         throw new ArgumentException();
       return result;               }

   public Int32 Divide(Int32 x, Int32 y)
   {       // Check input values
      if (x < 0 || y < 0)
         throw new ArgumentException();
      if (y == 0)
         throw new ArgumentException();

      // Perform the operation
      Int32 result = x / y;

      // Check output
      if (result < 0)
         throw new ArgumentException();
       return result;
   }            }
```

-Но какво става с теста, ако имаме множество exit points?
-А ако някои от тези exit points се влияят от други резултати, където също може да възникнат грешки?

Нещата се усложняват откъм логичска, но главно откъм структурна гледна точка.

Въвеждане на ‘ Code Contracts’

Във .NET Framework 4, Code Contracts e framework създаващ по-подходящ синтаксис за да се изразят подобни условия.
Code Contracts поддържа 3 типа contracts:

1. preconditions,
2. Postconditions,
3. invariants.

Preconditions се занимават с предусловия, които следва да бъдат проверени за да може метод да се изпълни.
Postconditions се занимават с условия, които следва да бъдат проверени в момента, когато метод е завършил изпълнението си – коректно или с хвърлено изключение.
Invariants описват условия, които следва да са винаги true за времето на която и да е инстанция на класа. Казано по друг начин, invariant указват условие, което следва да се поддържа през времето на всяко взаимодействие между класа и негов клиент — което значи при всяко изпълнение на public members, ***включително и на constructors.***

Code Contracts API се състои от static методи на класа Contract.
Методът Requires() се ползва за preconditions , а Ensures() за postconditions.

Ето предния пример с използване на тези конструкции:

```
using System.Diagnostics.Contracts;

public class Calculator
{
    public Int32 Sum(Int32 x, Int32 y)
    {
        Contract.Requires<ArgumentOutOfRangeException>(x >= 0 && y >= 0);
        Contract.Ensures(Contract.Result<Int32>() >= 0);

        if (x == y)
            return 2 * x;

        return x + y;
    }

    public Int32 Divide(Int32 x, Int32 y)
    {   Contract.Requires<ArgumentOutOfRangeException>(x >= 0 && y >= 0);
        Contract.Requires<ArgumentOutOfRangeException>(y > 0);
        Contract.Ensures(Contract.Result<Int32>() >= 0);

        return x / y;
    }
}
```

Въведен е т. нар . Code Contracts rewriter, който пробразува кода на етап компилация след анализ на целта на preconditions или postconditions. Той разширява автоматично кода и поставя ново-генерираните блокове там, където им е мястото.
Това означава, че

разработчикът не се грижи къде да постави postcondition и дали ги е дублирал някъде в кода (особено при добавяне на нова exit point).

Синтаксис на конструкциите в .NET

Contract.Requires<TException> (Boolean condition)

Методът има няколко overloads , които могат да се използват.

Contract.Ensures(Boolean condition)

При preconditions изразът съдържа input параметри. Допуска се и друг method или property от същия клас. Към метода следва да се добави и атрибут 'Pure' за да се отбележи, че няма да се променят данни. Пропъртитата (getters) се подразбира че са Pure.

...

При postconditions обикновено се реферира и друга информация, като например връщана стойност, или начална стойност на local variable. Това става с конструкции :

Contract.Result<T> ....

- за да провери стойност (от тип T) връщана от метод и

Contract.OldValue<T> .....

- за да вземе стойност (съхранявана в специална променлива) от началото на изпълнението на метода.

Има възможност да се провери и условието в момент на генериране на exception (ако това стане по време на изпълнение на метода).

Това става с:

Contract.EnsuresOnThrow<TException> ....

Ето вече по-добре структуриран код:

```
public class Calculator
{
    public Int32 Sum(Int32 x, Int32 y)
    {
        // Check input values
        ValidateOperands(x, y);
        ValidateResult();

        // Perform the operation
        if (x == y)
            return x<<1;
        return x + y;
    }

    public Int32 Divide(Int32 x, Int32 y)
    {
        // Check input values
        ValidateOperandsForDivision(x, y);
        ValidateResult();

        // Perform the operation
        return x / y;
    }

[ContractAbbreviator]
    private void ValidateOperands(Int32 x, Int32 y)
    {
        Contract.Requires<ArgumentOutOfRangeException>(x >= 0 && y >= 0);
    }
 [ContractAbbreviator]
    private void ValidateOperandsForDivision(Int32 x, Int32 y)
    {
        Contract.Requires<ArgumentOutOfRangeException>(x >= 0 && y >= 0);
        Contract.Requires<ArgumentOutOfRangeException>(y > 0);
    }

