Das Virtual Programming Lab (VPL) für Moodle

20.03.2019

Agenda

- Einführung und erste Schritte
- Grundlagen: Tests mit Testfällen (Wertepaare)
- Wir finden dich: Plagiaterkennung
- Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App
- Was geht noch?

Einführung und erste Schritte

Einführung und erste Schritte:
Motivation – viel zu tun

Das übliche Prozedere für Dozent_in in Programmierung ü.ä.:

1. Übungsaufgaben neu schreiben, anpassen und in der Übung stellen.
2. Studierende bearbeiten Aufgabe und geben ihre Lösung ab.
3. Dozent_in lädt Lösung herunter.
4. Dozent_in liest, kompiliert und testet Lösung.
5. Dozent_in vergleicht Lösung mit anderen Abgaben (Plagiat?).
6. Dozent_in bewertet Lösung.

Einführung und erste Schritte:
Motivation – VPL übernimmt

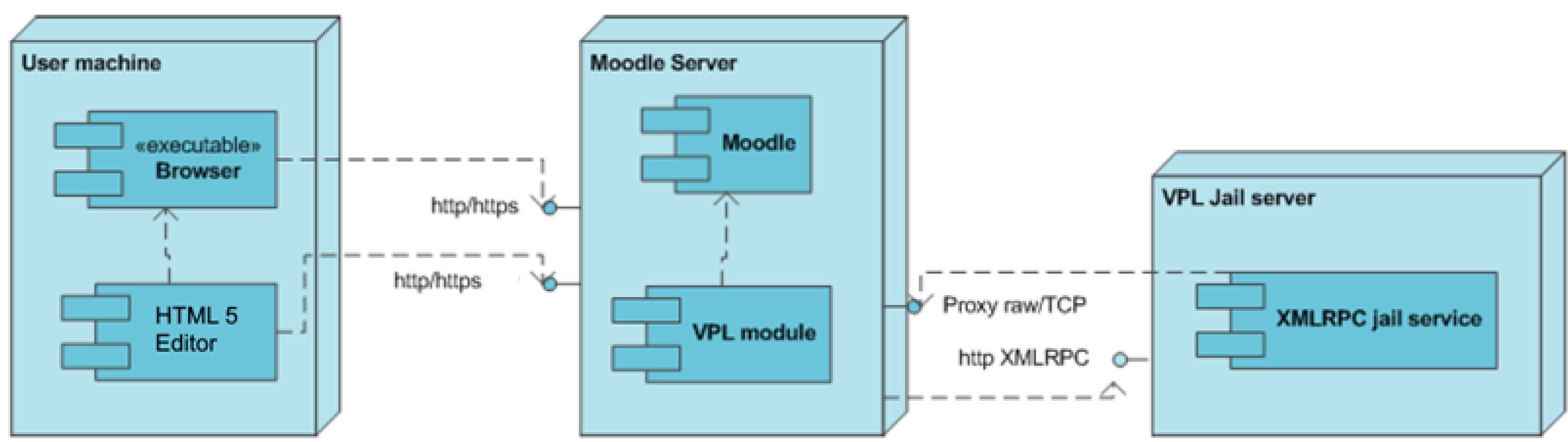
Diese repetitiven Schritten automatisiert VPL und bietet folgende Funktionen – natürlich alles per Browser:

- Studierende: Editieren von Quellcode für zahlreiche Programmiersprachen
- Studierende: Ausführen des Programms
- Dozent_innen können automatische Tests für Reviews hinterlegen und die Abgabe bewerten lassen.
- Dozent_innen können ähnlichen Quellcode in Abgaben finden.
- Dozent_innen können Restriktionen z.B. gegen Copy&Paste hinterlegen.
- Dozent_innen können oblig. Quelldateien einbinden, für erw. Testen, Entwurfsmuster sowie Codingstyle.

Einführung und erste Schritte:
Überblick

- Entwickelt durch J.C. Rodríguez-del-Pino von der Universität Las Palmas auf Gran Canaria (ULPGC) seit 2009/10
- PHP-Modul für Moodle (Add-In) und separate Sandbox für die Programmausführung (Jail-Server)
- GPL 3.0 Lizenz
- Alles unter Linux…

Einführung und erste Schritte:
Bestandteile



Quelle: nach Rodríguez-del-Pino (2012)

Einführung und erste Schritte:
Nutzung einer VPL Aktivität als Teiln.

