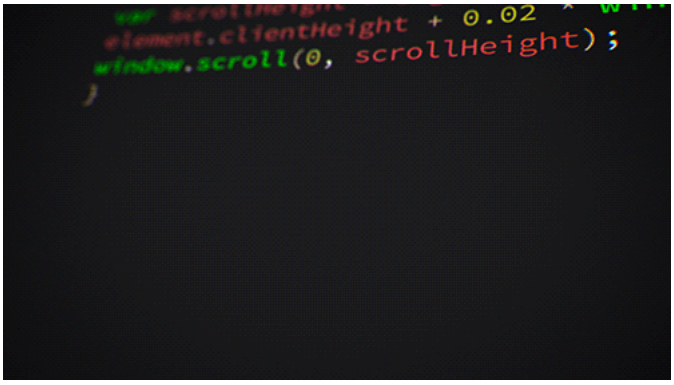
Continuous Integration

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2022`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Verwendung der Buildtools und der Features von GitHub in einer CI Toolchain`
Link auf den GitHub:	https://github.com/TUBAF-Ifi-LiaScript/VL_Softwareentwicklung/blob/master/18_ContinuousIntegration.md
Autoren	Sebastian Zug, Galina Rudolf & André Dietrich



Exkurs: Alternative Konzepte der Programmentwicklung

Das Jupyter Notebook ist eine Open-Source-Webanwendung Dokumente zu erstellen und zu teilen, die Live-Code, Gleichungen, Berechnungsergebnisse, Visualisierungen und andere Multimedia-Ressourcen zusammen mit erklärendem Text in einem einzigen Dokument integrieren. Dabei werden Markdown-Elemente mit verschiedenen Ausführungsumgebungen unterschiedlicher Sprachen (sogenannten Kernels) kombiniert. Notebooks sind insbesondere im Data Science-Kontext präsent, wo ganze Verarbeitungsketten von der Datenbereinigung und -transformation, numerische Simulation, explorative Datenanalyse, Datenvisualisierung, statistische Modellierung, maschinelles Lernen, Deep Learning usw. damit in Python umgesetzt werden.

Ein Jupyter-Notebook strukturiert die Implementierung in einzelnen Codeblöcke, die in beliebiger Reihung ausgeführt werden können. Zunächst geben Datenwissenschaftler Programmiercode oder Text in rechteckige "Zellen" auf einer Front-End-Webseite ein. Der Browser leitet den Code dann an einen Back-End-"Kernel" weiter, der den Code ausführt und die Ergebnisse zurück gibt. Es wurden bereits viele Jupyter-Kernel erstellt, die Dutzende von Programmiersprachen - unter anderem C# - unterstützen. Die Kernel müssen sich nicht auf dem lokalen Computer befinden.

Damit ist es eine interaktive Datenverarbeitung möglich, eine Umgebung, in der Benutzer Code ausführen, beobachten was passiert, interaktive Änderungen einpflegen usw. Aus diesem Grund eignet sich das Format gut um Tutorials oder interaktive Handbücher zu erstellen.

Neben lokalen Jupyter Installationen bieten sich verschiedene Webdienste an:

Projekt	Link	Kernel	Besonderheit
Collaboratory-Projekt von Google	colab	Python	GPU Unterstützung, Teamarbeit möglich
Binder	binder	Python, R, Julia, ...	
Kaggle	kaggle	Python, R	Sammlung von ausführbaren Lerninhalten

Eine gut Übersicht zu den Features bietet die Webseite DataSchool.

Visual Studio Code integriert ein eigenes Plugin für die Ausführung von .Net Interactive Sessions Tutorial

Nachteil 1: Die Kombinierbarkeit von Markdown und ausführbarem Sourcecode ist die zentrale Stärke von Jupyter Notebooks aber auch ihre Schwäche!

Das übergreifende Datenformat macht die Nachvollziehbarkeit von Code Änderungen schwer. Vor jedem Commit sollten entsprechend zumindest die Ausgaben gelöscht werden.