    [ContractAbbreviator]
    private void ValidateResult()
    {
        Contract.Ensures(Contract.Result<Int32>() >= 0);
    }
}
```

забележка:

Атрибут ContractAbbreviator указва на компилатора че следва да интерпретира кода по указания в предните слайдове начин.

За да се ползва този атрибут, той следва да се дефинира (в сегашната версия това не е). То става с подвключване на готов конфигурационен файл.

Code Contracts API

Класът Contract е включен в .NET Framework 4 - в mscorlib assembly.

Visual Studio 2010 предоставя готови *configuration pages*.
За всеки нов проект, следва да се укаже опция:

‘*enable runtime checking of contracts*’.

Ще са нужни също и tools от DevLabs Web site:

Runtime tools включват

Code Contracts rewriter

interface generator
както и
static checker

Инварианти

Инвариантът е условие, което винаги е истина в обкръжението на определен контекст.
Отнесено към ООП – условието следва да е истина за всяка инстанция на класа

Пример:

Имаме клас представляващ новини в сайт. Вероятно всяка новина изисква ‘заглавие’, ‘ резюме’ и ‘дата на публикуване’, която да е валидна. Понятието ‘валидна’ следва да е дефинирано в контекста.
Можете да представите тези полета като инварианти (няма значение дали са public, protected и т.н.).

Def:
В .NET 4
инвариантният контракт за даден клас е колекция от условия, които се задържат инстина за периода на съществуване на инстанцията на класа.

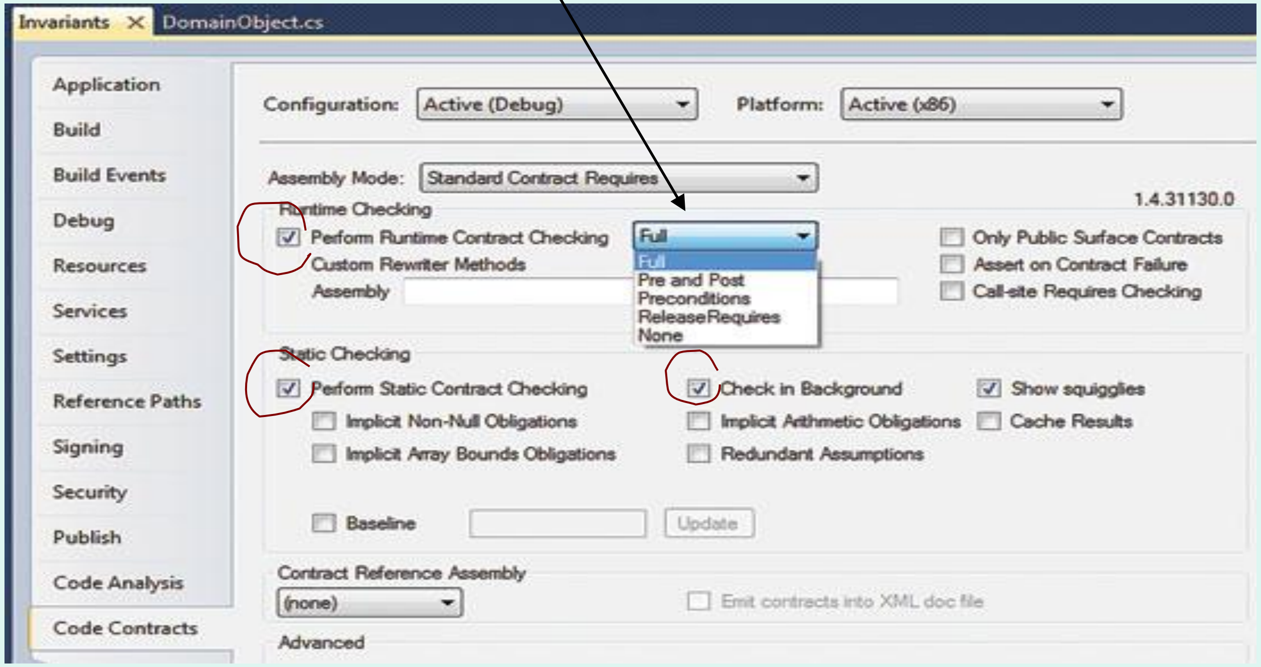
Preconditions контрактите се ползват от Викащата страна !
Poscondition контрактите и invariants се ползват от самия клас и наследниците му.

- Инвариантният контракт се дефинира чрез 1 или повече методи.
- Те са private, void и с атрибут (както ще видим в примера).
- не съдържат друг код , освен условието за проверка.

Пример:

```
public class News      {
  public String Title {get; set;}
  public String Body {get; set;}