Dafür bitte in den Kurs „eLT0319 - Virtual Programming LAB“ wechseln und dann:

- Die C++ Programmieraufgabe „Übung 1: Hallo HTW (C++)“ bearbeiten - in der Rolle *Teilnehmer_in*
- Für C++ Newcomer - kleine Hilfestellung:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hallo HTW";
    return 0;
}
```

Einführung und erste Schritte:
Editor

main.cpp

```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hallo HTW";
    return 0;
}
```

Bewertungsvorschlag:
10 / 10

Kommentare

Summary of tests

```
+------------------
|  1  test run/ 1  t
+------------------
```

Ausführung

Beschreibung

VPL

Einführung und erste Schritte:

Erkenntnisse

- Standardmäßig testet und bewertet VPL den Quellcode über STDIN/OUT.

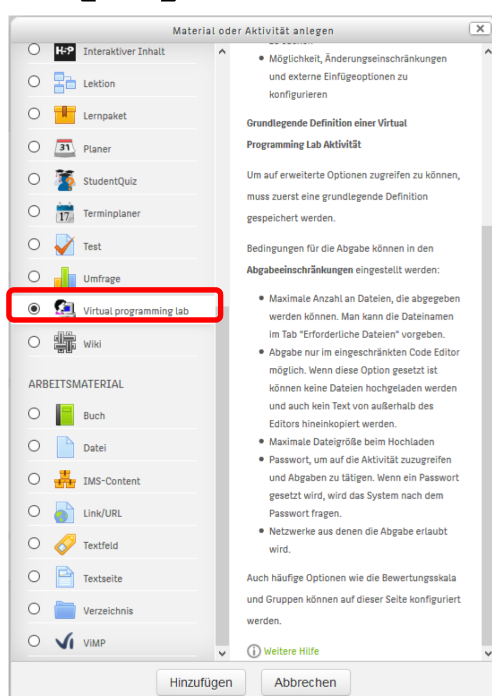
- Die jeweilige Programmiersprache wird durch die verwendete Erweiterung (hier: CPP) bestimmt.

- Editor bietet zwar Hervorhebungen für die Syntax, aber keine Quellcodevervollständigung.

Einführung und erste Schritte:
Anlegen einer VPL Aktivität (1)

Dafür bitte in den Testkurs mit Ihrem Namen wechseln - in der Rolle *Dozent_in*.

- Bitte legen Sie eine VPL Aktivität vergleichbar zu „Hallo HTW“ an.



Einführung und erste Schritte:
Anlegen einer VPL Aktivität (2)

Hinweise:

- Allgemeines:
  - Beschreibung: *Aufgabenstellung hinterlegen*
- Abgabeeinschränkungen:
  - Maximale Anzahl an Dateien: 1
- Bewertung:
  - Maximalpunkte: 10
  - Bewertung zum Bestehen: 5

Dann speichern.

Einführung und erste Schritte:

Anlegen einer VPL Aktivität (3)

Bearbeiten -> Einstellungen:

- Testfälle:

  Case = *Name des Testfalls*
  output = "Hallo HTW‟

  dann speichern.
- Ausführungsoptionen:
  - Ausführen: *Ja*
  - Evaluieren: *Ja*
  - Automatische Bewertung: *Ja*
- Erforderliche Dateien:
  - Neue Datei erstellen: *main.cpp*, danach speichern.

Einführung und erste Schritte:
Welche Programmiersprachen?

Die auf dem Jail-Server verfügbaren Programmiersprachen können folgendermaßen ermittelt werden:

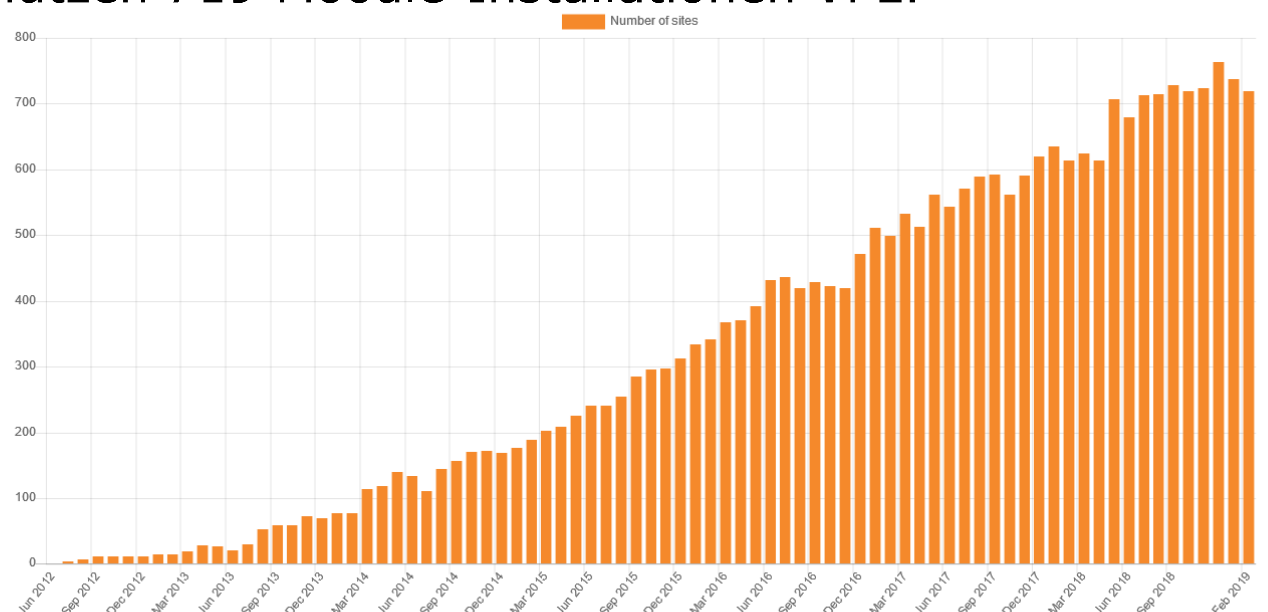
- VPL Aktivität anlegen
- Erweiterte Einstellungen -> "Maximaler Ressourcenverbrauch bei Ausführung" -> "Maximaler Speicherbedarf" z.B. auf 128 MB setzen
- a.all anlegen, Leerzeile einfügen, speichern und ausführen

Konsole: Verbindung geschlossen (Läuft: 4 seg)