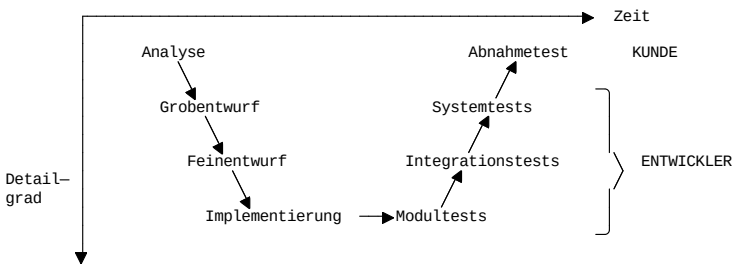
```
"cells": [
  {
   "cell_type": "markdown",
   "id": "33b4a983",
   "metadata": {},
   "source": [
    "# C# in Jupyter Notebooks"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "6c91aadb",
   "metadata": {},
   "outputs": [],
   "source": [
    "Console.WriteLine(\"Jupyter + .Net\")"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "bc55e2c6",
   "metadata": {},
   "outputs": [],
   "source": [
    "var i = 5;\n",
    "var j = 6;"
   ]
  },
```

Nachteil 2: Die beliebige Reihung der Aufrufe lässt einen mitunter die Übersicht verlieren.

Merke: Jupyter Notebooks sind ein hervorragendes Werkzeug für schnelle Prototypen, API-Dokumentationen oder Vorträge mit Live Hacks aber ungeeignet für Projekte [persönliche Meinung des Vortragenden 🙂].

Hinweis: Eine Beschreibung der Installationsprozedur für einen C#-Kernel finden Sie unter Link

Continuous integration (CI)



Continuous integration (CI) zielt darauf ab, die Qualität der Software zu verbessern und die Lieferzeit zu verkürzen, indem die das Gesamtprojekt kontinuierlich realisiert wird. Darauf aufbauend können die Konzepte der Qualitätskontroller automatisiert auf einer höheren Integrationsebene umgesetzt werden.

Tests lokal ausführen ... erste Stufe der CI ist die Durchführung lokaler Unit-Test in der lokalen Umgebung des Entwicklers. Auf diese Weise wird vermieden, dass die individuelle, aktuelle Entwicklungsarbeit das Gesamtprojekt beeinflusst.

Code in CI kompilieren ... ein Build-Server kompiliert den Code periodisch oder sogar nach jedem Commit und meldet die Ergebnisse an die Entwickler.

Tests in CI durchführen ... Zusätzlich zu automatisierten Unit-Tests werden kontinuierliche Prozesse der Qualitätskontrolle implementieren, die statische Analysen durchführen, die Leistung vermessen und profilieren, Dokumentationen extrahieren, etc.

Bereitstellen eines Artifacts von CI ... Nun ist CI oft mit kontinuierlicher Bereitstellung oder kontinuierlichem Einsatz in der so genannten CI/CD-Pipeline verflochten. CI stellt sicher, dass die auf der Hauptleitung eingecheckte Software immer in einem Zustand ist, der den Benutzern zur Verfügung gestellt werden kann, und CD macht den Bereitstellungsprozess vollständig automatisiert.

Damit ergeben sich folgende Aktivitäten, die für einen CI Realisierung benötigt werden:

- Aufbau und Management eines zentralen Code-Repositorys.
- Aufbau und Management eines zentralen Build-Tools.
- Schreiben von automatisierten Tests und Integration in ein Feedbacksystem
- Darstellung von Build-Resultaten und Artifacts

> Merke: Unterschätzen Sie den Aufwand für die Realisierung und Wartung der Tool-Chain nicht. Häufig müssen hier zu Beginn des Projektes grundsätzliche Entscheidungen getroffen werden, die zumindest mittelfristige Auswirkungen auf das Projekt haben.