  [ContractInvariantMethod]
  private void ObjectInvariant()
  {
    Contract.Invariant(!String.IsNullOrEmpty(Title));
    Contract.Invariant(!String.IsNullOrEmpty(Body));
   }
                                          }
```

За да работи проверката, следва да разрешите ‘full runtime checking’ опциите за проекта си:


Invariants × DomainObject.cs

Application
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing
Security
Publish
Code Analysis
Code Contracts

Configuration: Active (Debug) Platform: Active (x86)

Assembly Mode: Standard Contract Requires 1.4.31130.0

Runtime Checking
[✓] Perform Runtime Contract Checking Full (Full / Pre and Post / Preconditions / ReleaseRequires / None)
Custom Rewriter Methods
Assembly
[ ] Only Public Surface Contracts
[ ] Assert on Contract Failure
[ ] Call-site Requires Checking

Static Checking
[✓] Perform Static Contract Checking
[ ] Implicit Non-Null Obligations
[ ] Implicit Array Bounds Obligations
[✓] Check in Background
[ ] Implicit Arithmetic Obligations
[ ] Redundant Assumptions
[✓] Show squiggles
[ ] Cache Results
[ ] Baseline Update

Contract Reference Assembly
(none)
[ ] Emit contracts into XML doc file

Advanced

Когато това е направено, опитваме с оператор:
var n = new News();

Получваме contract failed exception ? Така е защото в конструктора липсва инициализация.

-Инвариантите са асоциират с концепцията за 'factory'. Factory е просто public метод, отговорен за създаване на инстанция на класа (която за да е успешна -значи обектът е във валидно състояние !)

-За конкретният пример: invariants се проверяват на изхода от всеки public метод (вкл. конструктор и set properties). Т.е. следва да попроменим конструктора и опишем 'factory' метод:

```
public class News
{  public News(String title, String body)
  {
    Contract.Requires<ArgumentException>(
                                        !String.IsNullOrEmpty(title));
    Contract.Requires<ArgumentException>(
                                        !String.IsNullOrEmpty(body));


    Title = title;
    Body = body;
  }

  public String Title { get; set; }
  public String Body { get; set; }

  [ContractInvariantMethod]
  private void ObjectInvariant()
  {
    Contract.Invariant(!String.IsNullOrEmpty(Title));
    Contract.Invariant(!String.IsNullOrEmpty(Body));
  }          }
```

Методът- Factory е същия като конструктора.
Той, обаче, е static и с описателно име.

Сега вече, ако имаме оператор от вида:
var n = new News("Title", "This is the news");
Всичко ще е ОК! Инстанцията ще се създаде и върне обект – казваме: " in a state that meets invariants."

Ако напишете:
var n = new News("Title", "This is the news");
n.Title = "";
Ще получите exception защото title не отговаря на условието по време на exit от 'setter' метода.

Има и друг проблем: инвариантите се проверяват в края на public методите. Но в тялото, временно статусът им може да стане невалиден. С инварианти можете да следите само преди и след изпълнение на public метод.
Има отделен MS Static Code Checker който следи за присвоявания в тялото противоречащи на инвариантните ограничения.

Contract Inheritance – примерен подход:

```
public abstract class DomainObject
{  public abstract Boolean   IsValid();

  [Pure]
  private Boolean IsValidState()
  {
    return IsValid();
  }

  [ContractInvariantMethod]
  private void ObjectInvariant()
  {
    Contract.Invariant(IsValidState());
  }
```

Пример на клас с инвариантен метод.
Инвариантният контракт е реализиран с private метод – pure (т.е. непроменящ състоянието). Той от своя страна вика public, abstract метод, който наследник може да дефинира за да опише своите собствени инварианти.
Например така:

```
public class Customer : DomainObject
{
  private Int32  Id;
  private String  CompanyName, Contact;

  public Customer(Int32 id, String company)
  {
    Contract.Requires(id > 0);
    Contract.Requires(company.Length > 5);
    Contract.Requires(!String.IsNullOrWhiteSpace(company));

    Id = id;
    CompanyName = company;
  }
  ...
  public override bool IsValid()
  {
    return (Id > 0 && !String.IsNullOrWhiteSpace(CompanyName));
  }
}
```

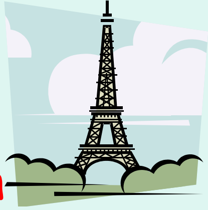
Сега вече всичко изглежда ОК:

```
var c = new Customer(1, “Наков");          //ОК
```

Но има bug: вика се конструкторът на ‘DomainObject’, т.е инвариантът се проверява. IsValid() е виртуален, т.е. изпълнява се реализацията в ‘Customer’. Проверката, обаче, става преди инстанцията да е напълно инициализирана. Вдига се exception (*в последната версия на .NET това е променено – invariant checking в конструктори се отлага докато и най-крайният от веригата на вложените се повика).***

Добре е да спазате общо правило: не викайте виртуални методи в тялото на конструктор.

Modern trip, based on the OOP theory:
tips around design, philosophy, classes, Interfaces, virtuality, abstraction



The question is: how many times do you start writing a brand new application from nothing versus the number of times you start by adding new functionality to an existing codebase? Chances are good that you spend far more time adding new features to an existing codebase.

Then ask yourself another question: is it easier to write all new code or to make changes to existing code? Modifying old code adds the risk of breaking existing functionality.

SOLID is a popular acronym that results from the initials of five key principles of software design including Single responsibility (2), Open/Closed (1), Interface segregation (3) and Dependency inversion (4). The L in SOLID stands for the Liskov substitution principle

So you generally work on an existing codebase, but yet it's easier to write all new code than it is to change old code. This is where the Open Closed Principle comes into play. To paraphrase, the Open Closed Principle is stated as:

1 open/closed principle

software entities should be open for extension but closed for modification.

2 Single Responsibility Principle

In following the Open Closed Principle, we want to be able to write a class or a method and then turn my back on it, comfortable that it does its job and I won't have to go back and change it.
Follow the related Single Responsibility Principle:

<u>a class should have one, and only one, reason to change</u>

One of the easiest ways to write classes that never have to change is to write classes that only do one thing. The code shows a sample that does not follow the Single Responsibility Principle.

This class /function

are doing to much

```
public class OrderProcessingModule
{ …
public void Process(OrderStatusMessage orderStatusMessage)
                                        { // Get the connection string from configuration
string connectionString =                                        ….;

using (SqlConnection connection = new SqlConnection(connectionString)) {
// go get some data from the database
                                                                 {…. }
// Apply the changes to the Order from the OrderStatusMessage
updateTheOrder(order);
// International orders have a unique set of business rules
if (order.IsInternational)                                       { …. }
// We need to treat larger orders in a special manner
else if (order.LineItems.Count > 10)                             { …. }
// Smaller domestic orders else                                  { …. }
 // Ship the order if it's ready
if (order.IsReadyToShip())                                       { ..;
 // Transform the Order object into a Shipment
                                        …. }
}
```

In terms of the Open/ Closed Principle, by dividing the business logic and data access responsibilities into separate classes, you should be able to change either concern independently without affecting the other.

The point of the Single Responsibility Principle isn't just to write smaller classes and methods. The point is that each class should implement a cohesive set of related functions. An easy way to follow the Single Responsibility Principle is to constantly ask yourself whether

every method and operation of a class is directly related to the name of that class.

If you find some methods that do not fit with the name of the class, you should consider moving those methods to another class.

3. The Chain of Responsibility Pattern

Business rules will probably face more changes throughout the lifecycle of a codebase than any other part of the system.

In the *OrderProcessingModule* class, there was quite a bit of branching logic for order processing based on what type of order was received:

```
if (order.IsInternational) {
  processInternationalOrder(order);
}

else if (order.LineItems.Count > 10) {
  processLargeDomesticOrder(order);
}

else {
  processRegularDomesticOrder(order);
}
```

To that end, you can move closer to the Open/Closed Principle for the order processing example by using a form of the Chain of Responsibility pattern, as demonstrated.

The first thing I did was to make every conditional branch in the original OrderProcessingModule into a separate class that implements the IOrderHandler interface:

```
public interface IOrderHandler
{
   void ProcessOrder(Order order);
   bool CanProcess(Order order);
}
```

I would then write a separate implementation of IOrderHandler for each type of Order, including the logic that basically says, "I know what to do with this Order, let me handle it."

```
public class OrderProcessingModule
{
                private IOrderHandler[] _handlers;

                public OrderProcessingModule()
                {
                _handlers = new IOrderHandler[]
                    {
                                new InternationalOrderHandler(),
                                new SmallDomesticOrderHandler(),
                                new LargeDomesticOrderHandler(),
                   };
                }


…
  public void Process (OrderStatusMessage orderStatusMessage, Order order)
                {
                // Apply first change to the 'Order', using the OrderStatusMessage
                updateTheOrder(order);

                // Find the first IOrderHandler '_handlers[x]' that "knows" how to process this Order
                ….;
                _handlers[x].ProcessOrder(order);
                  }

  private void updateTheOrder(Order order) {..  }                    //implementation
}
```

Introducing a <u>chain of responsibility.</u>
We have multiple realizations of the interface IOrderHandler

3 different implementations of the interface

Let's say that, at a later time, we have to add support for government orders in the system.
With the ***Chain of Responsibility pattern***:

(1) we can write a completely new interface method called GovernmentOrderHandler().

Once we are satisfied that GovernmentOrderHandler() works the way it's supposed to

(2) We can add the new government order processing rules by a one-line change to the constructor function of OrderProcessingModule:

```
public OrderProcessingModule()
{
  _handlers = new IOrderHandler[]        {
                new InternationalOrderHandler(),
                new SmallDomesticOrderHandler(),
                new LargeDomesticOrderHandler(),
                new GovernmentOrderHandler(),        //the new-added handler
                                         };
}
```

I was able to add the government order rules with much less risk of destabilizing the other types of orders than we would have faced if a single class had implemented all of the various types of order handling.

4. Double Dispatch pattern

What if the steps become more complicated in the future? What if pure polymorphism just isn't enough to allow for all the possible variations coming up in the future?

We can use a pattern called Double Dispatch to pull out the variation into the subclasses in such a way that we don't break existing interface contracts.

For example, let's say that I am building a composite desktop application that displays one screen at a time in some sort of main panel. Every time I open a new screen in the application I need to a number of things. I might need to change the available *menus*, check the state of the screens already open, do any number of things to customize the display of the whole screen and, at the end, display the new screen in some way.

If all we need to do is simply to show the View in the ApplicationShell on activation, the code might look like in the next slide

A simple view-based application, **with some** (not more) **abstraction** included:

```
public interface IApplicationShell {
                    void DisplayMainView(object view);
                                                      }

public interface IPresenter          {
  // Just exposes a getter for the inner WinForms UserControl or Form
                   object View { get; }
                                                      }

public class ApplicationController
{
  private IApplicationShell  _shell;

  public ApplicationController(IApplicationShell shell)
                   { _shell = shell;  }

  public void ActivateScreen(IPresenter presenter)
                   {   ….  }
}
```

which shell is to be chosen for the purpose

which set of controls or forms to be shown

This is perfectly workable for a simple application, but what if the application becomes more complicated?

- What if, in the second release, I have new requirements <u>to add menu items</u> to the main shell when some screens are active?

- **What if I want <u>to start showing additional controls</u> in a new pane along the left edge of the main screen for some views, but not all?**

We still want the architecture to be pluggable so that we can add new screens to the application by simply plugging in new Presenters, so the knowledge of these new menu and left pane constructs should go into the existing Presenter abstraction :

```
public interface IApplicationShell
{               void DisplayMainView(object view);
                // New-added behavior
                void AddMenuCommands(MenuCommand[] commands);
                void DisplayInExplorerPane(object paneView);
}
public interface IPresenter
{               object View { get; }
                // New-added properties
                MenuCommand[] Commands{ get; }
                object[] ExplorerViews { get; }                  }
public class ApplicationController
                {
                private IApplicationShell _shell;
                public ApplicationController(IApplicationShell shell)
                                {               _shell = shell;            }

                public void ActivateScreen(IPresenter presenter)
                {               // Setup the new screen
                                _shell.DisplayMainView(presenter.View);

                                // New code
                                _shell.AddMenuCommands(presenter.Commands);
                                foreach (var explorerView in presenter.ExplorerViews)
                                {
                                _shell.DisplayInExplorerPane(explorerView);
                                }
                }               }
```

Trying to extend IPresenter, IAplicationShell

I've added new properties to the IPresenter interface to model the new menu items and any additional controls that might be added to the new left pane. I also added some new members to IApplicationShell for these new concepts. Then I added new code to the ApplicationController.ActivateScreen(IPresenter) method.

So, did this solution conform to the Open Closed Principle? No!

1. I had to modify the IPresenter interface. Since it is an interface, I would have had to find every implementer of the IPresenter interface in my codebase and add empty implementations of these new methods just so that my code could compile again.

2. I also had to modify the ApplicationController class so it knows about all of the new types of customizations each screen might need on the main ApplicationShell.

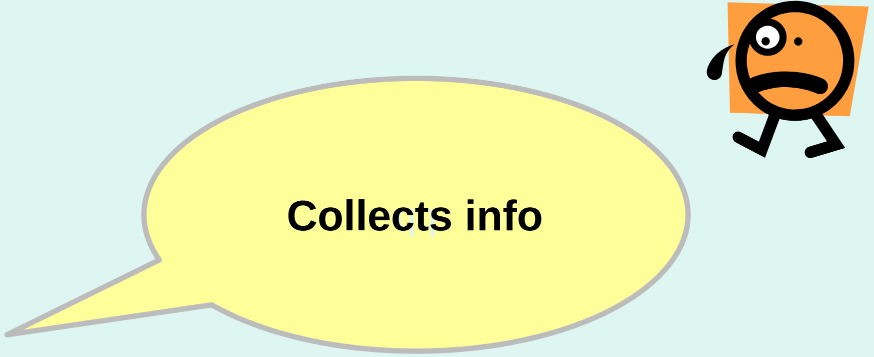
3. Finally, I needed to modify ApplicationShell to support the new shell customizations.

The change to each IPresenter implementation could be alleviated by using an
abstract class called Presenter instead of an interface.

I could just add default implementations to the abstract class as shown into the next slide

And I wouldn't have to change any of the existing Presenter implementations to add the new behavior.

