

Design Patterns 1

Corné Hoskam

Modulaire Principes	1
Behavioural Patterns	2
Mediator Pattern	2
Template Method	3
Visitor Pattern	4
Strategy Pattern & Dependency Injection	5
Decorator Pattern	6
Composite Pattern	6
State Pattern	7
Creation Patterns	8
Singleton	8
Object Pool	8
Abstract Factory & Factory Method	9
Prototype	10
Builder	11
Bad Code Smells	12
Bloaters	12
Object-Orientation Abusers	12
Change preventers	12
Dispensables	13
Couplers	13
Test Driven Development	13
Red/Green refactoring	13
Triple A Testing:	14
TDD Anti-Patterns	14
Begrippen	15
Overerving in patronen	15
Virtual & Final keywords	15
Early/Late binding	16
Refactoring	16
Bijlage	17
TDD Anti-Patterns	17

Modulaire Principes

- **Decompositie** (met name in een top-down design)
A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex sub-problems, connected by a simple structure, and independent enough to allow further work to proceed separately on each item."
Slecht:
 Een klasse / methode die alles initialiseert.
 Hoge afhankelijkheid van die ene klasse / methode.
- **Compositie**
"A method satisfies Modular Composability if it favours the products of software elements which may then be freely combined with each other or produce new systems, possibly in an environment quite different from the one in which they were initially developed."
- **Begrijpelijkheid**
"A method favours Modular Understandability if it helps produce software which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others"
- **Continuïteit**
"A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in the problem specification will trigger a change of just one module, or a small number of modules."
- **Bescherming**
"A method satisfied Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighbouring modules."
Voorbeeld: Nagaan of de ingangscondities en de uitgangscondities kloppen.

Behavioural Patterns

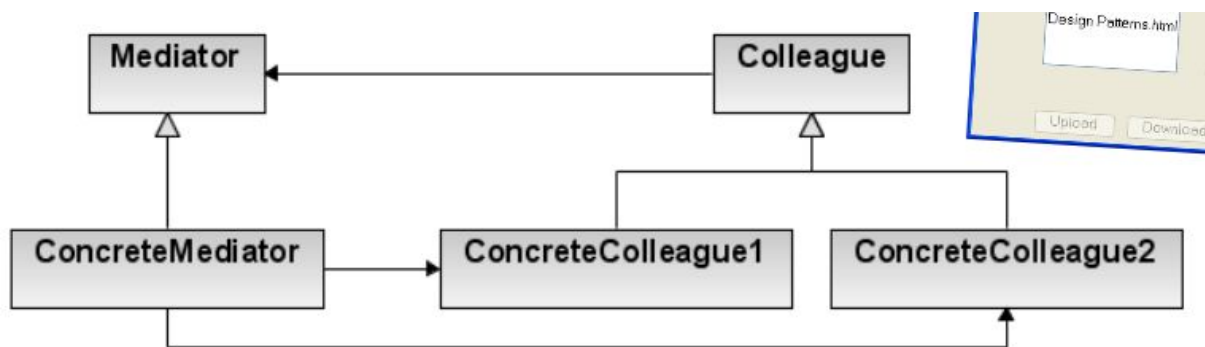
Mediator Pattern

Doel:

Het loskoppelen van verschillende onderdelen in je applicatie.
Expliciete verwijzingen naar andere objecten verwijderen.

Middel:

Een centraal systeem die de onderlinge communicatie regisseert (Denk aan een controle toren bij een vliegveld)



Vergeleken met een observer:

- Observer is slechts éénrichtingsverkeer
- Observer is live luisteren in plaats van bemiddelen.

