Professional Software Engineering

Andrea Carrara and Patrick Berggold

Hritik Singh and Mohab Hassaan – Tutors

Chair of Computational Modeling and Simulation

andrea.carrara@tum.de; patrick.berggold@tum.de

Schedule Week 3

- Recap OOP
- Interfaces
- Managing Projects
  - GIT
  - .net core projects
  - NUnit

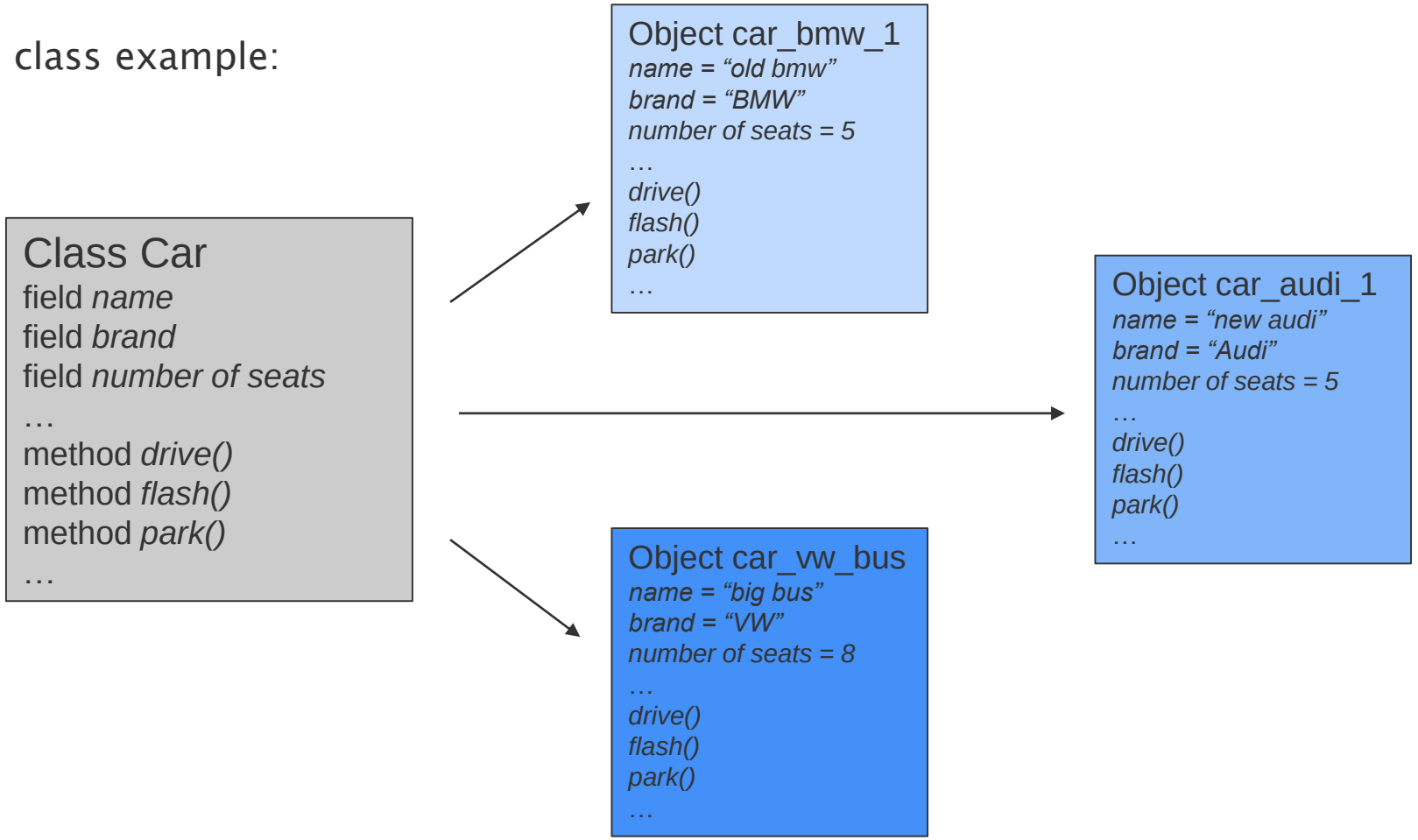
andrea.carrara@tum.de; patrick.berggold@tum.de

RECAP OOP

andrea.carrara@tum.de; patrick.berggold@tum.de

Class example

» A simple class example:



andrea.carrara@tum.de; patrick.berggold@tum.de

Class Declaration and Object Instantiation in C#

```
// class declaration
class Vector2D {
    // fields = member variables
    public double x;
    public double y;

    // methods = member functions
    public double Norm() {
        double nrm = Math.Sqrt(x * x + y * y);
        return nrm;
    }
}
```

```
// object instantiation
Vector2D vec1 = new Vector2D();
vec1.x = 5;
vec2.y = 10;
Console.Write(vec1.Norm());
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Constructor

- Special method(s) that is always called automatically when the object is created
- Method name = class name
- If no constructer is defined, the compiler automatically creates the standard constructor

```
class Circle {
   private double r;
   private double x;
   private double y;

   //hide the standard constructor
   private Circle() {
   }

   public Circle(double radius) {
       this.r = radius;
   }
}
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Virtual Methods – override