CI Umsetzung mit GitHub

Welche Elemente machen somit eine CI Pipeline aus:

- **Wann** (Trigger) soll
- **Was** (Job)
- **Wo** (Runner) ausgeführt werden um
- **Welches Ergebnis** (Artefakt) zu generieren

Beschrieben werden die Pipelines unter GitHub and Gitlab in YAML einer Auszeichnungssprache für Datenstrukturen über sogenannte Folgen. Die YAML-Datei, die die Pipeline-Konfiguration spezifizieren müssen im GitHub-Repositorys im Verzeichnis .github/workflows liegen.

```yaml
# Kommentare sind auch erlaubt
scalar : 5
collection:
    variable1 : Tralla
    variable2 : Trulla
    variable3: |
            Hier steht jetzt plötzlich
            Ein Ausdruck mit mehreren Zeilen
list:
    - variableA: Wert1
    - variableB: 5
```

Die Grundstruktur folgt dabei einer Kombination der Elemente `name`, `on` (Wann) und `jobs` (was):

```yaml
Generate Documentation

on:
  push:
    branches:
    - master
    paths:
    - 'docs/**'

jobs:
  build:
    name: Build
    runs-on: ubuntu-latest
    steps:
      - run: scripts/generate_documentation.sh
        env:
          PUBLISH_TOKEN: ${{ secrets.PUBLISH_TOKEN }}
```

Trigger

Das Ausführen einer Pipeline wird aus verschiedenen Quellen resultieren und mit Bedingungen verknüpft werden.

```yaml
on: push

on: [push, pull_request]
```

Runner

Die Ausführung des "Was" kann auf jedem Rechner in Form eines Runners erfolgen, der entweder bei GitHub oder unabhängig gehostet wird. Wenn ein Runner einen Job aufnimmt, führt er die Aktionen des Jobs aus und meldet den Fortschritt, die Protokolle und die Endergebnisse an GitHub zurück.

```yaml
runs-on: ubuntu-latest

runs-on: [self-hosted, linux, ARM32]

runs-on: ${{ matrix.os }}
strategy:
  matrix:
    os: [ubuntu-16.04, ubuntu-18.04]
    node: [6, 8, 10]
```

Jobs

Der Runner stellt eine Ausführungsumgebung bereit innerhalb derer Sie nun alle auch lokal möglichen Operationen umsetzen können. Ein häufig verwendeter Ansatz ist die Verwendung von docker-Containern, die eine Softwarekonfiguration bereitstellen.

Der Marketplace Link stellt verschieden Actions bereit, die häufige Verwendung finden. Das bedeutet, dass man sich die Funktionalität nicht selbstständig zusammentragen muss, sondern allein über die Parameterisierung eine Anpassung vornimmt.

Für den Buildprozess einer .NET Anwendung werden im folgenden Beispiel Actions für das auschecken und die Bereitstellung des .NET Cores der Version 3.1.1 genutzt.

```yaml
name: .NET Build App

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Setup .NET Core
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: 3.1.101
    - name: Install dependencies
      run: |
        cd src/App
        dotnet restore
    - name: Build
      run: |
        cd src/App
        dotnet build --configuration Release --no-restore
```

Ergebnis

Artefakte ermöglichen es, Daten zwischen Aufträgen in einem Workflow auszutauschen und Daten zu speichern, sobald dieser Workflow abgeschlossen ist.

```
- uses: actions/upload-artifact@v2
  with:
    name: my-artifact
    path: path/to/artifact/world.txt
```

Die Stärken von GitHub-Actions liegen in der unmittelbaren Integration in das Repository-System. Damit entfällt die Bereitstellung einer weiteren Infrastruktur zumal automatische Check-ins von Ergebnissen einer CI Toolchain sehr kompakt möglich sind.

Welche Mängel gibt es noch im Gebrauch der GitHub-Actions?

