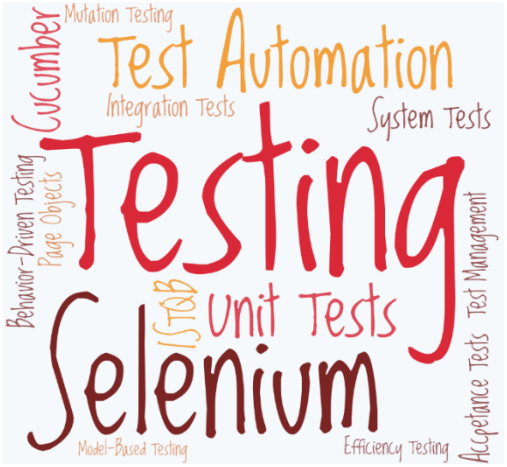
# 183.290 Software Testing




## Test Automation

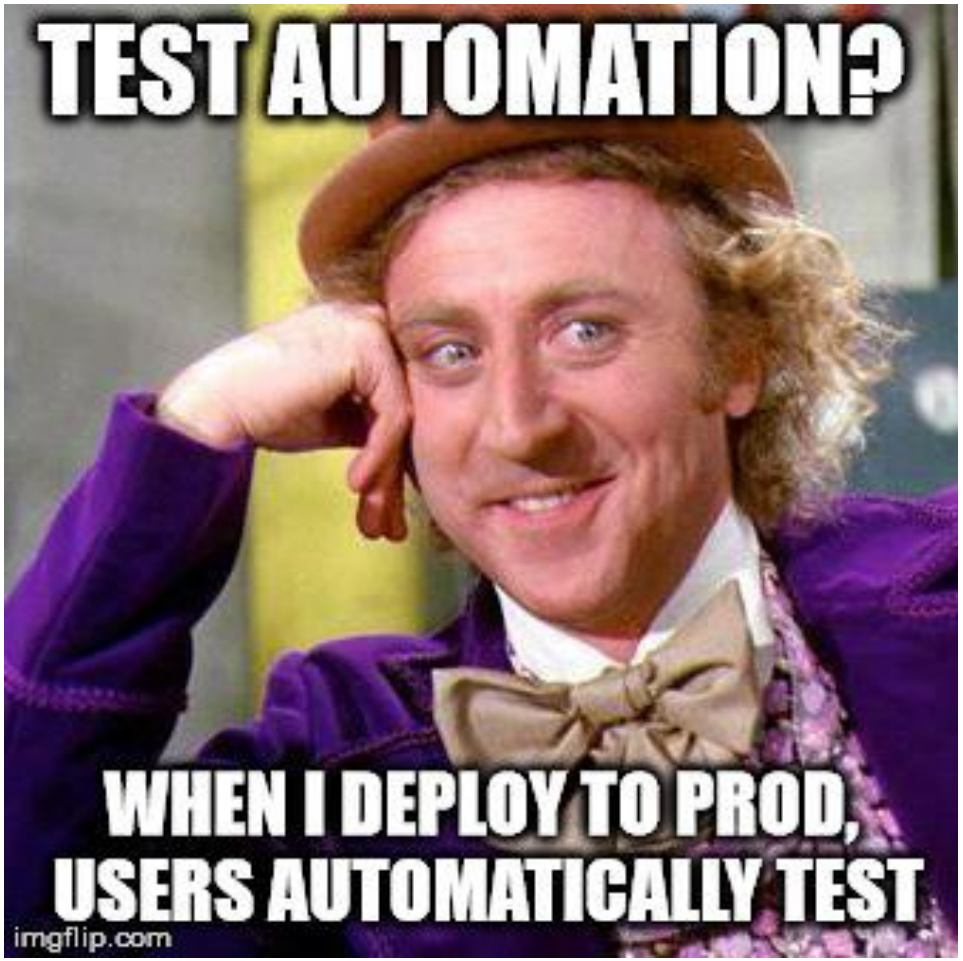
**WS 2020**

Christina Zoffi

peso@inso.tuwien.ac.at
https://peso.inso.tuwien.ac.at

# Content

# Test Automation



# Test Automation

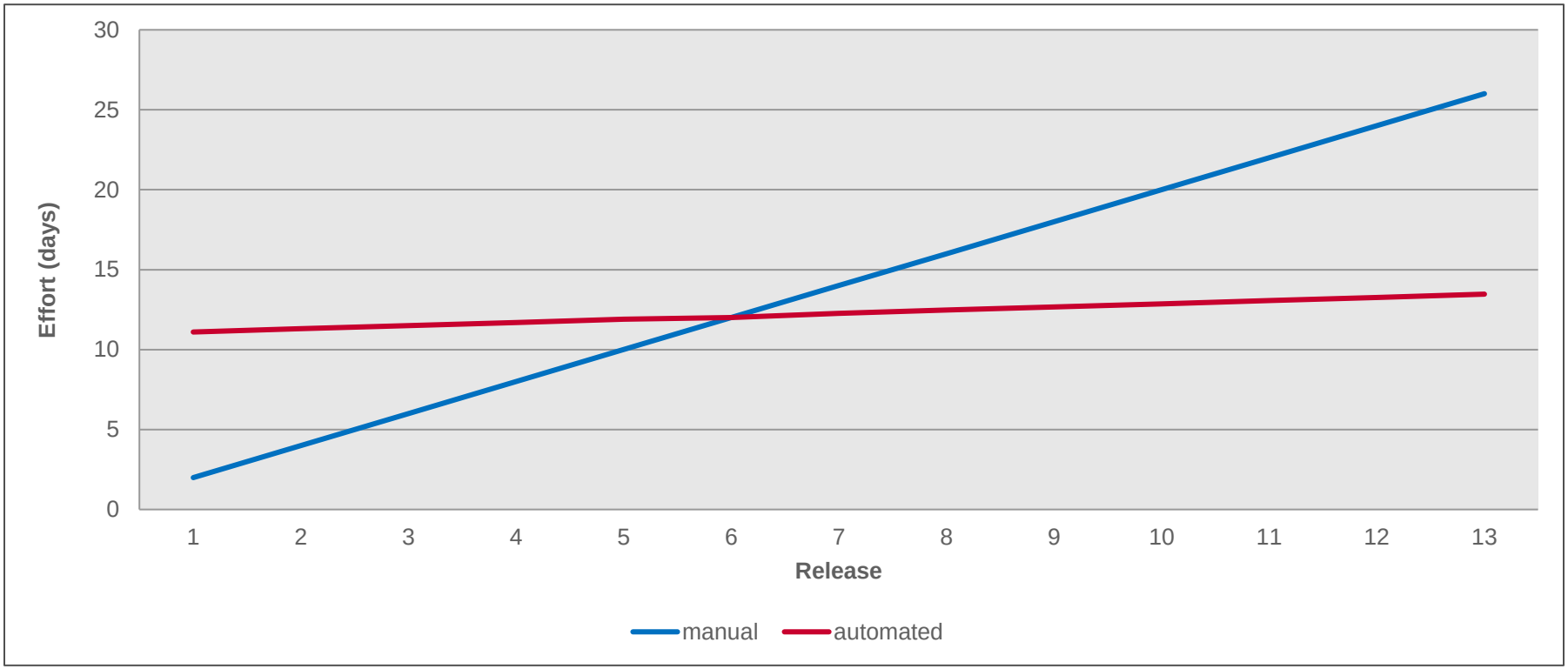
- **Definition**
  - The use of software to perform or support test activities, e.g., test management, test design, test execution and results checking.

- **Aims to:**
  - Reduce recurring and repetitive manual tasks (regression tests)
  - Repeatedly execute a high number of tests within reasonable time/effort
  - Automate activities that require significant resources
  - Automate activities that cannot be done manually
  - Improve efficiency of testing