- A base method marked as virtual can be overridden by derived classes
- Keyword: override
- Used to provide a specialized implementation
- If casted to base again, will invoke the override the base method
- Not using override will issue a warning (not an error though)…

```
public class Figure {
    public virtual void Output() {
        Console.Write("Figure object");
    }
}

public class Circle : Figure {
    public override void Output() {
        Console.Write("Circle object");
    }
}

public class Rectangle : Figure {
    public override void Output() {
        Console.Write("Rectangle object");
    }
}
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Abstract classes

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

public class Circle : Figure {
    public override double Area() {
        return Math.Pi * Radius * Radius;
}
```

- Basically “concept” classes
- Can be used to indicate missing components or implementations

- Abstract classes can’t be instantiated (no objects!!!)
- Abstract methods can only be contained in abstract classes
- All abstract methods must be implemented by the child class

andrea.carrara@tum.de; patrick.berggold@tum.de

INTERFACES

andrea.carrara@tum.de; patrick.berggold@tum.de

What are Interfaces?

- Another way to achieve **abstraction**
- Interfaces is a **completely abstract** class that can only contain abstract methods and properties,
- Interface methods do not have a body – the body needs to be provided by the "implement" class
- By default, members of an interface are **abstract** and **public**!
- Like abstract classes, interfaces **cannot** be used to create objects – they cannot contain a constructor

andrea.carrara@tum.de; patrick.berggold@tum.de

What are Interfaces?

- Interfaces can be seen as a **contract** between itself and any class that implements it
- The “contract” assures the class implementing the interface will implement the interface’s properties and methods.
- Interface contains a public set of members:
  - **Properties**
  - **Methods**
  - **Events**
  - **Indexers**

andrea.carrara@tum.de; patrick.berggold@tum.de

Example for an Interface

```
interface IPolygon {
  // method without body
  void calculateArea();
}
```

- Start with an I (convention)
- All public!
- Properties
- No fields!
- Methods

```
interface IPolygon{
    // method without body
    void calculateArea(int l, int b);
}
class Rectangle : IPolygon{
    // implementation of methods inside interface
    public void calculateArea(int l, int b){
        int area = l * b;
        Console.WriteLine("Area of Rectangle: " + area);
    }
}
class Program{
    static void Main(string[] args){
        Rectangle r1 = new Rectangle();
        r1.calculateArea(100, 200);
    }
}
```

Output

```
Area of Rectangle: 20000
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Why Do We Want Interfaces?

- Make the code **maintainable, extensible,** and easily **testable**.
- **Achieve security** – hide certain details and only show the important details of an object (interface)
- Interfaces provide **loose coupling** (having no or least effect on other parts of code when we change one part of a code).
- C# does not support "multiple inheritances" (a class can only inherit from one base class)
- Achieve **multiple inheritances** with multiple interfaces

andrea.carrara@tum.de; patrick.berggold@tum.de

Example Multiple Inheritance

1

```
interface IPolygon{
    // method without body
    void calculateArea(int a, int b);
}
interface IColor{
    void getColor();
}
// implements two interface
class Rectangle : IPolygon, IColor{
    // implementation of IPolygon interface
    public void calculateArea(int a, int b){
        int area = a * b;
        Console.WriteLine("Area of Rectangle: " + area);
    }
    // implementation of IColor interface
    public void getColor(){
        Console.WriteLine("Red Rectangle");
    }
}
```

2

```
class Program{
    static void Main(string[] args)
    {
        Rectangle r1 = new Rectangle();
        r1.calculateArea(100, 200);
        r1.getColor();
    }
}
```

Output

```
Area of Rectangle: 20000
Red Rectangle
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Practical Example of Inheritance

1

```
using System;
namespace CsharpInterface{
    interface IPolygon{
        // method without body
        void calculateArea();
    }

    // implements interface
    class Rectangle : IPolygon{
        // implementation of IPolygon interface
        public void calculateArea(){
            int l = 30;
            int b = 90;
            int area = l * b;
            Console.WriteLine("Area of Rectangle: " + area);
        }
    }
```

2

```
class Square : IPolygon{
        // implementation of IPolygon interface
        public void calculateArea(){
            int l = 30;
            int area = l * l;
            Console.WriteLine("Area of Square: " + area);
        }
    }
    class Program{
        static void Main(string[] args){
            Rectangle r1 = new Rectangle();
            r1.calculateArea();
            Square s1 = new Square();
            s1.calculateArea();
        }
    }
}
```