- Workflows können nur eingeschränkt wiederverwendet werden.
- Testergebnisse und Analyseresultate können nicht über integrierte Formate ausgegeben werden (Wir sehen es gleich bei der Visualisierung der Unit-Tests im Beispiel)
- die Sequenz der Ausführungen mehrerer Pipelines ist nicht steuerbar
- die Liste der verfügbaren Actions ist noch sehr überschaubar

> Merke: Man merkt an einigen Stellen der GitHub CI Implmentierung an, dass diese noch recht jung ist. Features die für andere Tools etabliert sind, fehlen noch.

Alternative Realisierungen lassen sich zum Beispiel mit Jenkins oder TravisCI unabhängig von einem ursprünglichen Versionsmanagementsystem evaluieren.

Anwendungsbeispiel 1

Geben Sie die jeweiligen Anteile der verschiedenen Mitstreiter an einem Projekt in der README.md aus. Wer hat zum Beispiel wie viele Codezeilen realisiert.

Werfen wir dazu einen Blick auf das Anwendungsbeispiel. Folgende Bearbeitungsschritte werden im zugehörigen Python Skript durchlaufen:

	Bedeutung
1.	Extrahieren der Informationen mit Hilfe des Paketes `github2pandas`
2.	Generierung der Basisdatentabelle durch Merge von `pdEdits` und `pdCommits`
3.	Aggregieren der hinzugefügten und gelöschten Zeilen durch `groupby`
4.	Austauschen der Daten in der `README.md` Datei
5.	Generieren eines neuen Diagramms das bereits in der `README.md` Datei eingebunden ist.
6.	Commit und Push Operation mit den neuen Daten

Die zweimalige Ausführung der Action pro Tag wird durch einen Timer getriggert.

Anwendungsbeispiel 2

Wir wollen folgenden Entwurf in einem kontinuierlichen Entwicklungsfluss realisieren.

Für das Projekt entwerfen wir folgende Struktur:

```
├── doc
│   └── ...                     <- Generierte Dokumentation
├── Doxyfile                    <- Konfigurationsfile für die Doku
├── README.md                   <- Beschreibung des Projektes
├── src                         <- Sourcecode der Anwendung
│   ├── Animals
│   │   ├── Rooms.cs
│   │   ├── Cat.cs
│   │   ├── Dog.cs
│   │   ├── Pet.cs
│   │   ├── bin
│   │   │   └── Debug
│   │   │       └── ...
│   │   ├── obj
│   │   │   └── ...
│   │   └── Animals.csproj      <- .NET Projektdatei
```

Sie finden das Projekt unter https://github.com/SebastianZug/CSharpExample.

Realisierung der Projektstruktur

```
mkdir src
cd src
mkdir Animals
dotnet new classlib
cd ..
mkdir App
cd App
dotnet new console
```

Allerdings fehlt jetzt noch die Verbindung zwischen den beiden Projekten! Entsprechend müssen wir die zugehörigen Referenzen in den Project-Files integrieren. Dies kann entweder in der Konsole oder mittels Textoperation erfolgen.

```
dotnet add reference ..\Animals\Animals.csproj
```

.csproj

```
<ItemGroup>
  <ProjectReference Include="..\Animals\Animals.csproj" />
</ItemGroup>
```

Unsere Anwendung ruft einige der Funktionalitäten des `classlib` Projektes auf.

Program.cs

```
using System;
using System.Collections.Generic;
using Pets;

namespace ConsoleApplication {
    public class MyLittleZoo {
        public static void Main (string[] args) {
            Dog Willy = new Dog("Willy", Rooms.DiningRoom);
            Cat Kitty = new Cat ("KatziTatzi", Rooms.Kitchen);
            List<Pet> pets = new List<Pet> {Willy, Kitty};

            foreach (var pet in pets) {
                Console.WriteLine (pet.TalkToOwner ());
            }

            Willy.SearchForCat(Kitty);
            Willy.LocalizedInRoom = Rooms.Kitchen;
            Willy.SearchForCat(Kitty);
        }
    }
}
```

Automatischer Build Prozess

Der automatische Buildprozess ist im einfachsten Fall eine Kopie des lokalen. Allerdings ist das Resultat in diesem Fall nur bedingt aussagekräftig. Vielmehr wollen wir mit dem serverseitigen Test ja die Übertragbarkeit unserer Lösung auf unterschiedliche Betriebssysteme /.NET Frameworks evaluieren.