```
 1 ADA: Hello from the Ada language!
 2 C: Hello from the C language!
 3 CLOJURE: Hello from the Clojure language!
 4 CPP: Hello from the C++ language!
 5 CSHARP: Hello from the C# language!
 6 D: Hello from the D language!
 7 ERLANG: Hello from the Erlang language!
 8 FORTRAN:  Hello from the Fortran language!
 9 GO: Hello from the Go language!
10 HASKELL: Hello from the Haskell language!
11 JAVA: Hello from the Java language!
12 JAVASCRIPT: Hello from the JavaScript language!
13 LISP: Hello from the LISP language!
14 LUA: Hello from the Lua language!
15 MATLAB: Hello from Matlab/Octave!
16 PASCAL: Hello from the Pascal language!
17 PERL: Hello from the Perl language!
18 PHP: Hello from the PHP language!
19 PROLOG: Hello from the Prolog language!
20 PYTHON-2: Hello from the Python2 language!
21 PYTHON-3: Hello from the Python3 language!
22 PYTHON: Hello from the Python language!
23 RUBY: Hello from the Ruby language!
24 SCALA: Hello from the Scala language!
```

Einführung und erste Schritte:
Wer nutzt VPL?

Aktuell nutzen 719 Moodle-Installationen VPL:



Quelle: moodle.org (2019)

Einführung und erste Schritte:
Literaturquellen

- Informationen etc. zum Add-In: http://vpl.dis.ulpgc.es/
- eBook: Wangenheim, A. et al. „Developing Programming Courses with Moodle and VPL", z.B. verfügbar hier: https://www.buecher.de/shop/fachbuecher/developing-programming-courses-with-moodle-and-vpl-ebook-epub/von-wangenheim-aldo/products_products/detail/prod_id/48776398/
- Moodle VPL Tutorials: http://www.science.smith.edu/dftwiki/index.php/Moodle_VPL_Tutorials
- Im Forschung&Lehre Wiki: https://wiki.htw-berlin.de/confluence/display/htwmoodlevpl/HTW+Moodle+VPL+Startseite

Grundlagen: Tests mit Testfällen (Wertepaare)

Grundlagen: Tests mit Testfällen (Wertepaare): Fakultät (1)

Dafür bitte in den Testkurs mit Ihrem Namen wechseln - in der Rolle *Dozent_in*.

- Bitte legen Sie eine VPL Aktivität in C# (Erweiterung cs) oder in Java (Erweiterung java) für folgende Aufgabenstellung an:

Erstellen Sie in C#/Java ein einfaches Konsolenprogramm, das von einer über die Konsole eingegebenen positiven Ganzzahl die Fakultät berechnet und das Ergebnis auf der Konsole ausgibt.
Verwenden Sie dafür die bereits vorhandene *main.cs/Program.java* Datei.

Grundlagen: Tests mit Testfällen (Wertepaare): Fakultät (2)

Bearbeiten -> Einstellungen:

- Testfälle:

  Case = Test 0!
  input = 0
  output = 1
  Case = Test 1!
  input = 1
  output = 1
  Case = Test 3!
  input = 3
  output = 6
  Case = Test 10!
  input = 10
  output = 3628800

Grundlagen: Tests mit Testfällen (Wertepaare): Fakultät (3)

Bearbeiten -> Einstellungen:

- Erweiterte Einstellungen -> Maximaler Ressourcenverbrauch bei Ausführung:
  - Maximaler Speicherbedarf: 128 MB

Grundlagen: Tests mit Testfällen (Wertepaare): Fakultät (4)

Kleine Hilfestellung (C#):