Output

```
Area of Rectangle: 2700
Area of Square: 900
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Abstract Class vs Interface

Abstract Class	Interface
It contains both **declaration** and **definition** part.	It contains only a **declaration** part.
Multiple inheritances is **not achieved** by abstract class.	Multiple inheritance is **achieved** by interface.
It **contains** constructor.	It does **not contain** constructor.
It can **contain static members**.	It does **not contain static members**.
Different types of access modifiers: public, private, protected etc.	Only public access modifiers.
Class can only use **one** abstract class	A class can use **multiple** interface.
It is used to **implement the core identity** of class.	It is used to implement **peripheral abilities** of class.
Abstract class can contain **methods, fields, constants**, etc.	Interface can only contains **methods, properties, indexers, events**.
It can be **fully**, **partially** or **not implemented**.	It should be fully implemented.

andrea.carrara@tum.de; patrick.berggold@tum.de

Summary

- virtual: indicates that a method may be overridden by an inheritor
- override: overrides the functionality of a virtual method in a base class, providing different functionality.
- abstract: abstract methods must be implemented by the child and don't contain a body. Abstract classes can only be inherited, never instantiated!
- interface: creates a "contract" for the derived class without any implementation. There can be many "contracts" for one single child class.

andrea.carrara@tum.de; patrick.berggold@tum.de

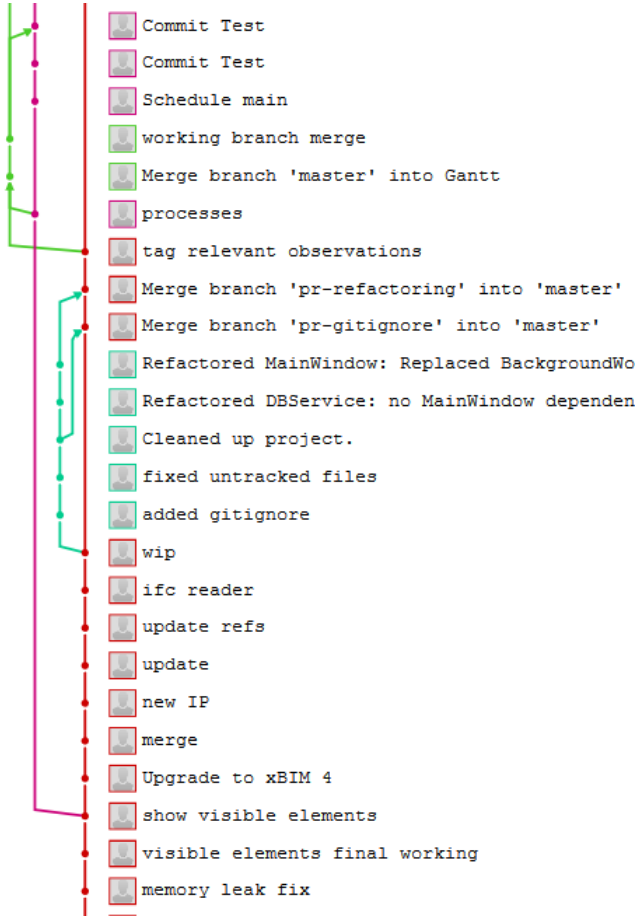
SOURCE CONTROL

A Git Introduction

andrea.carrara@tum.de; patrick.berggold@tum.de

What is Git?

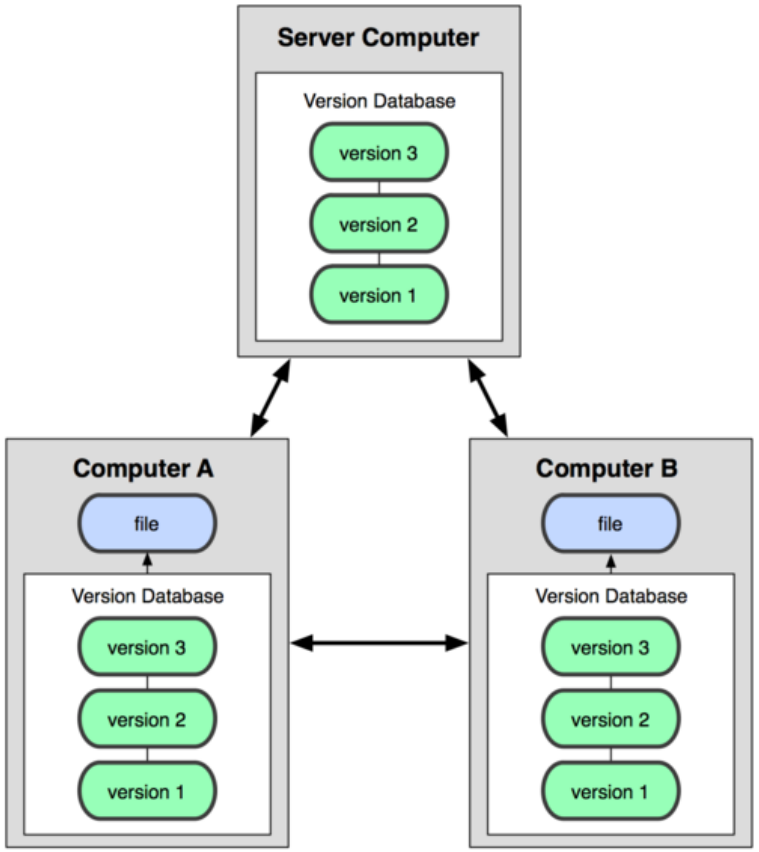
- Created by Linus Torvalds (the linux guy)
- Git is a *Version Control System (VCS)*
  - **aka source control**
- Keeps track of files
- Rollback -> previous states
- Comprehensible workflow
- Mostly used VCS available today



andrea.carrara@tum.de; patrick.berggold@tum.de

Distributed Version Control Systems

- Mirror the repository
- Server failure => new upload
- Client failure => new download
- Supports group collaboration
- Network independed
- *No permission required*
- *Private additions*
  - ***Local configurations***



andrea.carrara@tum.de; patrick.berggold@tum.de

Simple Workflow Example

- Start a new repository
  - git init
- Add and files you want to track
  - git add .
  - (or ignore those you don't want to)
  - .gitignore
- Commit them to the repository
  - git commit -m "…"

- Sync to server
  - Create a project
  - Add server to your local git
    - git remote add origin git@...
  - Send to changes to server
    - git push -u origin <branch>

andrea.carrara@tum.de; patrick.berggold@tum.de

Who are you?