Vergeleken met een façade:

- Façade abstraheert een module.
- Façade is éénrichtingsverkeer
- Façade biedt geen nieuwe functionaliteit

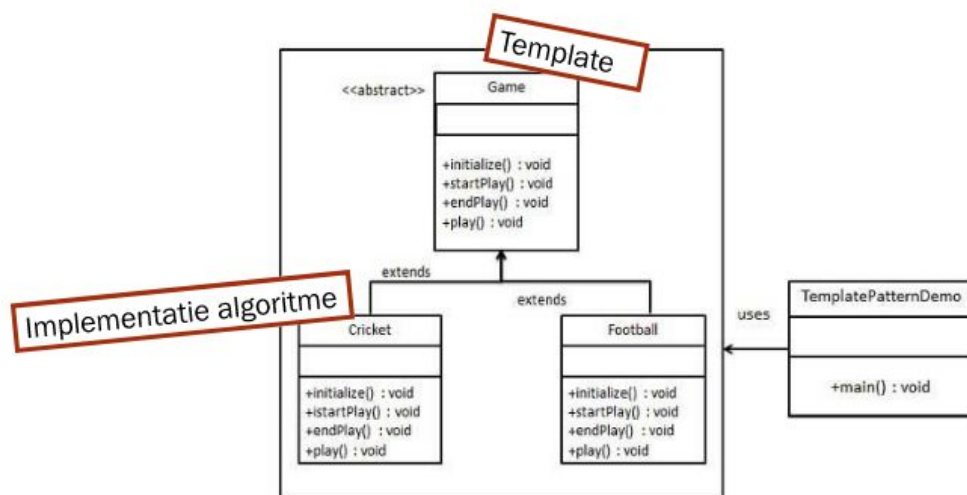
Template Method

Doel:

Een raamwerk creëren waarbij bepaalde algoritmes door subclasses worden uitgevoerd.

Voordeel:

Subklassen definiëren specifiek werk waar het raamwerk overall gelijk is. Code wordt overzichtelijker.



Implementatie:

```

public abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

    //template method
    public void play(){

        //initialize the game
        initialize();

        //start game
        startPlay();

        //end game
        endPlay();
    }
}
  
```

```

public class Football : Game {
    override void endPlay() {
        Console.WriteLine("Football Game Finished!");
    }

    override void initialize() {
        Console.WriteLine("Football Game Initialized! Start playing.");
    }

    override void startPlay() {
        Console.WriteLine("Football Game Started. Enjoy the game!");
    }
}
  
```

Visitor Pattern

Doel:

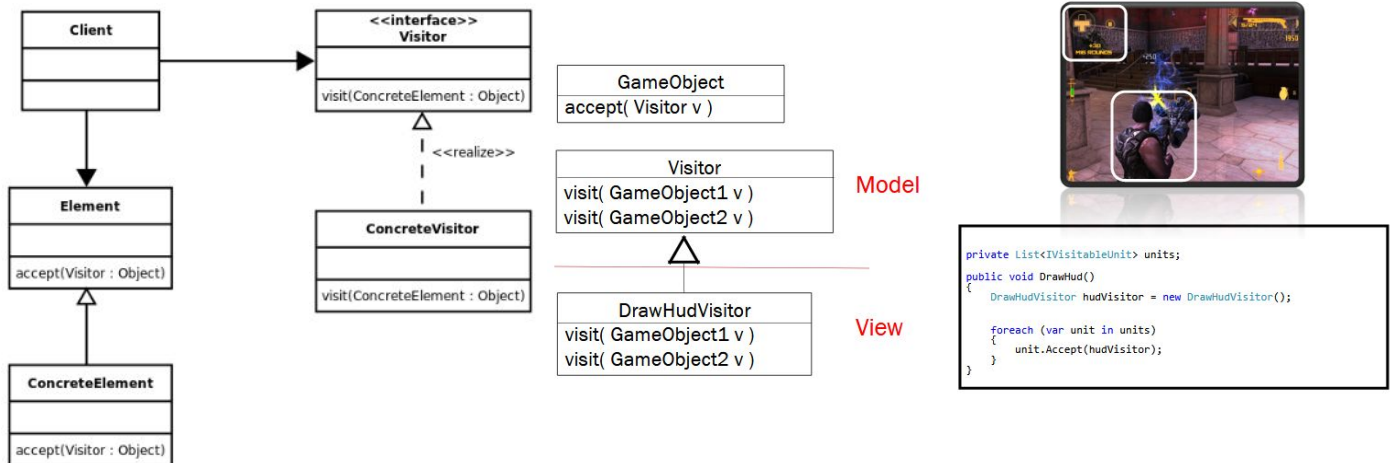
Nieuwe functionaliteit aan klassen toevoegen zonder deze klassen zelf aan te passen.