```
using System;
namespace Factorial
{
    class Program
    {
        static void Main(string[] args)
        {
            string input = Console.ReadLine();
            uint number;
            if (uint.TryParse(input, out number))
                 Console.WriteLine(CalculateFactorial(number));
        }

        static uint CalculateFactorial(uint number)
        {
            uint factorial = 1;
            while (number > 1)
            {
                factorial *= number;
                number--;
            }
            return factorial;
        }
    }
}
```

Grundlagen: Tests mit Testfällen (Wertepaare): Fakultät (5)

Kleine Hilfestellung (Java):

```
public class Program
{
        public static void main(String[] args)
        {
                java.util.Scanner scanner = new java.util.Scanner(System.in);

                System.out.println(calculateFactorial(scanner.nextInt()));
        }

        static int calculateFactorial(int number)
        {
                int factorial = 1;
                while (number > 1)
                {
                        factorial *= number;
                        number--;
                }
                return factorial;
        }
}
```

Grundlagen: Tests mit Testfällen (Wertepaare): Zusammenfassung

- Für einfache Aufgaben recht gut geeignet.
- Für die Definition der Ausgangswerte der Testfälle können auch Regular Expressions verwendet werden.
- Für komplexere Aufgaben insbesondere im OO-Bereich und für komplexe Datentypen zeigen sich schnell die Grenzen dieser Vorgehensweise.

Grundlagen: Tests mit Testfällen (Wertepaare): Weitere Optionen (1)

- Abgabeeinschränkungen:
  - Arbeitstyp: Einzelarbeit vs. Gruppenarbeit – für Bewertung
  - Abgabe durch eingeschränkten Codeeditor: keine Upload, kein Copy&Paste
  - Dies ist eine Beispielaktivität: Beispielcode – keine Veränderungen möglich
  - Erlaubte Online-Abgabe: *Ignore this for now. …and for ever*
- Bewertung:
  - Reduction by automatic evaluation: Reduce final score by a value or percentage for each automatic evaluation requested by the student
  - Free evaluations: Number of automatic evaluations that do not reduce final score

Grundlagen: Tests mit Testfällen (Wertepaare):
Weitere Optionen (2)

- Ausführungsoptionen:
  - Nur bei Abgabe evaluieren: Die Abgabe wird automatisch evaluiert, sobald sie hochgeladen wird.
  - Automatische Bewertung: Wenn das Evaluierungsergebnis Bewertungen enthält, werden diese automatisch als Bewertung für die Abgabe angewandt.

Wir finden dich: Plagiaterkennung

Wir finden dich: Plagiaterkennung
Überblick

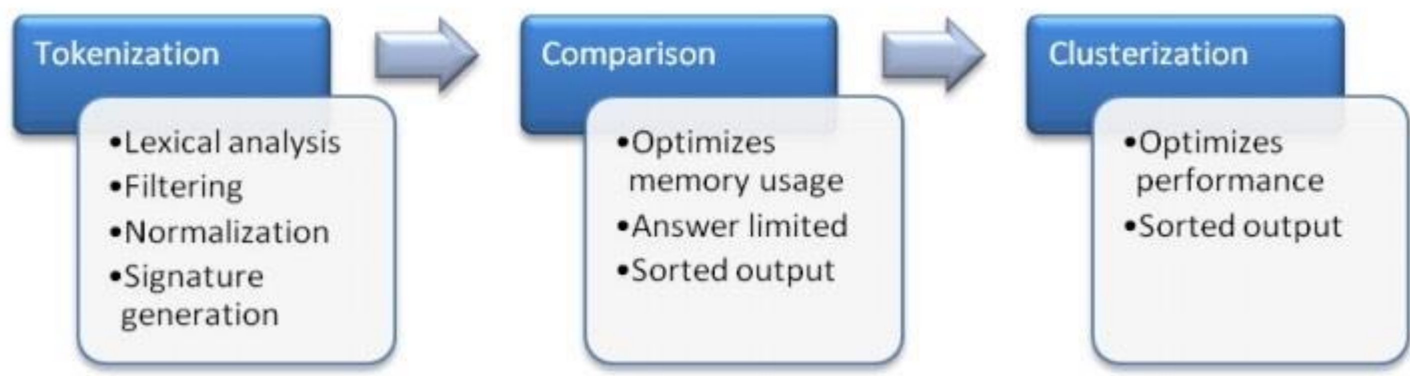
Am häufigsten genutzte Möglichkeiten ein Plagiat zu verschleiern:

- Änderungen an Kommentaren
- Umbenennungen von Klassen, Methoden, Variablen und Argumentnamen
- Umstrukturierung des Quellcode
- Ersetzung von Ausdrücken durch äquivalente Ausdrücke
- Komplexere Änderungen

Quelle: Wangenheim et al. (2015)

Wir finden dich: Plagiaterkennung
Konzept der Ähnlichkeitsanalyse in VPL

- Ähnlichkeitserkennung basiert auf syntaktischer Analyse



Quelle: Wangenheim et al. (2015)

Wir finden dich: Plagiaterkennung
Metriken (1)