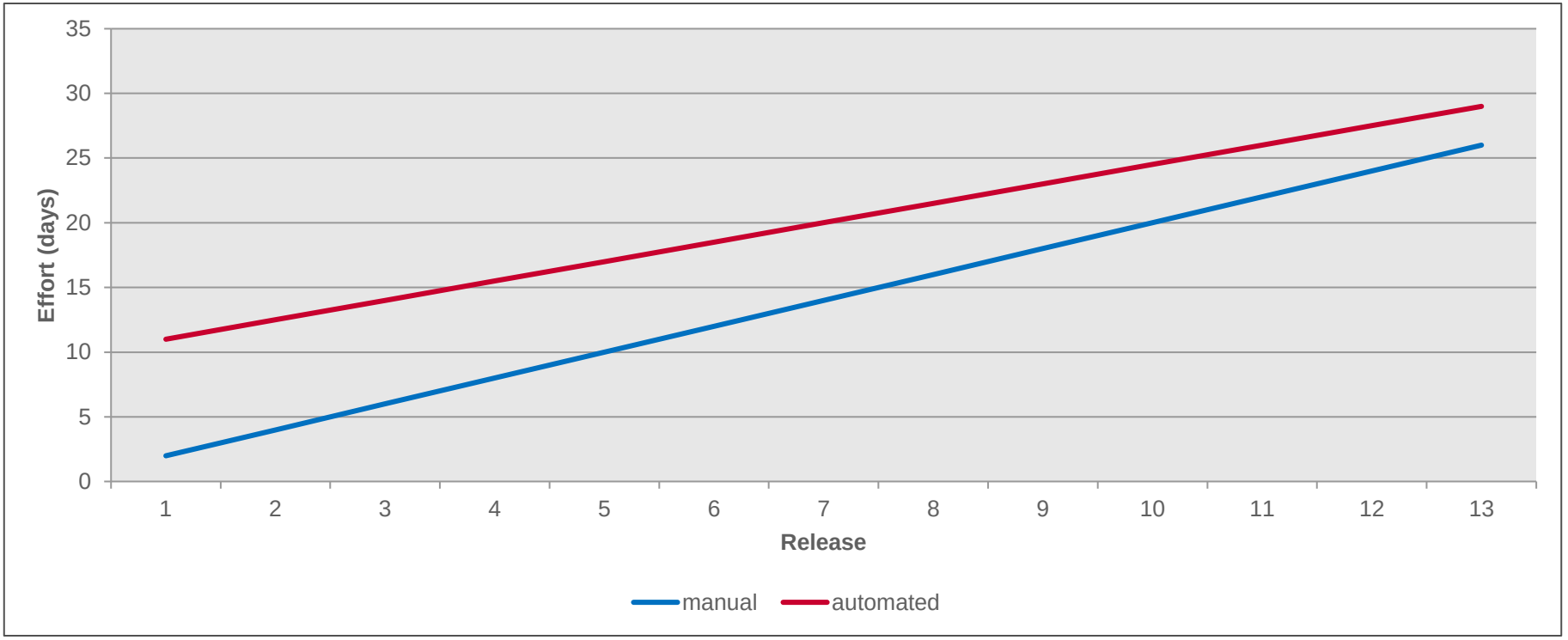
# Test Automation

- **Tool support does not automatically guarantee success/improvement**
- **Not every project benefits from test automation**
- **Decision for test automation is mostly driven economically, depending on:**
  - Planned duration/scope of the project
  - Planned number of releases (e.g. required test cycles)
  - Current project phase (begin, mid, end)
  - Type of project (greenfield, maintenance, legacy)
  - Knowledge/Experience of test team

# Test Automation Effort – As It Should Be



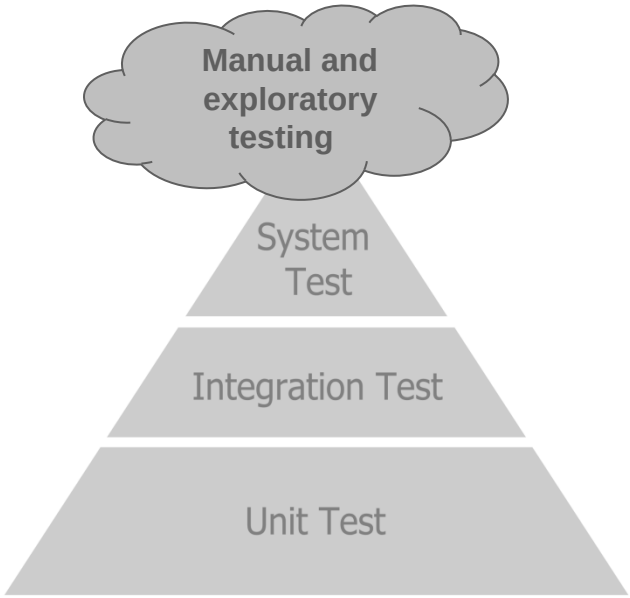
# Test Automation Effort – As It Should Not Be



# Content

# Automated Test Execution

- **Execution** of tests on **all test levels** can be **automated**
  - How to invest into automation efforts? → test automation pyramid
- **Execution frequency varies**
  - **Unit Tests**
    - Constant regression tests on all components
    - Most frequent execution
    - Candidates for continuous integration
    - e.g. every build
  - **Integration Tests**
    - Frequent execution
    - e.g. nigthly build
  - **System Test**
    - Rare execution
    - At least once per release



# Automated Test Execution – Benefits

- **Less manpower** required for test execution
- **Shorter duration** of test execution
- More **reliable regression** tests
- Enable test activities that **cannot** be **performed manually** (e.g. load/stress testing)
- **Reusability** of test scripts
- **More transparency**
  - More detailed test reports
  - Faster feedback
- **Continuous Integration**
  - Execution can be triggered automatically