```
Felix@TUBVCMS-FEI-1-W MINGW64 ~/Desktop/ProSD (master)
$ git commit -m "inital commit"

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository

fatal: empty ident name (for <(NULL)>) not allowed
```

… from and Existing Project

- Copy Repository from Server
  - git clone git@...
- Download from Server
  - git pull

- If the folder is not empty
  - *Basically you messed up earlier*
    - git init
    - git remote add origin git@...
    - git fetch
    - git checkout -t origin/master

andrea.carrara@tum.de; patrick.berggold@tum.de

Branching …

- List all branches
  - git branch -av
- Change branch
  - git checkout <branch>
- Create branch
  - git branch -t <new brach>
- TAKE CARE
  - Make current branch master
    - git rebase

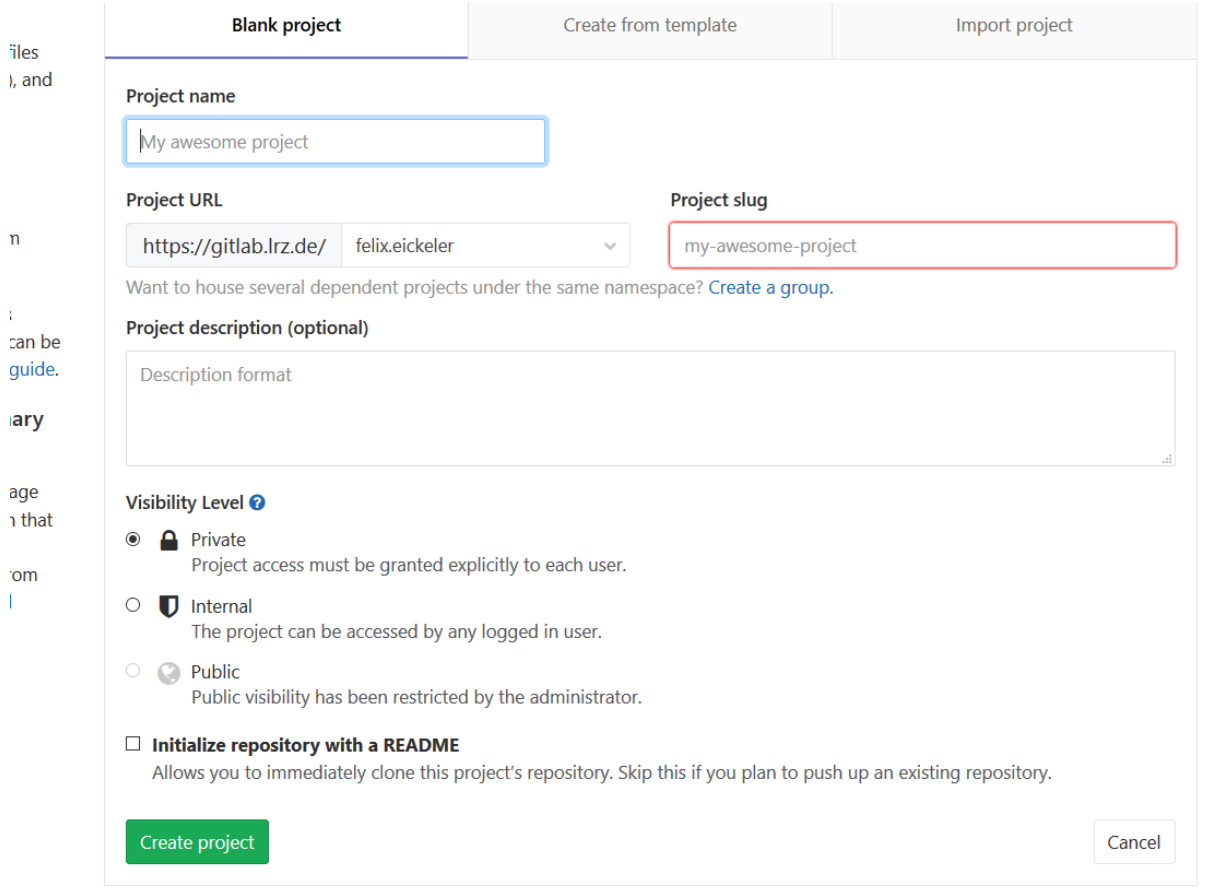
andrea.carrara@tum.de; patrick.berggold@tum.de

Publish & Undo

- Difference
  - git status
- Publish in repository
  - git push.
  - … in Branch
    - git push <remote> <branch>
- Discard local
  - git reset --hard
- Revert to old version
  - git checkout <file>
- Revert to old commit
  - git revert <commit>

andrea.carrara@tum.de; patrick.berggold@tum.de

SetUp your own @ gitlab.lrz.de !