Basierend auf den erzeugten Signaturen wird ein Vergleich mittels dreier Ähnlichkeitsmetriken (in %) vorgenommen:

Change / Metric	Comments	Name of Identifiers	Code reorder	Systematic change (expression)	Complex change (expression)
Metric 1	Filtered	Filtered	Not affected	Slightly affected	Affected
Metric 2	Filtered	Filtered	Not affected	Affected by size of changes	Affected by size of changes
Metric 3	Filtered	Filtered	Affected	Affected by number of changes	Affected by number of changes

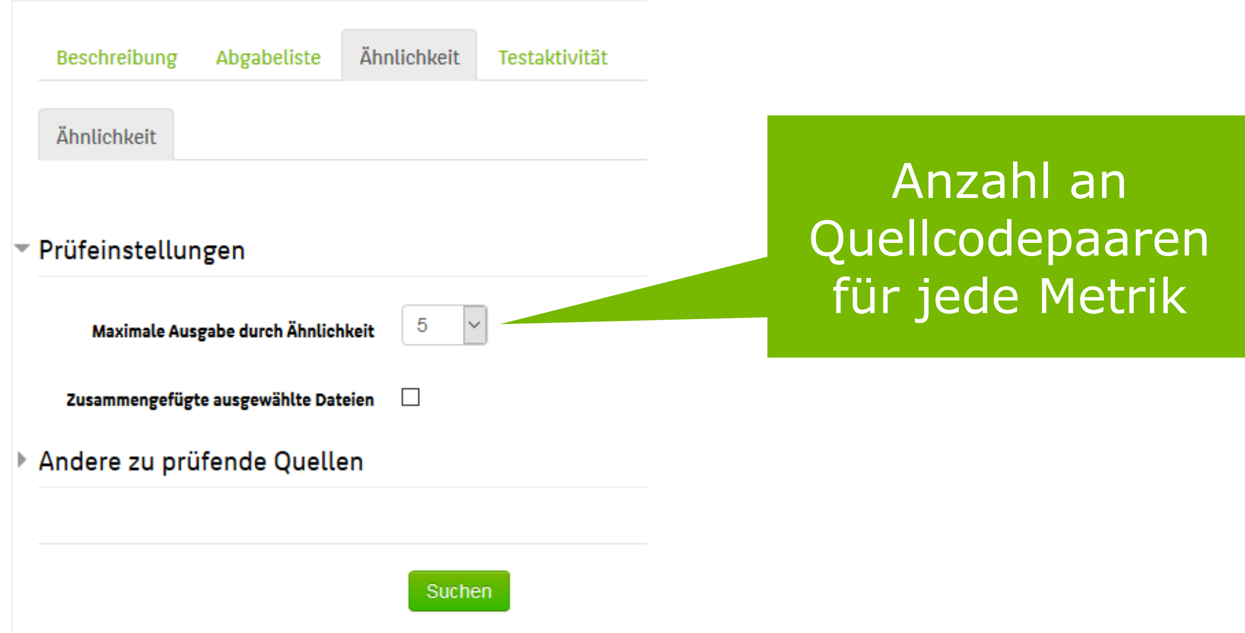
Quelle: Wangenheim et al. (2015)

Wir finden dich: Plagiaterkennung
Metriken (2)

- Metrik 1:
  - Erfasst Quellcodeähnlichkeiten, selbst wenn der Code umstrukturiert wurde
- Metrik 2:
  - Erfasst Quellcodeähnlichkeiten, selbst wenn der Code umstrukturiert wurde. Beeinflusst durch die Größe von Änderungen von Ausdrücken etc.
- Metrik 3:
  - Beeinflusst durch die Anzahl der Änderungen von Ausdrücken etc.

Quelle: Wangenheim et al. (2015)

Wir finden dich: Plagiaterkennung
Praktisches Vorgehen (1)



Wir finden dich: Plagiaterkennung
Praktisches Vorgehen (2)

Wir finden dich: Plagiaterkennung

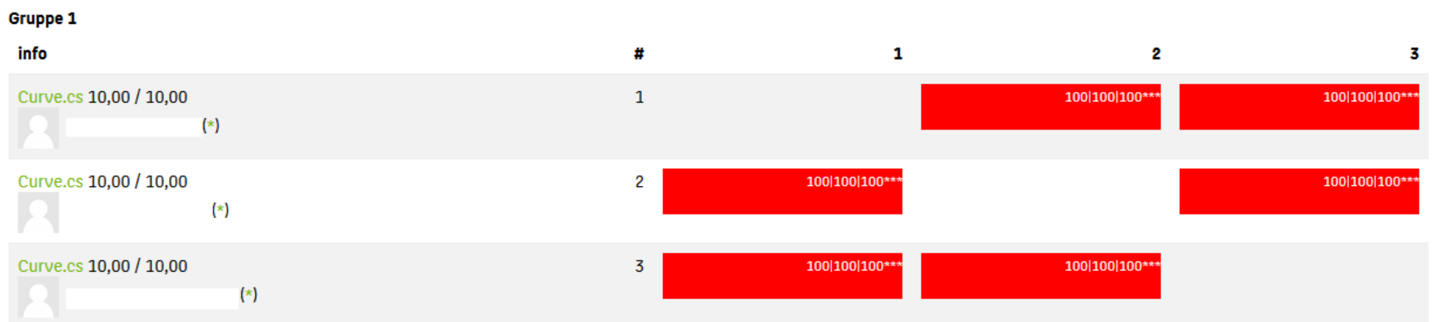
Praktisches Vorgehen (3)

Klick auf Ähnlichkeitswerte öffnet Diff:

6	`namespace LSP`	===	5	`namespace LSP`
7	`{`	===	6	`{`
8	`/// <summary>`	===	7	`/// <summary>`
9	`/// Class representing a line consisting of multi`	===	8	`/// Class representing a line consisting of multi`
10	`/// The polyline may be open or closed.`	===	9	`/// The polyline may be open or closed.`
11	`/// </summary>`	===	10	`/// </summary>`
12	`public class Polyline : Curve`	===	11	`public class Polyline : Curve`
13	`{`	===	12	`{`
14	`protected List<Point> _points = new List<Point`	=##	13	`protected readonly List<Point> _points = new`
15		===	14	
16	`public Polyline(bool isClosed = false)`	===	15	`public Polyline(bool isClosed = false)`
17	`: base(isClosed)`	===	16	`: base(isClosed)`
18	`{`	===	17	`{`
19	`}`	===	18	`}`
20		===	19	
21	`public Polyline(IEnumerable<Point> points, boo`	===	20	`public Polyline(IEnumerable<Point> points, boo`
22	`: this(isClosed)`	===	21	`: this(isClosed)`
23	`{`	===	22	`{`
24	`_points = points.ToList();`	===	23	`_points = points.ToList();`
		>>>	24	`}`
		>>>	25	
		>>>	26	`/// <summary>`
		>>>	27	`/// Add a point to the polyline.`
		>>>	28	`/// </summary>`
		>>>	29	`/// <param name="point">The point to add.</pa`
		>>>	30	`public void AddPoint(Point point)`
		>>>	31	`{`
		>>>	32	`_points.Add(point);`
		>>>	33	`}`
		>>>	34	
		>>>	35	`/// <summary>`
		>>>	36	`/// Remove a point from the polyline at the g`
		>>>	37	`/// </summary>`
		>>>	38	`/// <param name="index">The index for the poi`
		>>>	39	`public virtual void RemovePoint(int index)`
		>>>	40	`{`
		>>>	41	`_points.RemoveAt(index);`
25	`}`	===	42	`}`
26		===	43	
27	`/// <summary>`	===	44	`/// <summary>`
28	`/// The polyline's points.`	===	45	`/// The polyline's points.`
29	`/// </summary>`	===	46	`/// </summary>`
30	`public IReadOnlyList<Point> Points`	===	47	`public IReadOnlyList<Point> Points`
31	`{`	===	48	`{`
32	`get { return _points; }`	===	49	`get { return _points; }`
33	`}`	===	50	`}`
34		<<<		

Wir finden dich: Plagiaterkennung
Praktisches Vorgehen (4)

Clustering deckt auf, dass alle Curve.cs Dateien vom selben Original stammen:



Gruppe 1

<table>
<tr><th>info</th><th>#</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td>Curve.cs 10,00 / 10,00
(*)</td><td>1</td><td></td><td>100|100|100***</td><td>100|100|100***</td></tr>
<tr><td>Curve.cs 10,00 / 10,00
(*)</td><td>2</td><td>100|100|100***</td><td></td><td>100|100|100***</td></tr>
<tr><td>Curve.cs 10,00 / 10,00
(*)</td><td>3</td><td>100|100|100***</td><td>100|100|100***</td><td></td></tr>
</table>

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Überblick

Unit-Testing-Frameworks:

- Implementierung von Unit-Tests als Quellcode während der Entwicklung
- Automatisierte Ausführung der Unit-Tests z.B. im Rahmen von CI-Umgebungen
- Hierbei werden elaborierte Assertions für die Verifikation der Übereinstimmung zwischen aktuellem und erwartetem Wert angeboten.
- Für die relevanten Programmiersprachen verfügbar

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Idee für Anfänger

Die Studierenden implementieren den notwendigen Quellcode für die gestellte Aufgabenstellung zusammen mit der **main-Methode** z.B. in externer IDE.

- Anwendbar für **erste Semester** ohne Kenntnisse im Unit-Testing. Die Unit-Tests sind in VPL bereits hinterlegt (Ausführungsdateien).

- **Ausführen**: Kompilieren + ausführen – wie bisher

- **Evaluieren**: Kompilieren + ausführen der Unit-Tests + Testen und Bewerten

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Fakultät (1)

Dafür bitte in den Testkurs mit Ihrem Namen wechseln - in der Rolle *Dozent_in*.

- Bitte die vorherige Übung in C# kopieren oder neu anlegen.
- Folgende Aufgabenstellung verwenden:

Erstellen Sie in C# ein einfaches Konsolenprogramm, das von einer über die Konsole eingegebenen positiven Ganzzahl die Fakultät berechnet und das Ergebnis auf der Konsole ausgibt.
Verwenden Sie dafür die bereits vorhandene *main.cs* Datei.
Benennen Sie die interne, statische Methode zur Fakultätsberechnung *CalculateFactorial* und platzieren Sie diese im Namensraum *Factorial* in der Klasse *Program*. Deklarieren Sie ein sinnvolles Argument und einen passenden Rückgabetyp.

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Fakultät (2)

Bearbeiten -> Einstellungen:

- Testfälle – bitte leeren

- Erweiterte Einstellungen -> Ausführungsdateien
  - Hinzufügen der 2 Dateien aus Materialien „Für Übung 4: Bash-Skripte für NUnit-Integration (C#)"
  - Für *vpl_run.sh* kopieren wir das Original für C# (*csharp_run.sh*) aus der aktuellen VPL-Version:

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Fakultät (3)

Zugriff auf VPL-Quellcode:

- VPL-Quellcode auf github: https://github.com/jcrodriguez-dis/moodle-mod_vpl
- Die Skripte für Run, Debug und Evaluate befinden sich im jeweils aktuellen Branch (z.B. *v3.3.4*) nicht in *master*:
  jail/default_scripts/

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Fakultät (4)

Bearbeiten -> Einstellungen:

- Erweiterte Einstellungen -> Ausführungsdateien
  - Als 1. Zeile von *vpl_run.sh* fügen wir hinzu:
    [-e FactorialTests.cs] && rm FactorialTests.cs
  - Dann fügen wir unsere Unit-Testfälle hinzu: *FactorialTests.cs*

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Fakultät (5)

Kleine Hilfestellung (C#):