Für den ersten Fall müssen wir im Action File statt eines einfachen `run-on` eine Matrix von Runnern angeben. Im zweiten Fall muss auch der zugehörige Programmcode angepasst werden. Informieren Sie sich unter Link, wie Sie unterschiedliche Frameworks adressieren können.

.github/workflows/buildApp.yml

```yaml
name: .NET Build App

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Setup .NET Core
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: 3.1.101
    - name: Install dependencies
      run: |
        cd src/App
        dotnet restore
    - name: Build
      run: |
        cd src/App
        dotnet build --configuration Release --no-restore
```

Generierung der Dokumentation

Die Dokumentation wird unter Hinzuziehung von Doxygen erzeugt. Dieser Vorgang besteht aus drei Schritten:

1. dem Checkout des aktuellen Projektes in den Runner
2. dem Erzeugen der HTML Dokumentation ausgehend von der Konfiguration in der Datei Doxyfile
3. dem Publizieren des Ergebnisses im Branch `gh-pages`.

Mit dem letzten Schritt entsteht eine Projektwebseite, die die Dokumentation enthält. Zusätzlich wurde im Doxyfile die Integration des README.md Files als "Masterseite" konfiguriert.

Doxyfile

```
...
INPUT                  += ./README.md \
                          ./src/Animals

USE_MDFILE_AS_MAINPAGE = ./README.md
...
```

.github/workflows/buildDocs.yml

```yaml
name: Doxygen

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Doxygen Action
      uses: mattnotmitt/doxygen-action@v1.1.0
      with:
        doxyfile-path: "./Doxyfile"
        working-directory: "."
    - name: Deploy
      uses: peaceiris/actions-gh-pages@v3
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./docs
```

Die Darstellung ist unter https://sebastianzug.github.io/CSharpExample/ für das Projekt sichtbar.

Erweiterung der Tests

Bisher werden nur die Ausgaben von `Dog` und `Cat` untersucht. Lassen Sie uns einen Test integrieren, der die Methode `.SearchForCat()` evaluiert.

```
[Theory]
[InlineData(Rooms.Kitchen, Rooms.DiningRoom, false)]
[InlineData(Rooms.Kitchen, Rooms.Kitchen, true)]
public void DocCheckCatSearching(Rooms kittysLocation, Rooms willysLocation, bool
  pattern) {
    Cat Kitty = new Cat ("KatziTatzi", kittysLocation);
    Dog Willy = new Dog("Willy", willysLocation);
    bool actual = Willy.SearchForCat(Kitty);
    Assert.Equal (pattern, actual);
}
```

Die Tests sollen nun als CI-Tests auf dem GitHub-Server ausgeführt werden. Entsprechend ist die Integration einer neuen Action notwendig.

```yaml
name: .NET Test Animals

on:
  push:
    branches: [ master ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Setup .NET Core
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: 3.1.101
    - name: Run Tests
      run: |
        cd src/Tests
        dotnet test
```

Aufgaben der Woche

[ ] Klonen Sie mein Repository von https://github.com/SebastianZug/CSharpExample [ ] Ergänzen Sie in der Build Action ein weiteres Target, zum Beispiel "Windows" [ ] Erweitern Sie das Ganze um weitere Unit-Tests, ergänzen Sie den Eintrag in der README.md Datei [ ] Übernehmen Sie das Konzept der automatischen Generierung eines Klassendiagramm aus den Übungsblättern, so dass auf dem Deckblatt der Dokumentation die aktuelle Klassenstruktur, die automatisch generiert wurde, sichtbar wird.