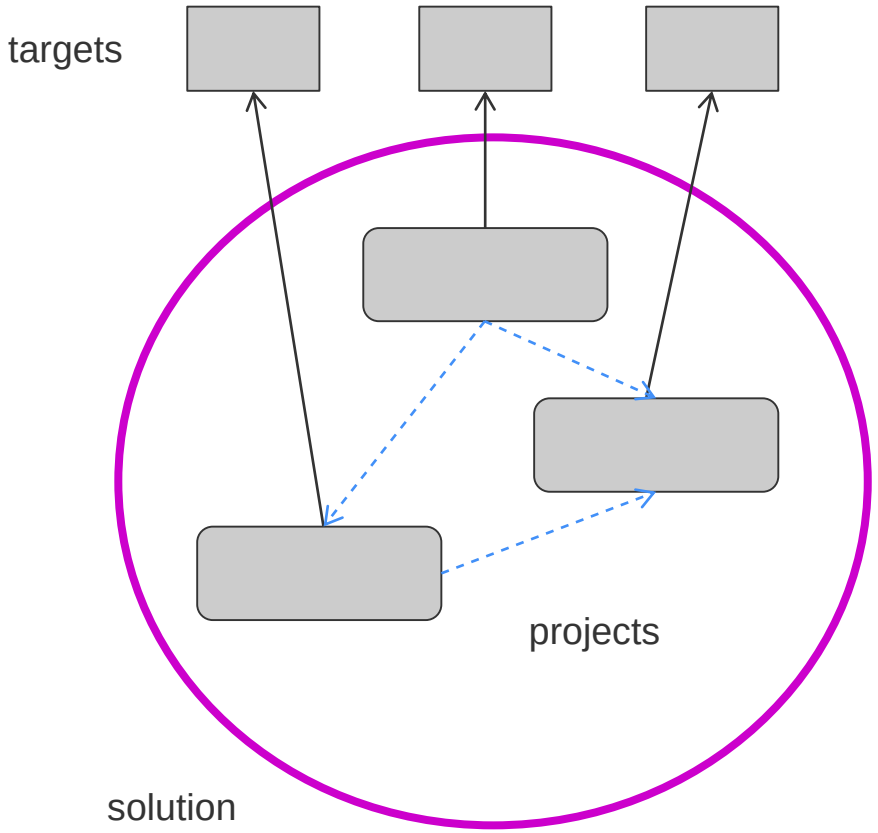
A .net Core 3.1 starter kit

PROJECTS AND SOLUTIONS

andrea.carrara@tum.de; patrick.berggold@tum.de

Terminology

- A **target** is software that is compiled to one specific output
- A **project** is a way of organizing you code, and normally corresponds to one target
- A **solution** is the sum of all projects that are part of your program and support the development cycle.
- Inside the solution, projects can access other progress via **references**
  - **no circular references allowed!**
- dotnet .core provides template to create those from scratch.
  - **you can create templates yourself and distribute them !**



andrea.carrara@tum.de; patrick.berggold@tum.de

Motivation, Implementation in C#, Test-Driven Development

TESTING

andrea.carrara@tum.de; patrick.berggold@tum.de

Motivation

- Maybe the most important topic in professional software development
- Not seen at universities
  - **incredible small product cycles**
- Could be handled in multiple lectures on its own!
  - **We will take a practical approach!**
  - **We will use Nunit**

andrea.carrara@tum.de; patrick.berggold@tum.de

Motivation

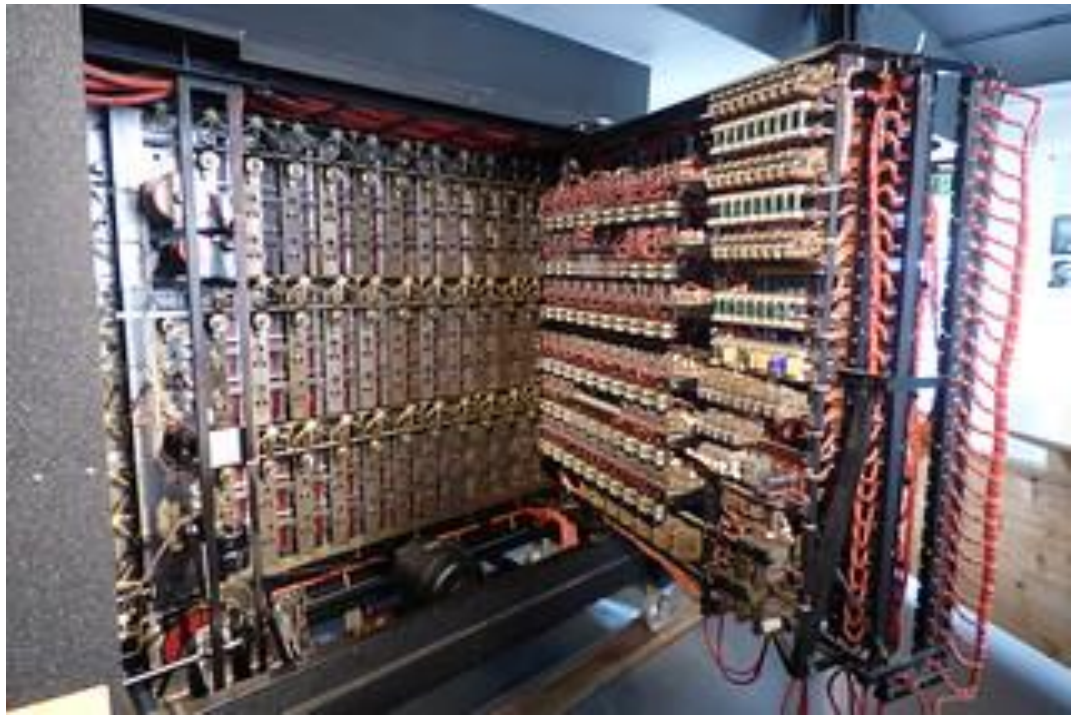