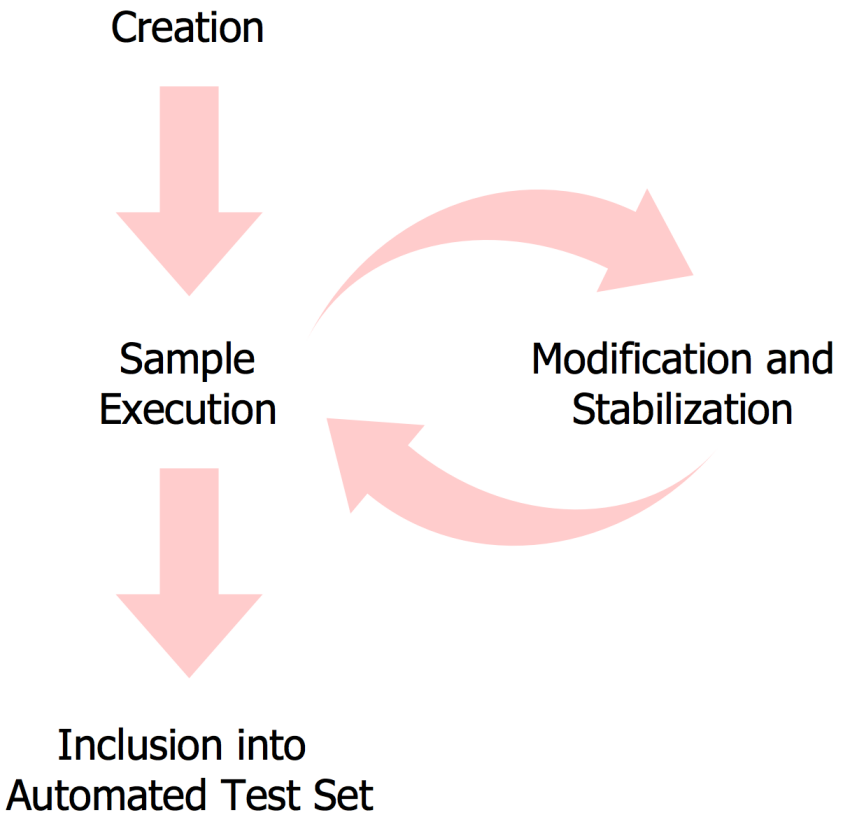
# Automated Test Execution – Drawbacks

- **High initial costs**
  - License
  - Training
  - Setup/Configuration
- **High effort** required for **initial creation of test scripts**
- **Higher maintenance costs**
  - Test scripts need to be adapted when software changes
  - Test scripts contain more details/logic
- Potential **loss of knowledge** in testing team
- „A tool can‘t think“
  - Tools can support, but not replace human testers

# Automated Test Execution

- **Automated tests only add value if they are stable and trustworthy**

- **Workflow when creating automated test scripts**
  - Create first draft
  - Sample execution to check if the test works
  - Improvement until the script is stable
  - Frequent execution



# Test Execution Approaches

- **Different approaches for test execution, depending on:**
  - Project scope
  - Project duration
  - Available staff
  - Test level

- **Common test approaches are**
  - Manual Testing
  - Script-Based Testing
  - Capture/Replay Testing
  - Model-Based Testing

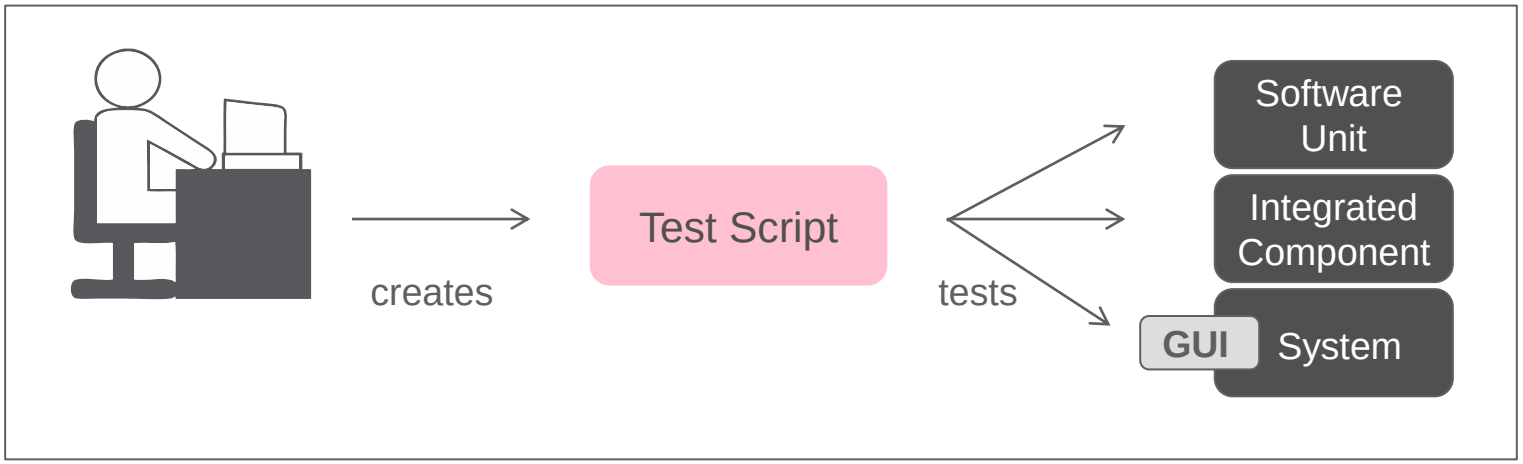
# Test Execution Approaches – Manual Execution

- **The following activities are done manually**
  - Test execution
  - Comparision between actual/expected result
  - Documentation of results in test report

- **No automation at all** (manual end-to-end test)
- **Can only be applied for system/acceptance testing**
- **Drawbacks**
  - Time intensive
  - Repetitive
  - Provides no information on general test coverage
  - No continuous status

# Test Execution Approaches – Script-Based

- Automated execution using test scripts
- Heavily used for regression testing on unit/integration level



# Test Execution Approaches – Script-Based

- **Benefits**
  - Fast execution
  - Automated result comparison
  - Gather information on general test coverage

- **Drawbacks**
  - Additional effort/time required for creation of scripts
  - Additional costs for maintenance