Voordeel:

- Uitbreidbare code voor toekomstige nieuwe functionaliteit.
- Scheiding van verschillende functionaliteit/modules.

Uitleg:

- Visitor Pattern brengt functionaliteit bij elkaar (Dus niet in models)
- Models kennen erg weinig van de rest
- Toekomstbestendig
- Wel een aantal extra klassen (wat onoverzichtelijk kan worden)



Implementatie:

```
public class Player : IVisitableUnit
{
    public void Accept(IUnitVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

```
public class Enemy : IVisitableUnit
{
    public void Accept(IUnitVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

```
public class DrawHudVisitor : IUnitVisitor
{
    public void Visit(Player player) { /* Draw player's health */ }
    public void Visit(Enemy enemy) { /* Draw enemy in map */ }
}
```

```
public class DrawUnitVisitor : IUnitVisitor
{
    public void Visit(Player player) { /* Draw player's body */ }
    public void Visit(Enemy enemy) { /* Draw enemy's body */ }
}
```

Strategy Pattern & Dependency Injection

Uitleg:

Overerving:

- Statisch. Je legt overerving vast in de klassendefinitie.
- Niet gemakkelijk @ runtime te verwisselen.

Associate/Delegatie:

- Dynamisch. Laat een ander object het werk doen.
- Gemakkelijker @ runtime te verwisselen.

Voordelen:

- Het delegeren van implementatie
- @ runtime wisselen van implementatie, o.a. aan de hand van user input.
- Geen switch cases maar gewoon het werk laten uitvoeren door een ander.

```
public abstract class Car {
    public abstract void Start();

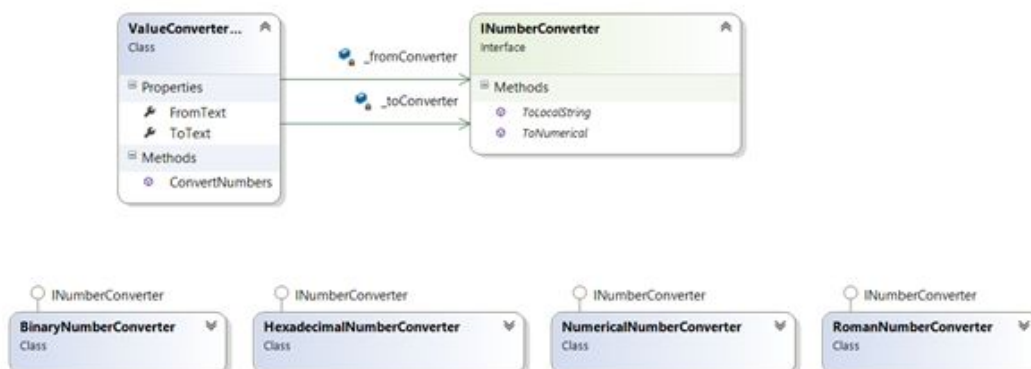
    public void Drive() {
        Start();
    }
}

public class Truck : Car {
    public override void Start() {
        Console.WriteLine("V8 Starting");
    }
}
```

```
public class Car {
    private Engine _engine;

    public void Drive() {
        _engine.Start();
    }
}

public class Engine {
    public void Start() {
        Console.WriteLine("V8 Starting");
    }
}
```



Dependency Injection:

Opties:

- Constructor injection
- Setter injection
- Interface injection

Voordelen:

Makkelijk om een mock te injecten en te testen!

Decorator Pattern

Doel:

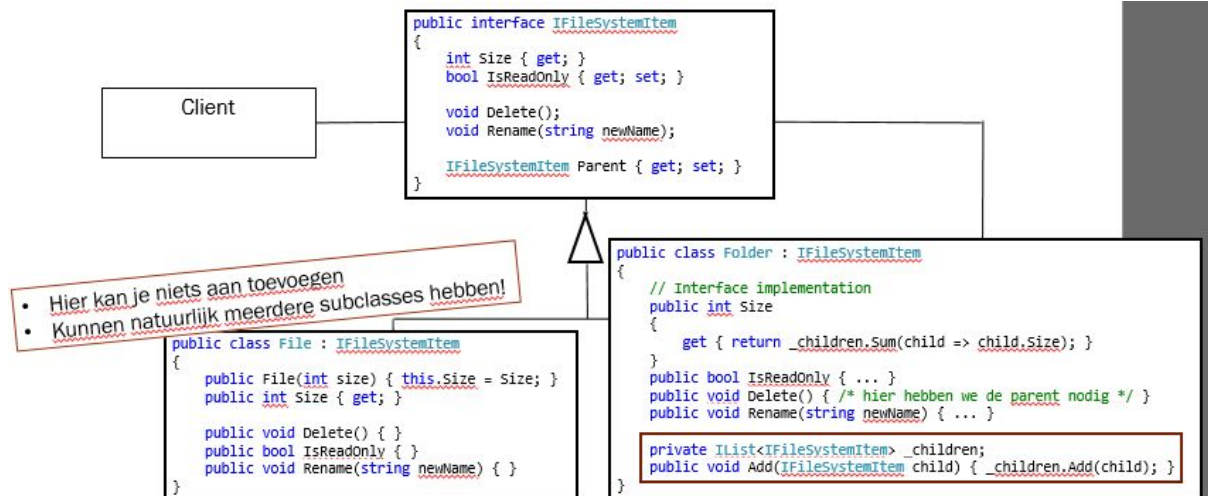
Dynamisch functionaliteit toevoegen aan een object i.p.v. voor iedere combinatie een subklasse maken. (Denk aan de Ninja app in C# WPF)



Composite Pattern

Uitleg:

Bestaat uit een boomstructuur waarin de hiërarchie benaderd kan worden als één object.



State Pattern

Uitleg:

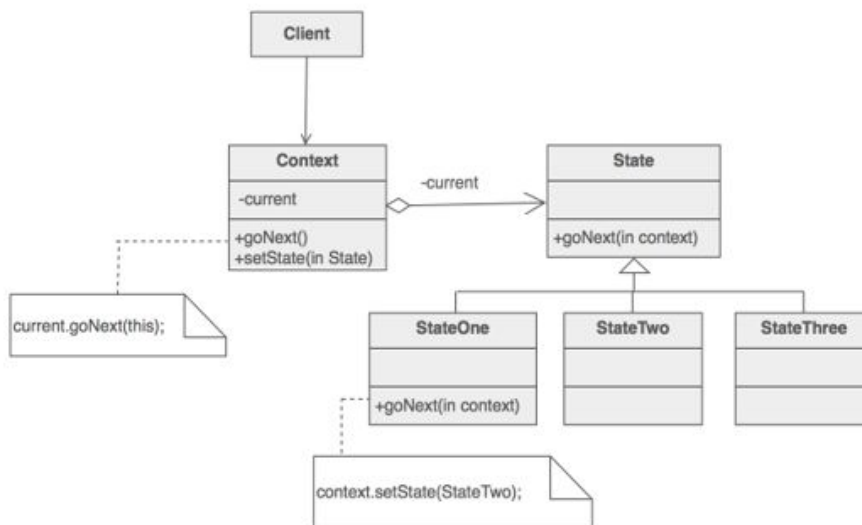
Gedrag kan veranderen op basis van de status.

Status kan zelf bepalen wanneer het object naar een nieuwe status gaat.

Voorbeeld:

Een Heart rate monitor:

- Monitoring
- Warning
- Error
- Muted



Creation Patterns

Singleton

Uitleg:

Het gebruik van één instantie van een object die overal benaderbaar is.

```
public class MySingleton
{
    private MySingleton();
    private static MySingleton _instance;

    public static MySingleton Instance
    {
        get
        {
            if ( _instance == null )
                _instance = new MySingleton();

            return _instance;
        }
    }
}
```

- Overall benaderbaar:
`MySingleton.Instance.Sum(1, 2);`
- Goed voor configuratie
- Goed voor (Abstract) factories

"The Singleton design pattern is one of the most inappropriately used patterns."

"When is Singleton unnecessary?"

Short answer: most of the time.

Long answer: ...The real problem with Singletons is that they give you such a good excuse not to think carefully about the appropriate visibility of an object..."