- Does it meet the design and development requirements?
  - **think first, develop later**

- Does it respond correctly to all kinds of inputs (**reliability**)
- Does it perform within an acceptable time (**performance**)
- Is it sufficiently usable (**ergonomically designed**)
- Can it be installed and run in its intended environments (**integration test**)

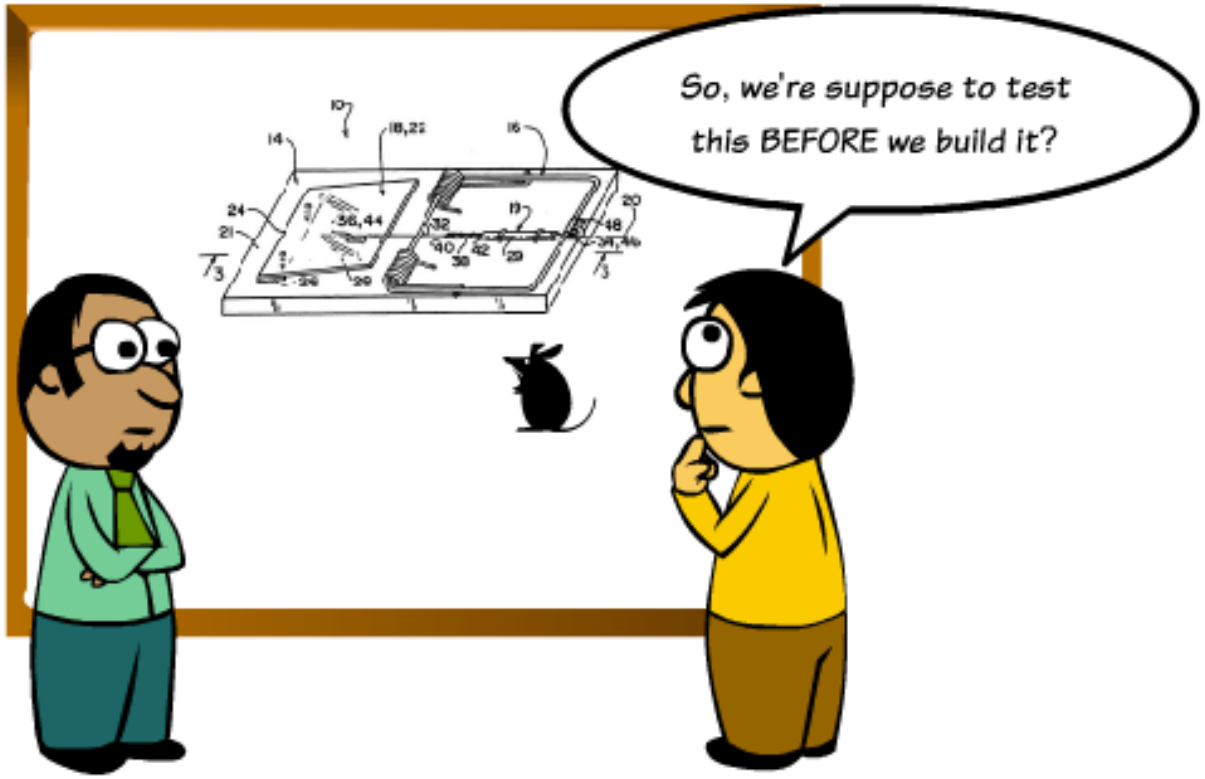
from: https://en.wikipedia.org/wiki/Software_testing

Motivation: Find Bugs Early!

- It is very expensive to fix bugs in released software
- Finding errors is tiresome
- TDD -> Inspection on creation
- a test is a specification how the software behave
- Unit tests are basically a written contract
- Writing tests from the start is easier than adding them later:
  - **Old code is not written with testability in mind**



andrea.carrara@tum.de; patrick.berggold@tum.de

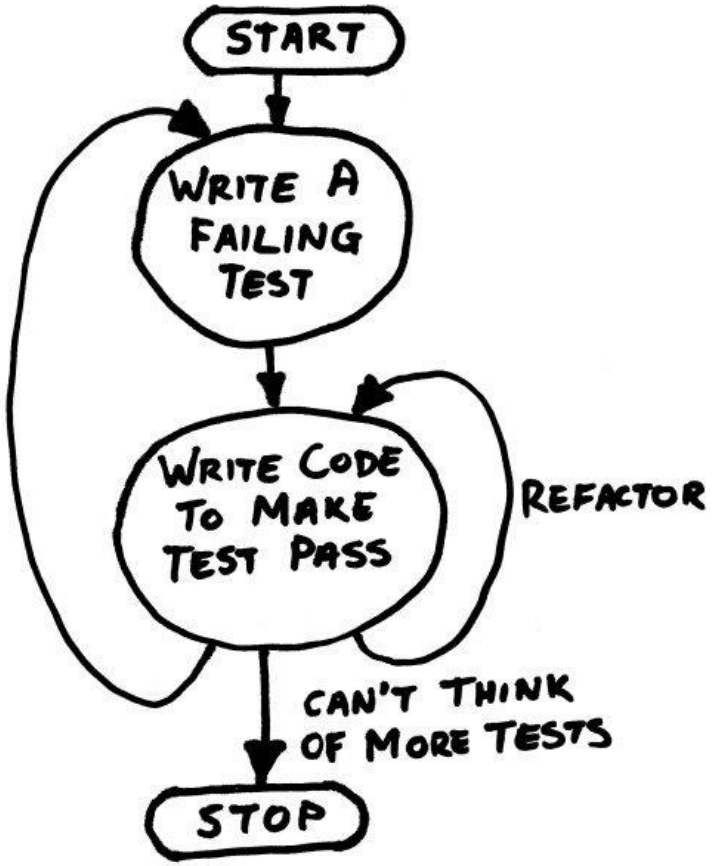


TEST DRIVEN DEVELOPMENT

andrea.carrara@tum.de; patrick.berggold@tum.de

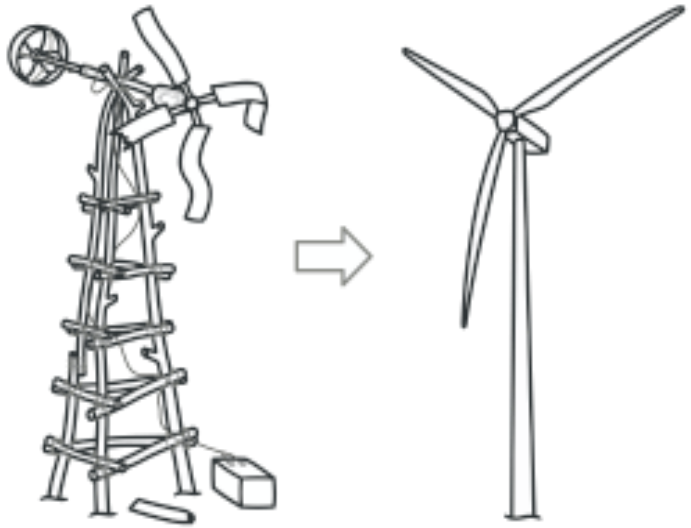
Test Driven Development

- Start with the easy tests
- After your code takes shape take on the bigger tasks
- Grouping of Tests
  - **One per class**