- **Knowledge of scripting/programming language is required**

# Test Execution Approaches – Script-Based

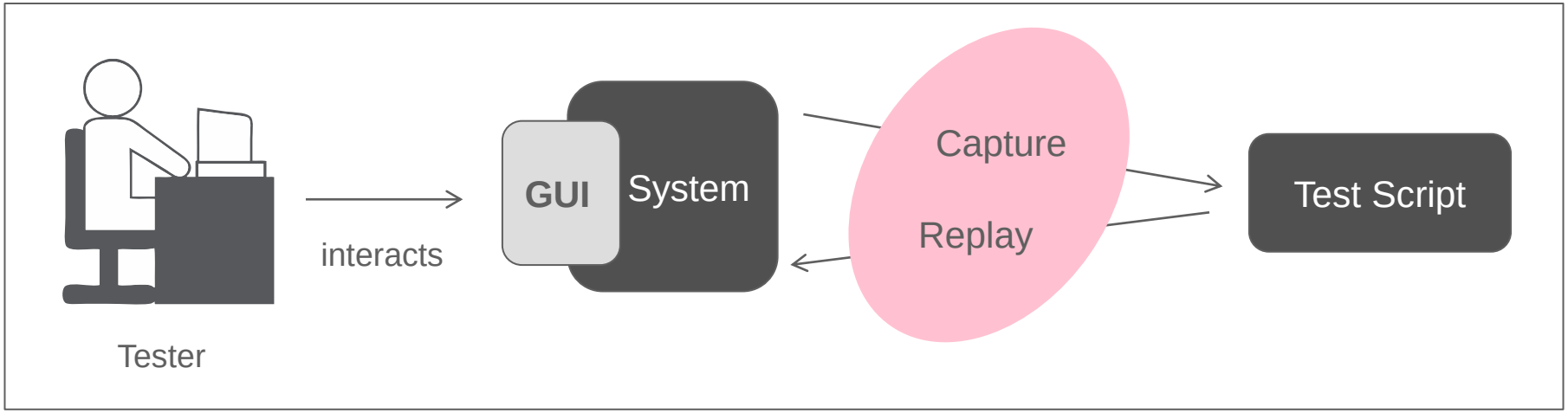
- **Usually implemented using a test framework**
  - Provides the testing environment
  - Handles the execution of the scripts
  - Provides assertion mechanisms
  - Supports definition of test suites/execution groups

- **Unit/Integration test scripts are usually maintained as part of the development project**

- **Examples – Java**
  - JUnit
  - TestNG

# Test Execution Approaches – Capture/Replay

- **Approach for automated UI system tests** (automated end-to-end test)
- **Capture Phase**
  - Test case is executed manually
  - Triggered actions/events are captured and exported to test script
- **Replay Phase**
  - Recorded actions/events are replayed automatically



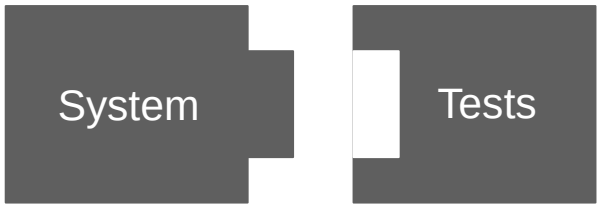
# Test Execution Approaches – Capture/Replay

- **Challenges**
  - High maintenance costs
  - Error-prone since scripts lack of abstraction
    - Code duplication
    - Missing modularity
    - Missing separation of input data
  - Quality/Robustness of scripts depends on quality of capturing tool and features of underlying scripting language
  - Format of exported scripts may be proprietary

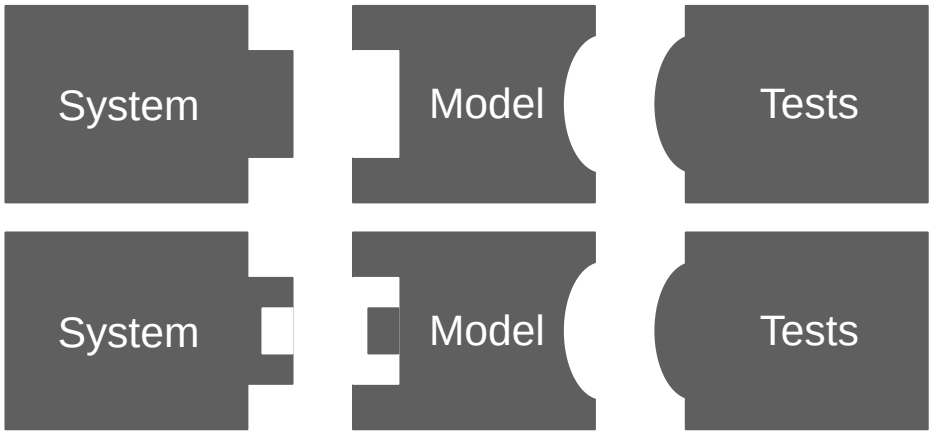
- **Tool Examples**
  - Marathon
  - Abbot
  - Selenium IDE (currently no export)

# Test Execution Approaches – Model-Based

- **Testing based on models**
- Creation of an **abstract model of the SUT**
- Model describes **requirements/desired behavior** of a system
- Model serves as **decoupling layer** between **productive system** and **test code**




**Usual Testing**




**Model-BasedTesting**

# Content

# Automated UI Tests

- **Definition**
  - Testing performed by interacting with the SUT via the graphical user interface.
  - Systemtest – Simulating interactions of real users
- **Challenges**
  - Addressing different platforms
    - Browsers
    - Devices (web vs. mobile)
    - Displays / Resolution
  - Robust test scripts are hard to achieve
    - Isolation of functionality from layout
    - Identification of UI elements
  - High complexity (event driven interaction)
  - Fragile against changes