Object Pool

Doel:

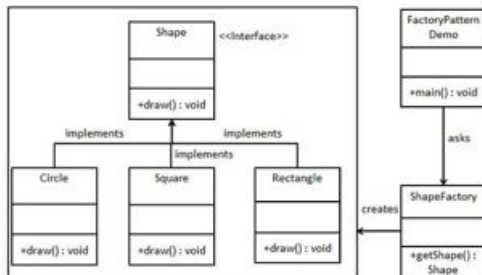
Herbruikbare objecten in reserve houden om zo geheugengebruik te verminderen, denk aan game particles die overbodig vaak worden gecreëerd en vernietigd.

Voordeel:

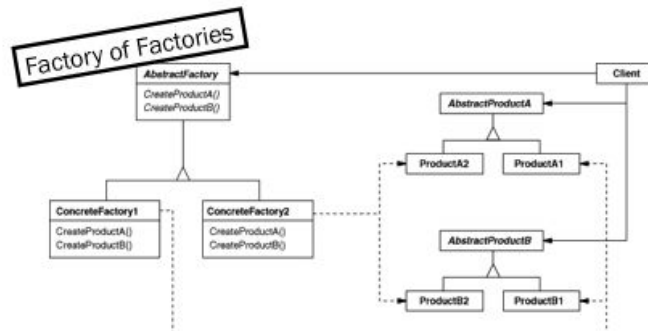
- Minder initialisatie van objecten
- Minder geheugengebruik

Abstract Factory & Factory Method

Factory method:



Abstract Factory



High Binding Factory Method:

```

public static class CarFactory
{
    public static Car CreateCar(string type)
    {
        if (type == "Jeep")
            return new Jeep();
        else if (type == "StationWagon")
            return new StationWagon();
        else if (type == "Convertible")
            return new Convertible();
        else
            return null;
    }
}
  
```

- High binding / High Coupling
- Bij nieuwe subclass elke keer aanpassen
- Foutgevoelig

Low Binding Factory Method:

```

public class CarFactory
{
    private Dictionary<string, Type> _types;

    public void AddCarType(string name, Type type)
    {
        _types[name] = type;
    }

    public Car CreateCar(string type)
    {
        Type t = _types[type];
        Car c = (Car)Activator.CreateInstance(t);
        return c;
    }
}
  
```

- Low binding / low coupling
- Factory kent alleen de base type
- Wie gaat de types vullen? Zelfregistratie?

Prototype

Doel:

Objecten clonen op basis van een bestaand object.

Voordelen:

- Initializatie/Configuratie is niet meer aan de client.
- Kies iets dat bevat en maak er nog één.

Voorbeeld:

Kopieer een noot of maat, en clone deze om te kunnen plakken op andere plekken in een muziekapplicatie.

```
public class Note
{
    public int Length { get; set; }
    public char Letter { get; set; }
    public int Octave { get; set; }

    public Note Clone()
    {
        return new Note(){ Length = this.Length, Letter = this.Letter, Octave = this.Octave };
    }
}
```

Onderstaand voorbeeld:

- Iedere subklasse kan nu ook een kopie van zijn eigen type teruggeven.
- Er moet een initialisatie- of registratiemethode zijn om de dictionary te vullen.

```
public class CarFactory
{
    private Dictionary<string, Car> _cars;

    public void RegisterCar(string name, Car prototype)
    {
        _cars[name] = prototype;
    }

    public Car CreateCar(string type)
    {
        Car prototype = _cars[type];
        return prototype.Clone();
    }
}
```


Bad Code Smells

Bloaters

Beschrijving:

Grote onhandelbare stukken code.

Te herkennen aan:

- Long methods.
- Large classes.
- Primitive obsession.
- Long parameter lists.
- Data clumps.

```
public abstract class AbstractCollection : Collection<string> {
    public void AddAll(AbstractCollection collection) {
        if (collection is Set) {
            Set set = (Set)collection;
            for (int i = 0; i < set.Size; i++) {
                if (!Contains(set.ElementAt(i)))
                    Add(set.ElementAt(i));
            }
        }
        else if (collection is List) {
            List list = (List)collection;
            for (int i = 0; i < list.Size; i++) {
                if (!Contains(list.Get(i)))
                    Add(list.Get(i));
            }
        }
        else if (collection is Map) {
            Map map = (Map)collection;
            for (int i = 0; i < map.Size; i++) {
                Add(map.Keys[i], map.Values[i]);
            }
        }
    }
}
```