  - **One per UnitOfWork**
- Phases:
  - **Make It Work,**
  - **Make It Beautiful,**
  - **Make It Fast**



from: Ashutosh Nilkanth's TDD Article

Refactor With Test



- Refactoring is a controlled technique for improving the design of an existing code base.
- A series of small behaviour-preserving transformations, each of which "too small to be worth doing".
- Significant cumulative effect !

- With TDD:
  - **Improve the internal implementation without changing the external behavior**

from: Martin Fowler - Improving the Design of Existing Code
image@refactoring.guru_

Basic process

- Write your tests
- Create the carcasses of the classes & methods
  - **To make the compiler happy: NotImplementedException or empty return of the given type (recommended)**
- If a test is passing, but the implementation is not valid.
  - **You have not added the test you wanted!**
- Run the test often!
- **Strive for one logical assertion per test!**

andrea.carrara@tum.de; patrick.berggold@tum.de

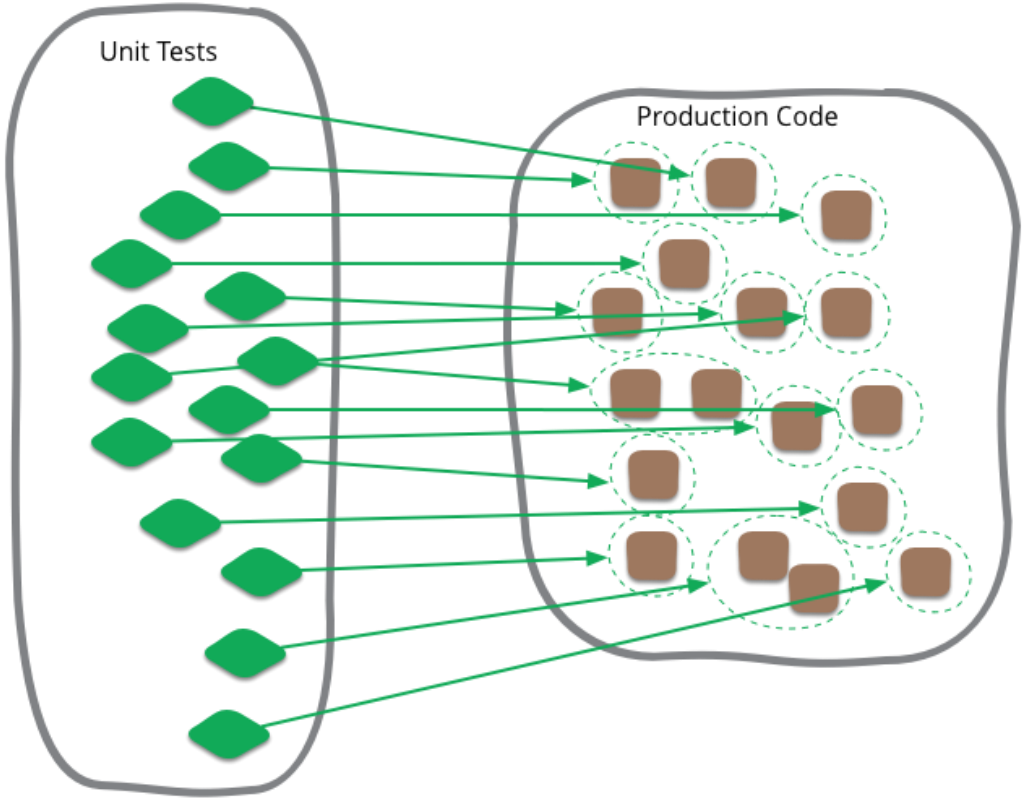
Focus On Unit Testing

- Verify the functionality of your code
- In OOP most of the time:
  - **public functions**
  - **constructors**
  - **Destructors**