# Software Volatility

- **Definition**
  - „Software volatility is defined as the propensity for software to change over time in response to evolving requirements.“
  - Software volatility describes the changeable nature of software.
    - Changing requirements
    - Changing technologies
- **UI is more fragile/volatile than server-side / backend code**
  - Changes in UI break test cases more easily
  - Worst Case Scenario:
    - Layout changes require complete rewrite of test cases

# Challenges – UI Element Identification

**How to automatically identify and interact with an UI element?**

# Challenges – UI Element Identification

- **Element characteristics**
  - **By id**
    - Very strong (unique, stable)
    - Often not defined for all elements by developers
    - Sometimes dynamically generated (by some UI frameworks)
  - **By CSS class**
  - **By visible text**
    - Language dependent
    - Prone to change over time
  - **By xPath query**
    - Use query language to describe path to element
    - Robustness depends on query
  - **By position**
    - Not robust/reliable
  - …

# Challenges – UI Element Identification

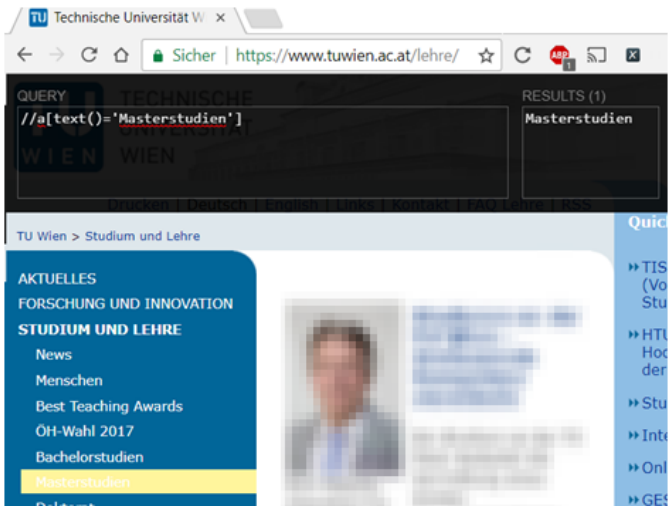
- **Absolute Position (Coordinates)**
  - click(3200, 422);


  - Not reliable, strongly depends on
    - Resolution
    - Hardware components (display, device)
    - Browser window size
  - Not robust against changes
    - e.g. swap/movement of UI elements
    - might lead to unexpected behavior

- **→ do not depend on absolute position of elements**

# Challenges – UI Element Identification

- **XML Path Language (XPath)**
  - Query language for selecting nodes from an XML document.
  - e.g. DOM tree

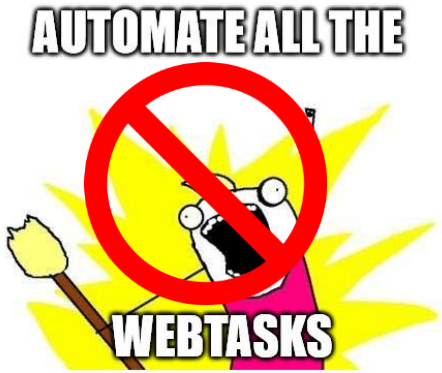



  - Relative and absolute statements
    - //button/span[text()= 'Start']
    - //select[@name = 'year']
    - //a[text()='Masterstudien']
  - **Use element characteristics that are less volatile**
  - **Avoid depending on absolute paths**
    - /html/body/div[5]/div/div[3]/ul/li[9]/a
    - /table[@id="postinglist"]/div[6]/div[1]/div/div[2]/div

# Selenium

- **Selenium is a portable software-testing framework for web applications.**
- ***Selenium automates browsers***
  - Automating web applications for testing purposes
  - Automating web-based administration tasks


- **Selenium WebDriver API**
  - create robust, browser-based regression automation suites and tests
- **Selenium IDE**
  - Chrome extension (capture/replay)
- **http://www.seleniumhq.org/**



# Selenium

- **Support for various browsers**
  - ChromeDriver, FirefoxDriver, InternetExplorerDriver, EdgeDriver SafariDriver, OperaDriver, …
- **Create more detailed reports of executed test cases**
  - Screenshots
  - Videos
- **Headless execution (CI Integration)**
  - Chrome headless mode
  - PhantomJS
- **Integrates well with test frameworks, e.g. JUnit**