- Switch statement
- Duplicate code
- Duplicate code
- Alternative classes with different interfaces
- Inappropriate intimacy
- Long method

Object-Orientation Abusers

Beschrijving:

Niet netjes OO-programmeren

Te herkennen aan:

- Switch statements
- Temporary fields
- Refused bequest
- Alternative classes with different interfaces

Change preventers

Beschrijving:

Onveranderbare code

Te herkennen aan:

- Divergent change
- Shotgun surgery
- Parallel inheritance

Dispensables

Beschrijving:

Dingen die weg kunnen (wel testen!)

Te herkennen aan:

- Comments
- Duplicate code
- Lazy class
- Data class
- Dead code
- Speculative Generality

Couplers

Beschrijving:

Te grote afhankelijkheid in code

Te herkennen aan:

- Feature envy
- Inappropriate intimacy
- Message chains
- Middle man
- Incomplete library class

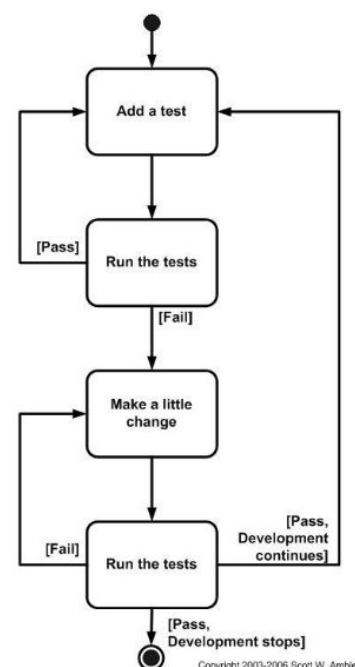
Test Driven Development

Goede tests:

- Zijn requirements
- Worden vaak uitgevoerd (20x per dag)
- Testen kleine stappen

Red/Green refactoring

1. Create a unit test that fails
2. Write production code that makes it pass
3. Clean up the mess you just made



Triple A Testing:

- Arrange
- Act
- Assert

```
[TestMethod]
public void Add AddingValidItem ShouldAddOperationToLink()
{
    // Arrange
    UndoableNawArray testSubject = new UndoableNawArray(10);
    Naw expectedNaw = new Naw(){ Naam = "testnaam" };
    Operation expectedOperation = Operation.ADD;

    // Act
    testSubject.Add(expectedNaw);
    Naw actualNaw = testSubject.First.Naw;
    Operation actualOperation = testSubject.First.Operation;

    // Assert
    Assert.AreEqual(expectedOperation, actualOperation, "Foutmelding hier...");
    Assert.AreEqual(expectedNaw, actualNaw, "Foutmelding hier...");
}
```

TDD Anti-Patterns

- The Liar
 - Wordt niet echt getest
- Excessive Setup
 - Enorme hoeveelheid code om een klein stukje te testen
- The Mockery
 - Alles wordt gemockt, wordt er nog wel echte code getest?
- Generous Leftovers
 - Unittests zijn afhankelijk van wat vorige tests hebben klaargezet.
- The Local Hero
 - Tests zijn afhankelijk van omgeving, werkt bij de ene wel en bij de andere niet.
- The Sequencer
 - Er wordt uitgegaan van een volgorde wanneer dat niet gegarandeerd is.
- Success against all odds
 - Pass first, lege methode zou dus al een slagende test geven.
- The Free Ride
 - Nieuw scenario, maar bij een bestaande test “want daar kon hij makkelijk bij”.