```
using NUnit.Framework;
namespace Factorial
{
    [TestFixture]
    public class CalculateFactorialTests
    {
        [Test]
        public void FactorialOf0()
        {
            Assert.AreEqual(1, Program.CalculateFactorial(0), "Factorial of 0 is 1.");
        }
        [Test]
        public void FactorialOf1()
        {
            Assert.AreEqual(1, Program.CalculateFactorial(1), "Factorial of 1 is 1.");
        }
        [Test]
        public void FactorialOf3()
        {
            Assert.AreEqual(6, Program.CalculateFactorial(3), "Factorial of 3 is 6.");
        }
        [Test]
        public void FactorialOf10()
        {
            Assert.AreEqual(3628800, Program.CalculateFactorial(10), "Factorial of 10 is 3628800.");
        }
    }
}
```

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: JUnit (1)

Für JUnit existiert eine JUnit VPL Test-App (https://github.com/bytebang/vpl-junit) mit folgendem Funktionsumfang:

- Unit-Testing mit spezifischer Punktvergabe per Test
- Testen des Console Output
- Interaktion mit dem Programm über Konsole
- Programmierstyle

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: JUnit (2)

Entsprechend dem dortigen **ReadMe**: Bearbeiten -> Einstellungen:

- Erweiterte Einstellungen -> Ausführungsdateien
  - Hinzufügen der JAR-Datei aus https://github.com/bytebang/vpl-junit/tree/master/release

- Erweiterte Einstellungen -> Dateien, die beim Ausführen behalten werden
  - Option setzen für JAR-Datei

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: JUnit (3)

Entsprechend dem dortigen **ReadMe**: Bearbeiten -> Einstellungen:

- Erweiterte Einstellungen -> Ausführungsdateien
  - Hinzufügen der 2 Dateien aus Materialien „Für Übung 4: Bash-Skripte für JUnit-Integration (Java)“
  - Dann fügen wir unsere Unit-Testfälle hinzu: *FactorialTests.java*

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: JUnit (6)

Kleine Hilfestellung (Java):