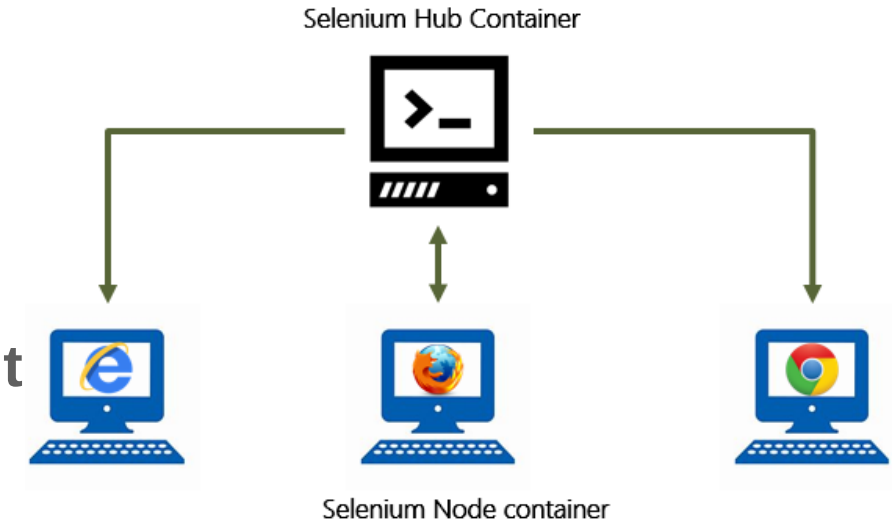
# Selenium Mobile Testing

- **Mobile applications (native, hybrid and web ) test automation framework**
- **Based on Selenium Web Driver**
- **Android**
  - Selendroid framework
  - Can be used on emulators and real devices.
  - http://selendroid.io/


- **iOS**
  - ios-driver
  - https://ios-driver.github.io/ios-driver/

# Selenium Grid

- **Parallel / distributed test execution on remote browser instances**
- **Run your tests on different machines against**
  - Multiple browsers
  - Multiple versions of browser
  - Browsers running on different operating systems
- **Docker support**
- **automate test execution**
- **automate test environment configuration**

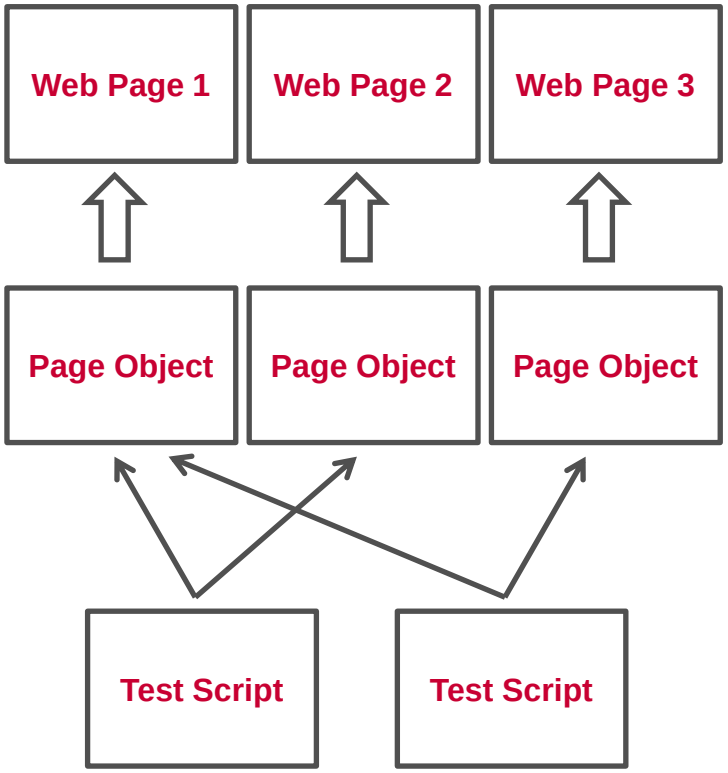


# UI Tests – Page Object Pattern

- **Object-oriented approach**
- **Abstract model of the UI**
- **Serves as decoupling layer between test code and web**
  - A page object wraps a single HTML page of a web application and encapsulates the interaction with that page

- **Advantages**
  - Reduces code duplication
  - Better modularization
  - Improves readability and robustness of tests
  - Improves the maintainability of tests

# UI Tests – Page Object Pattern




```java
public class LoginPage extends PageObject {

    @FindBy(id = "username")
    private WebElement usernameField;

    @FindBy(id = "passwort")
    private WebElement passwordField;

    @FindBy(id = "loginSubmit")
    private WebElement submitButton;

    public LoginPage(WebDriver driver) {
        super(driver);
    }

    public WelcomePage login(String name, String passw
        usernameField.sendKeys(name);
        passwordField.sendKeys(password);
        submitButton.click();

        return PageFactory.initElements(driver,
                WelcomePage.class);
    }
}
```

# UI Tests – Page Object Pattern

- **Best Practices:**
  - Page objects should not have any assertions
    - Assertion logic should happen in the test classes
  - Methods represent actions on the page
  - Navigation methods should return the page object of the subsequent page
  - Page objects should only include elements relevant for testing

# Content

# Behavior Driven Development (BDD)

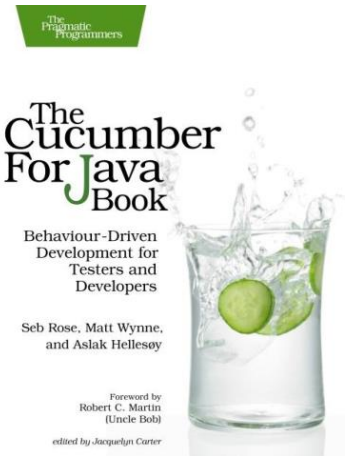
- Extension of **Test Driven Development (TDD)**
- Test first approach
- **Test scenarios** define the **expected behavior** of the application from a **user's perspective**