(Meer in bijlage)

Begrippen

Overerving in patronen

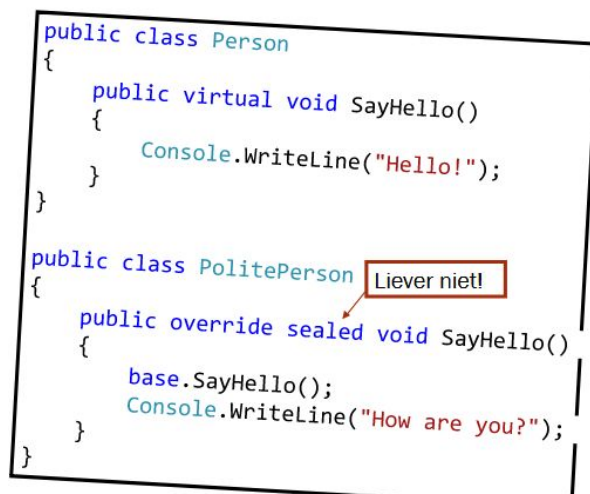
Uitleg:

- Overerving bepaald statische relaties, en is niet dynamisch tijdens runtime te beïnvloeden.
- Alleen afgeleide klasse wanneer elke instantie een speciaal soort van de superklasse is.
- Alle subklassen hebben dezelfde superklasse.
- Gebruik overerving met beleid: In geval van twijfel, niet doen!
- Geen overerving mogelijk bij referentie naar een object die @ runtime wijzigt, of wanneer er maar een deel van een datastructuur gebruikt moet worden..

Virtual & Final keywords

Uitleg:

- Final (Java) / Sealed (C#): mag niet overriden worden.
- Virtual (C#, geen java keyword): mag overriden worden.
- In C# krijg je geen compiler errors zonder keyword, altijd virtual en override gebruiken!



```

public class Person
{
    public virtual void SayHello()
    {
        Console.WriteLine("Hello!");
    }
}

public class PolitePerson
{
    public override sealed void SayHello()
    {
        base.SayHello();
        Console.WriteLine("How are you?");
    }
}
  
```


Early/Late binding

Uitleg:

Early Binding bestaat uit meerdere methodes met dezelfde naam. Op basis van de argumenten wordt bepaald welke methode uit wordt gevoerd.

Wanneer het tijdens compile-time wordt bepaald is het early.

Bij Late Binding wordt aan de hand van het object bepaalt welke code uit wordt gevoerd.

Wanneer het tijdens runtime wordt bepaald is het late.

Refactoring

Doel:

Het zorgen voor clean code, beschreven als:

- Duidelijk voor andere programmeurs.
- Bevat geen duplicate code.
- Bevat een minimaal aantal classes en andere 'bewegende onderdelen'.
- Slaagt bij alle tests.
- Is makkelijker en goedkoper te onderhouden.

Waarom:

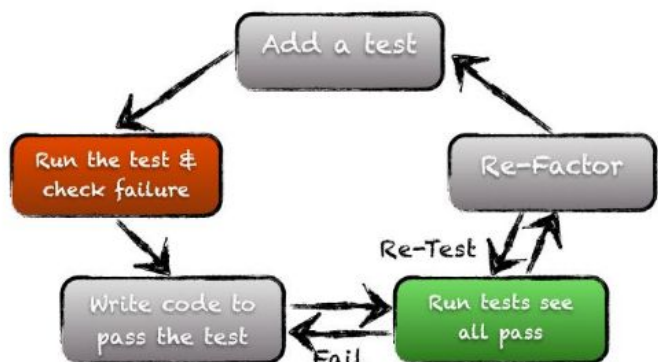
- Creëert een verbeterd design.
- Wordt gemakkelijker te begrijpen.
- Je vind bugs door refactoring.
- Sneller programmeren.

Wanneer:

- Bij het toevoegen van functionaliteit.
- Bij bugfixing.
- Bij code reviewing.

Niet refactoring wanneer:

- De functionaliteit verandert.
- De performance niet verbeterd.
- Je iets vanuit scratch gaat herschrijven.



Build, test, refactor, test -> build, test....

Bijlage

TDD Anti-Patterns

The Liar

An entire unit test that passes all of the test cases it has and appears valid, but upon closer inspection it is discovered that it doesn't really test the intended target at all.

Excessive Setup

A test that requires a lot of work setting up in order to even begin testing. Sometimes several hundred lines of code is used to setup the environment for one test, with several objects involved, which can make it difficult to really ascertain what is tested due to the noise of all of the setup going on.