```java
import static org.junit.Assert.*;
import org.junit.Test;
public class FactorialTests
{
    @Test
    public void factorialOf0_3P()
    {
        assertEquals("Factorial of 0 is 1.", 1, Program.calculateFactorial(0));
    }
    @Test
    public void factorialOf1_3P()
    {
        assertEquals("Factorial of 1 is 1.", 1, Program.calculateFactorial(1));
    }
    @Test
    public void factorialOf3_2P()
    {
        assertEquals("Factorial of 3 is 6.", 6, Program.calculateFactorial(3));
    }
    @Test
    public void factorialOf10_2P()
    {
        assertEquals("Factorial of 10 is 3628800.", 3628800, Program.calculateFactorial(10));
    }
}
```

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Idee für Fortgesch.

Die Studierenden implementieren den notwendigen Quellcode für die gestellte Aufgabenstellung z.B. in externer IDE. Sie erhalten die Unit-Tests als Quellcode und integrieren diese. Es existiert **keine main-Methode**.

- ➢ Anwendbar für **höhere Semester** mit Kenntnissen im Unit-Testing. Abgegeben wird nur die eigentliche Implementierung, die Unit-Tests sind in VPL bereits hinterlegt (Ausführungsdateien).

- **Ausführen**: Kompilieren + ausführen der Unit-Tests.

- **Evaluieren**: Wie **Ausführen** + Testen und Bewerten.

Schon besser: Generischer Ansatz mit Unit-Testing-Frameworks und Test-App: Idee für Fortgesch.

- Komplette NUnit-Integration (auch für **Ausführen**):
  Bash-Skripte in „Für Fortgeschrittene: Alle Bash-Skripte für NUnit-Integration (C#)“

- Komplette JUnit-Integration für **Ausführen** ohne main-Methode:
  Bereits in den Bash-Skripten aus der aktuellen VPL-Version enthalten

Was geht noch?

Was geht noch?

Basiert auf (1)

- „*Basiert auf*“ Option ermöglicht es, **generische** VPL-Aktivtäten zu schreiben, die die Grundlage für mehrere andere Aktivitäten darstellen.

- Dozent_innen verfassen generischen Aktivitäten, um ein Framework für Programmierübungen aufzubauen.

- Für Studierende sollten die generischen Aktivtäten nicht zugänglich sein.

Was geht noch?
Basiert auf (2)

Auswirkungen der *„Basiert auf“ Option*:

- Ausführungsdateien der generischen Aktivitäten werden übernommen und mit Skript-Dateien (*vpl_run.sh*, *vpl_debug.sh* und *vpl_evaluate.sh*) der aktuellen Aktivität zusammengefasst.
- Ressourcenverbrauch der generischen Aktivität: Wird übernommen und kann überschrieben werden.
- Beschreibung der generischen Aktivität wird übernommen und ggfs. mit der Beschreibung der aktuellen Aktitvät zusammengefasst.
- Variationen dto.

Was geht noch?
Variationen

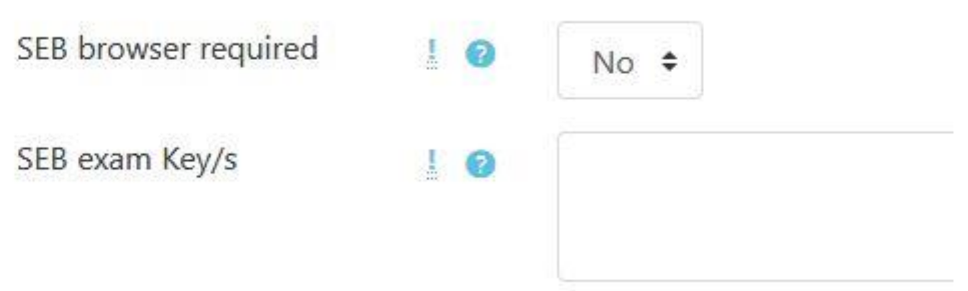
- Mit Variationen können VPL Aktivitäten spezialisiert werden.

- Letztlich erhält jeder Studierende eine individualisierte Aktivität.

➢ Bei Interesse weiter recherchieren.

Was geht noch?
VPL und eKlausuren

- Ab VPL 3.3.1 ist der Safe Exam Browser unterstützt:



Quelle: vpl.dis.ulpgc.es (2019)

Was geht noch?
VPL und Grafikprogramme

- Konsolenanwendungen werden mit dem Programmnamen *vpl_execution* gestartet.

- Grafikanwendungen hingegen mit dem Programmnamen *vpl_wexecution*.
  Dabei wird eine VNC Instanz erzeugt.
  - C# unter Mono: Für Windows.Forms
  - Java: Für Swing
  - C++ und andere Sprachen: Installation von GUI-Toolkits auf Jail-Server erforderlich…

Danke für die Ihre/Eure Aufmerksamkeit.

Gibt es Fragen?