- **Acceptance criteria** are specified in a simple **domain specific language (DSL)**
- **DSL allows to**
  - Express the behavior
  - Define the expected outcome
  - Use examples to clarify requirements

# Cucumber Framework

- **Framework for automated BDD acceptance tests**
- **Support several software platforms (e.g. Java, Ruby, …)**
  - Every implementation provides the same overall functionality
- **Expected software behavior is written in Gherkin**
  - Given – When – Then Syntax
  - Stored in Feature files

- **Online Reference**
  - https://docs.cucumber.io



# Cucumber Framework

```
Feature <Description of the use case>

    Scenario: <Description of one scenario of that feature>
       Given <some precondition>
       And <another precondition>
       ...
       When <an action taken by the user>
       And <another action taken by the user>
       ...
       Then <expected result>
       And <another expected result>

    Sce
   ...
```

```
Feature As a user I want to login to the application

    Scenario: Successful login
       Given I am on the login page
       When I enter username "user" and password "abc"
       And I click the login button
       Then I am logged in at the application

    Scenario: Invalid login
       Given I am on the login page
       When I enter invalid username "u" and password "wrong"
       And I click the login button
       Then I stay on the login page and an error message is shown
```

# Cucumber Framework

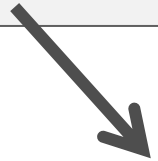
- **Data Driven**
  - Cucumber supports a data driven approach within feature files

```
Feature As a user I want to login to the application

    Scenario Outline: Successful login
       Given I am on the login page
       When I enter username "<user>" and password "<password>"
       And I click the login button
       Then I am logged in at the application

       Examples:
        | user            | password       |
        | admin           | abc            |
        | customer21      | mypass         |
        | maxmuster12     | maxm           |
```

# Cucumber Framework

- **Step Definitions**
  - Map each step in the feature file to code (Glue Code)
  - Define the actions that should be performed

```
Scenario: Successful login
  Given I am on the login page
  When I enter username "user" and password "abc"
  And I click the login button
  Then I am logged in at the application
```



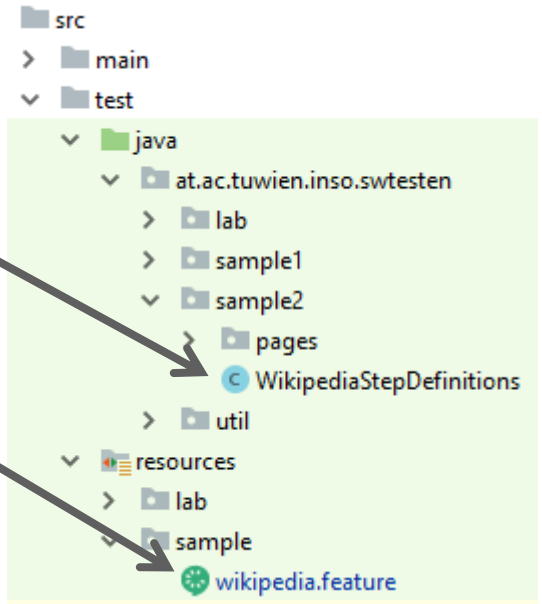
```
Given("I am on the login page", () -> {...});

When("I enter username {string} and password {string}",
        (String user, String password) -> {...});

When("I click the login button", () -> {...});

Then("I am logged in at the application", () -> {...});
```

Cucumber Expressions (Parameter Types, Regular Expressions…) Can be adapted manually

# Cucumber Framework

- **JUnit Integration**
  - Cucumber provides a JUnit Runner
  - Searches for feature files located in the defined path
  - Searches for step definitions located in the specified packages

```
@RunWith(Cucumber.class)
@CucumberOptions(
    glue = {"at.ac.tuwien.inso.swtesten.sample2"}
    features = {"src/test/resources/sample"},
)
public class CucumberJUnitTest {}
```



# References


- T. Grechenig et al; Softwaretechnik Mit Fallbeispielen aus realen Entwicklungsprojekten, 2010
- ISTQB Foundation Level Syllabus
- A. Spillner and T. Linz; Basiswissen Softwaretest. 5. Aufl. dpunkt Verlag, 2012.
- D. Graham et al; Foundations of Software Testing: ISTQB Certification. Cengage Learning EMEA, 2008.
- IEEE Std 610.12-1990 (R2002), IEEE Standard Glossary of Software Engineering Terminology IEEE, 1990.
- A. Spillner et al; Praxiswissen Softwaretest – Testmanagement: Aus- und Weiterbildung zum Certified Tester, 2014

# References


- M. Pezzé, M. Young, "Software testen und analysieren: Prozesse, Prinzipien und Techniken", Oldenbourg Verlag München, 2009
- T. Roßner u.a., „Basiswissen modellbasierter Test“, dpunkt Verlag, 2010