The Giant

A unit test that, although it is validly testing the object under test, can span thousands of lines and contain many many test cases. This can be an indicator that the system under tests is a [God Object](#)

The Mockery

Sometimes mocking can be good, and handy. But sometimes developers can lose themselves and in their effort to mock out what isn't being tested. In this case, a unit test contains so many mocks, stubs, and/or fakes that the system under test isn't even being tested at all, instead data returned from mocks is what is being tested.

The Inspector

A unit test that violates encapsulation in an effort to achieve 100% code coverage, but knows so much about what is going on in the object that any attempt to refactor will break the existing test and require any change to be reflected in the unit test.

Generous Leftovers ^[4]

An instance where one unit test creates data that is persisted somewhere, and another test reuses the data for its own devious purposes. If the generator is ran afterward, or not at all, the test using that data will outright fail.

The Local Hero ^[1]

A test case that is dependent on something specific to the development environment it was written on in order to run. The result is the test passes on development boxes, but fails when someone attempts to run it elsewhere.

The Nitpicker ^[1]

A unit test which compares a complete output when it's really only interested in small parts of it, so the test has to continually be kept in line with otherwise unimportant details. Endemic in web application testing.

The Secret Catcher ^[1]

A test that at first glance appears to be doing no testing due to the absence of assertions, but as they say, the devils in the details. The test is really relying on an exception to be thrown when a mishap occurs, and is expecting the testing framework to capture the exception and report it to the user as a failure.

The Dodger ^[1]

A unit test which has lots of tests for minor (and presumably easy to test) side effects, but never tests the core desired behavior. Sometimes you may find this in database access related tests, where a method is called, then the test selects from the database and runs assertions against the result.

The Loudmouth ^[1]

A unit test (or test suite) that clutters up the console with diagnostic messages, logging messages, and other miscellaneous chatter, even when tests are passing. Sometimes during test creation there was a desire to manually see output, but even though its no longer needed, it was left behind.

The Greedy Catcher ^[1]

A unit test which catches exceptions and swallows the stack trace, sometimes replacing it with a less informative failure message, but sometimes even just logging (c.f. Loudmouth) and letting the test pass.

The Sequencer ^[1]

A unit test that depends on items in an unordered list appearing in the same order during assertions.

Hidden Dependency

A close cousin of The Local Hero, a unit test that requires some existing data to have been populated somewhere before the test runs. If that data wasnt populated, the test will fail and leave little indication to the developer what it wanted, or why forcing them to dig through acres of code to find out where the data it was using was supposed to come from.

The Enumerator ^[2]

A unit test with each test case method name is only an enumeration, i.e. test1, test2, test3. As a result, the intention of the test case is unclear, and the only way to be sure is to read the test case code and pray for clarity.

The Stranger

A test case that doesnt even belong in the unit test it is part of. its really testing a separate object, most likely an object that is used by the object under test, but the test case has gone and tested that object directly without relying on the output from the object under test making use of that object for its own behavior. Also known as **TheDistantRelative** ^[3].

The Operating System Evangelist ^[3]

A unit test that relies on a specific operation system environment to be in place in order to work. A good example would be a test case that uses the newline sequence for windows in an assertion, only to break when ran on Linux.

Success Against All Odds ^[2]

A test that was written pass first rather than fail first. As an unfortunate side effect, the test case happens to always pass even though the test should fail.

The Free Ride

Rather than write a new test case method to test another feature or functionality, a new assertion rides along in an existing test case.

The One

A combination of several patterns, particularly **TheFreeRide** and **TheGiant**, a unit test that contains only one test method which tests the entire set of functionality an object has. A common indicator is that the test method is often the same as the unit test name, and contains multiple lines of setup and assertions.

The Peeping Tom

A test that, due to shared resources, can see the result data of another test, and may cause the test to fail even though the system under test is perfectly valid. This has been seen commonly in fitness, where the use of static member variables to hold collections aren't properly cleaned after test execution, often popping up unexpectedly in other test runs. Also known as **TheUninvitedGuests**

The Slow Poke

A unit test that runs incredibly slow. When developers kick it off, they have time to go to the bathroom, grab a smoke, or worse, kick the test off before they go home at the end of the day.

Well, that wraps it up for now. I'd like it if people could vote on the top ten anti-patterns listed here so I can go about writing up more detailed descriptions, examples, symptoms, and (hopefully) refactorings to move away from the anti-pattern listed.