- Increase of quality of your code!

andrea.carrara@tum.de; patrick.berggold@tum.de

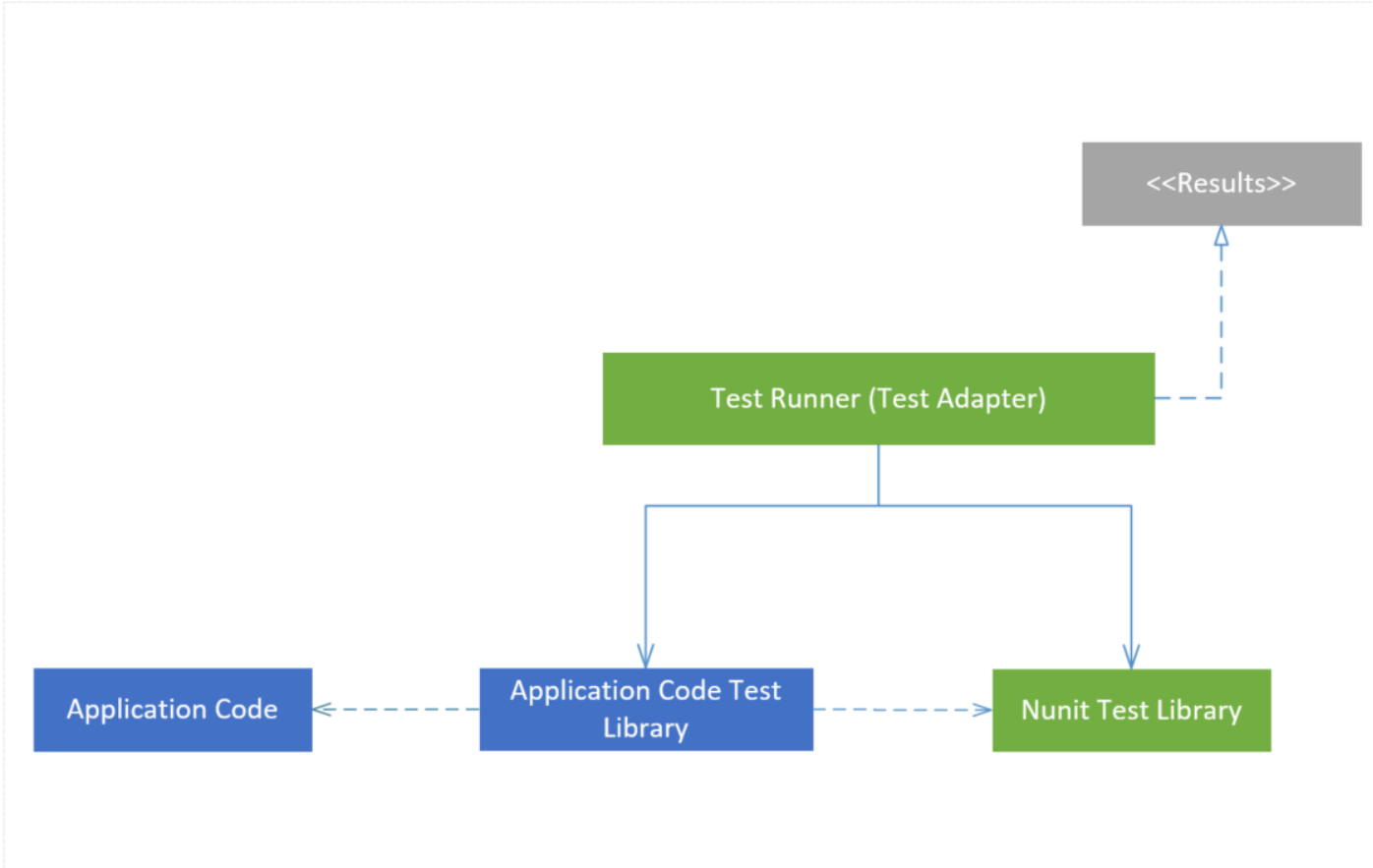
What Is A Unit Test?

- A tests can include:
  - static code analysis
  - data-flow analysis
  - metrics analysis
  - peer code reviews
  - **Written tests!**
- We will use the Nunit



from: https://en.wikipedia.org/wiki/Software_testing
image@martinfowler.com

Unit Test Framework



andrea.carrara@tum.de; patrick.berggold@tum.de

NUnit

- Opensource testing framework for .net
  - **template in .net core**
  - **runner Integrated in Jetbrains Rider**
- Test can also be run within visual studio
- Test can be categorized

```
using NUnit.Framework;

 [TestFixture]
public class ExampleTestOfNUnit {
    [Test]
    public void TestMultiplication() {
        Assert.AreEqual(4, 2 * 2, "Multiplication");

        //Equivalently, since version 2.4 NUnit
        //offers a new more intuitive assertion syntax
        //based on constraint objects
        Assert.That(2 * 2, Is.EqualTo(4),
                  "Multiplication constraint-based");
    }
}
```


andrea.carrara@tum.de; patrick.berggold@tum.de

DEMO:
TESTING;

andrea.carrara@tum.de; patrick.berggold@tum.de