```
public abstract class BasePresenter

{
   public abstract object View { get;}

   // Default implementation of Commands
   public virtual MenuCommand[] Commands
                {
        get
        {  return new MenuCommand[0];     }
                }

   // Default ExplorerViews
   public virtual object[] ExplorerViews
   {
        get
        { return new object[0];  }
   }
}
```



BUT

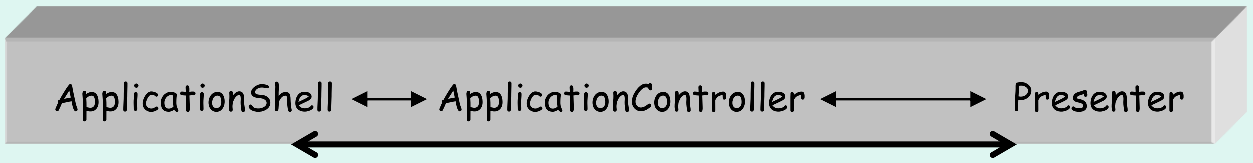
there is another way to move closer to the Open/ Closed Principle that might be cleaner.
Instead of putting getters on the IPresenter or BasePresenter abstraction, I could use the double dispatch pattern.

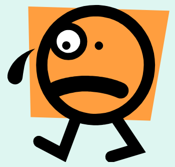
Let see a demonstration of the double dispatch pattern in real life:
We are moved into a new office space. Our networking guy
called us and started telling us what my co-worker should do to connect to.
Finally we just handed the phone off to my co-worker to let them talk directly.

Now, let's do the same thing for the ApplicationController. Instead of the ApplicationController (1) asking each Presenter(2) what needs to be displayed in the ApplicationShell (3), the Presenter will just skip the middle man and tell the ApplicationShell what it needs to do for each screen :

```
public interface IPresenter        {  void SetupView( IApplicationShell shell); }

public class ApplicationController {
  private IApplicationShell _shell;
  public ApplicationController(IApplicationShell shell) {    _shell = shell;  }

  public void ActivateScreen( IPresenter presenter )     {

         // Set up the new screen using Double Dispatch
         presenter.SetupView(_shell);                    // redirect to direct connection
                                          }
                    }
```

dual dispatching – because object&parameter are determined at run-time





I could have made the new changes with fewer modifications to both ApplicationController and each Presenter. I no longer need to touch either the ApplicationController or the Presenter classes to create additional screen concepts.

The architecture is open to be extended for new shell concepts, but the ApplicationController and each individual Presenter classes are closed for modification.

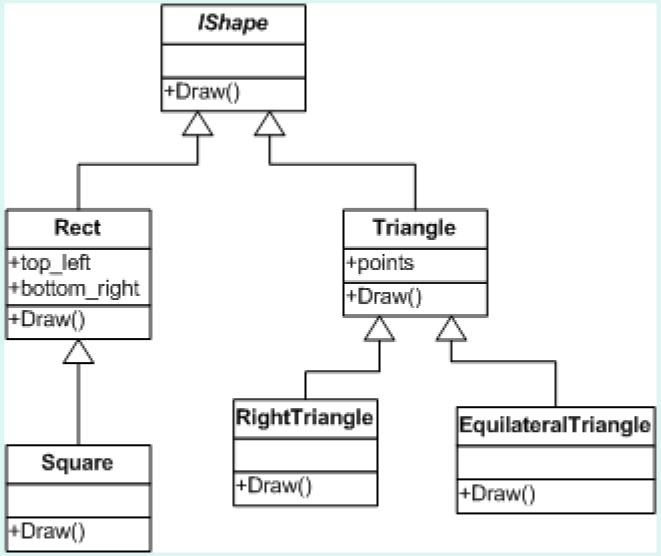
Further formalization of the method:

there is a regular way to determine concrete object type at run-time using so called Visitor pattern.
This (implemented through a method) is *dual dispatching formalization:*

Each visitable object should implement method Visit() that accepts reference to special object - *dispatcher*. Dispatcher should have a set of Dispatch methods that accepts references to concrete types of visitable objects:

To clarify the theory : let consider second example.

Suppose we have a simple hierarchy of geometrical shapes and want to write a function answering the question: "do two (or more) shapes intersect". Shapes hierarchy may look as follows.



Function, that checks for shapes intersections, must work with IShape references, i.e. in polymorphic manner (the classical sollution (1)) . For example:

```
void foo( IShape_& _shape0, IShape_& _shape1 )
                { bool f = CheckIntersection( _shape0, _shape1 );}
...
Rect _ rect( Point_( 1, 1), Point_( 2, 2) );
Triangle _ tri( Point_( 0, 0), Point_( 2, 0), Point_( 0, 2) );
foo( rect, tri );
```

The following more universal solution is better:

dual dispatching solution

```
struct Triangle_;
struct Rect_;
struct  IDispatcher_                                    //possesses a method for all possible types
          {  virtual void Dispatch( Triangle_& _shape ) = 0;
             virtual void Dispatch( Rect_& _shape ) = 0;          };
struct  IShape_
          {  virtual void Visit( IDispatcher_& _dsp ) = 0;};

struct Triangle_ : public IShape_
          {  virtual void Visit( IDispatcher_& _dsp ) {   _dsp.Dispatch( *this );  }};
struct Rect_ : public IShape_
{  virtual void Visit( IDispatcher_& _dsp ) {   _dsp.Dispatch( *this );  }};

struct Dispatcher_ : public IDispatcher_                //implementation of the dispatcher class
          {  virtual void Dispatch( Triangle_& _shape ) {   cout << "This is a triangle." << endl;  }
             virtual void Dispatch( Rect_& _shape )     {   cout << "This is a rect." << endl;  }      };

void PrintType( IShape_& _shape )                  //function that works with all types – run-time detected
          {  Dispatcher_  dsp;
             _shape.Visit( dsp );}                 //dual dispatch (2) solution – because object&parameter
                                                   // are determined at run-time

int main()
          {  Rect_ rect;
             Triangle_ triangle;
          PrintType( rect );                       //the type selection is in run-time
          PrintType( triangle );                   // the type selection is in run-time. Everything is OK
          return 0;}
```

The mentioned method Visit() – determines who had been visited

This programs prints:

```
This is a rect.
This is a triangle.
```

5. Liskov Substitution Principle

The most common manifestation of the Open Closed Principle is using polymorphism

to substitute an existing part of the application with a brand new class.

Let's say that at the beginning of the day you have a class called BusinessProcess whose job is to, well, execute a business process. Along the way, it needs to access data from a data source:

```
public class BusinessProcess
{
  private IDataSource _source;

  public BusinessProcess(IDataSource source) {
                                             _source = source;
                                                 }
...
}

public interface IDataSource
         {
           Entity  FindEntity(long key);
         }
```

The design follows the Open Closed Principle if you can extend the system by swapping out implementations of IDataSource without making any change to the BusinessProcess class.

- You might:
  - start out with a simple XML file-based mechanism;
  - then move to using a database for storage;
  - followed eventually by some sort of caching.

But you still don't want to change BusinessProcess class!

All of that is implemented following the: ***the Liskov Substitution Principle:***
In a nutshell, the Liskov principle states that it should

1/ **always be safe**
to use a subclass in any place where the parent class is expected

It's a developer's responsibility to ensure that it's **safe** to use any derived class in places where the parent class is expected. Notice - "safe."
Plain object orientation makes it possible to use any derived classes in places where the parent class is expected.

"Possible" isn't the same as "safe!"

To fulfill the Liskov principle, you need to adhere to a simple rule:

2/ **The domain of a method can't be shrunk in a subclass.**

Roughly stated, you are following the Liskov Substitution Principle if you can

3/ use any implementation of an abstraction in any place that accepts that abstraction.

The BusinessProcess should be able to use any implementation of IDataSource without modification. BusinessProcess should not know anything about the internals of IDataSource other than what is communicated through the public interface:

Bad version:

```
public class BusinessProcess {
  private IDataSource _source;

  public BusinessProcess(IDataSource source) {
    _source = source;
  }

  public void Process()
   {
    long theKey = 112;

    // Special code if we're using a FileSource
    if (_source is FileSource)  {
      ((FileSource)_source).LoadFile(); }

    try {
      Entity entity = _source.FindEntity(theKey);    }

    catch (System.Data.DataException) {
      // Special exception handling for the DatabaseSource,
      // This is an example of "Downcasting"
      ((DatabaseSource)_source).CleanUpTheConnection();
    }          }          }
```

BuisnessProcess class implementation that cannot abstract IDataSource

This version of the BusinessProcess class has specific logic to bootstrap a FileSource and also relies on knowledge of some specific error handling logic for the DatabaseSource class

Better BusinessProcess version

You create the implementers of IDataSource such that they can handle all of their specific infrastructure needs

```
public class BusinessProcess {
  private readonly IDataSource _source;

  public BusinessProcess(IDataSource source) {
    _source = source;
                                                  }

  public void Process(Message message)    {
    // the first part of the Process() method

    // There is NO code specific to any implementation of   IDataSource here
    Entity entity = _source.FindEntity(message.Key);

    // the last part of the Process() method
                                              }
}
```

Code Contracts and the Liskov Principle

the Liskov principle has a lot to do with software contracts.

<u>The key point is that a derived class can't just add preconditions.</u>
In doing so, it will restrict the range of possible values being accepted for a method, possibly creating runtime failures.

Imagine you have the code :

```
public class Rectangle
{
  public Int32 Width { get; private set; }
  public Int32 Height { get; private set; }

  public virtual void SetSize(Int32 width, Int32 height)
  {
    Width = width;
    Height = height;
  }
}
public class Square : Rectangle
{
  public override void SetSize(Int32 width, Int32 height)
  {
    Contract.Requires<ArgumentException>(width == height);
    base.SetSize(width, width);
  }
}
```

Слага „ограничения“ защото подменя логиката: ползва в SetSize() Width и за ширина и за височина

The class Square inherits from Rectangle and just adds one precondition. At this point, the following code written basically for Rectangle (which represents a possible counterexample) will fail:

```
private static void Transform(Rectangle rect)
{
  // Height becomes twice the width
  rect.SetSize(rect.Width, 2*rect.Width);
}
```

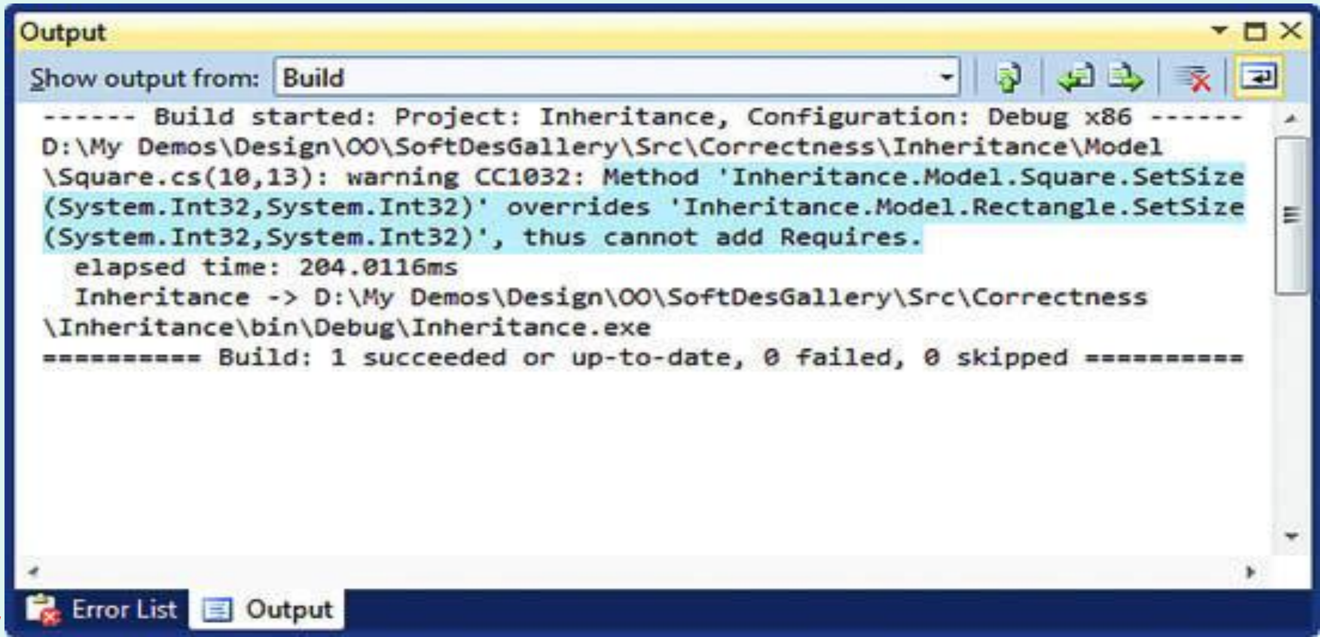
The method Transform was originally written to deal with instances of the Rectangle class, and it does its job quite well. Suppose that one day you extend the system and start passing instances of Square to the same (untouched) code, as shown here:

```
var square = new Square();
square.SetSize(20, 20);
Transform(square);
```

Depending on the relationship between Square and Rectangle, the Transform method may start failing without an apparent explanation.

The nice thing about .NET and the C# compiler is that if you use Code Contracts to express preconditions, you get a warning from the compiler if you're violating the Liskov principle:

(той е нарушен, защото Transform() не може да се използва в дъщерни класове , не е припокрит в тях, а ограничението само свива обсега – т.е. минава успешно.



```
------ Build started: Project: Inheritance, Configuration: Debug x86 ------
D:\My Demos\Design\OO\SoftDesGallery\Src\Correctness\Inheritance\Model
\Square.cs(10,13): warning CC1032: Method 'Inheritance.Model.Square.SetSize
(System.Int32,System.Int32)' overrides 'Inheritance.Model.Rectangle.SetSize
(System.Int32,System.Int32)', thus cannot add Requires.
  elapsed time: 204.0116ms
  Inheritance -> D:\My Demos\Design\OO\SoftDesGallery\Src\Correctness
\Inheritance\bin\Debug\Inheritance.exe
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```