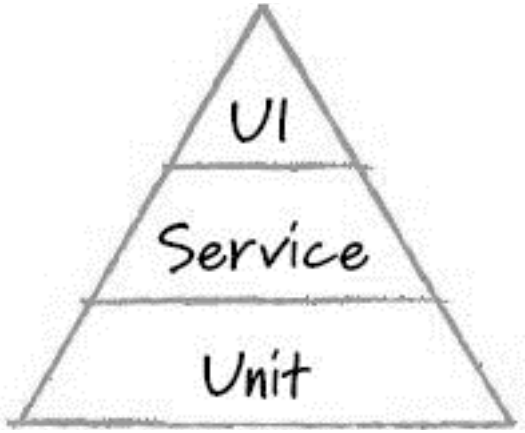
Hochschule Rosenheim
University of Applied Sciences

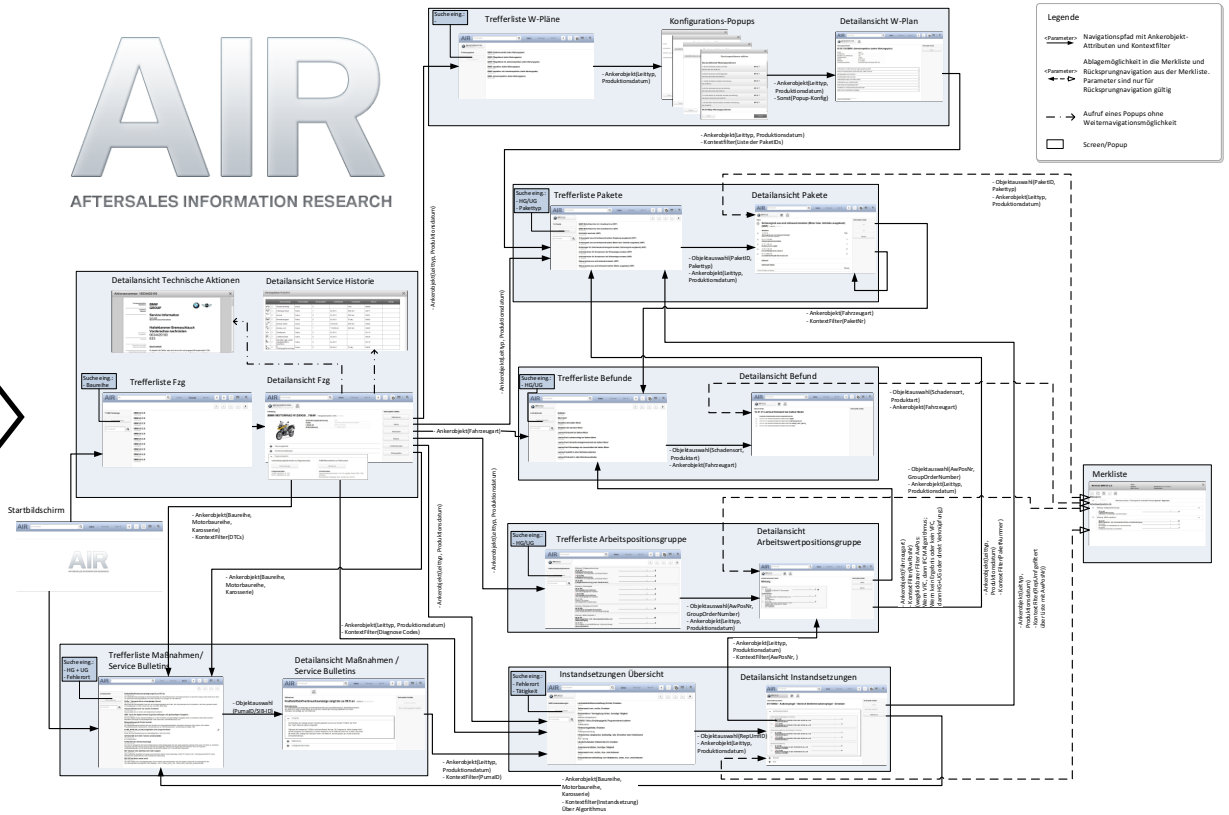
# Testgetriebene Entwicklung grafischer Benutzeroberflächen

## Vorgehen, Tools und Frameworks, Patterns, Automatisierung

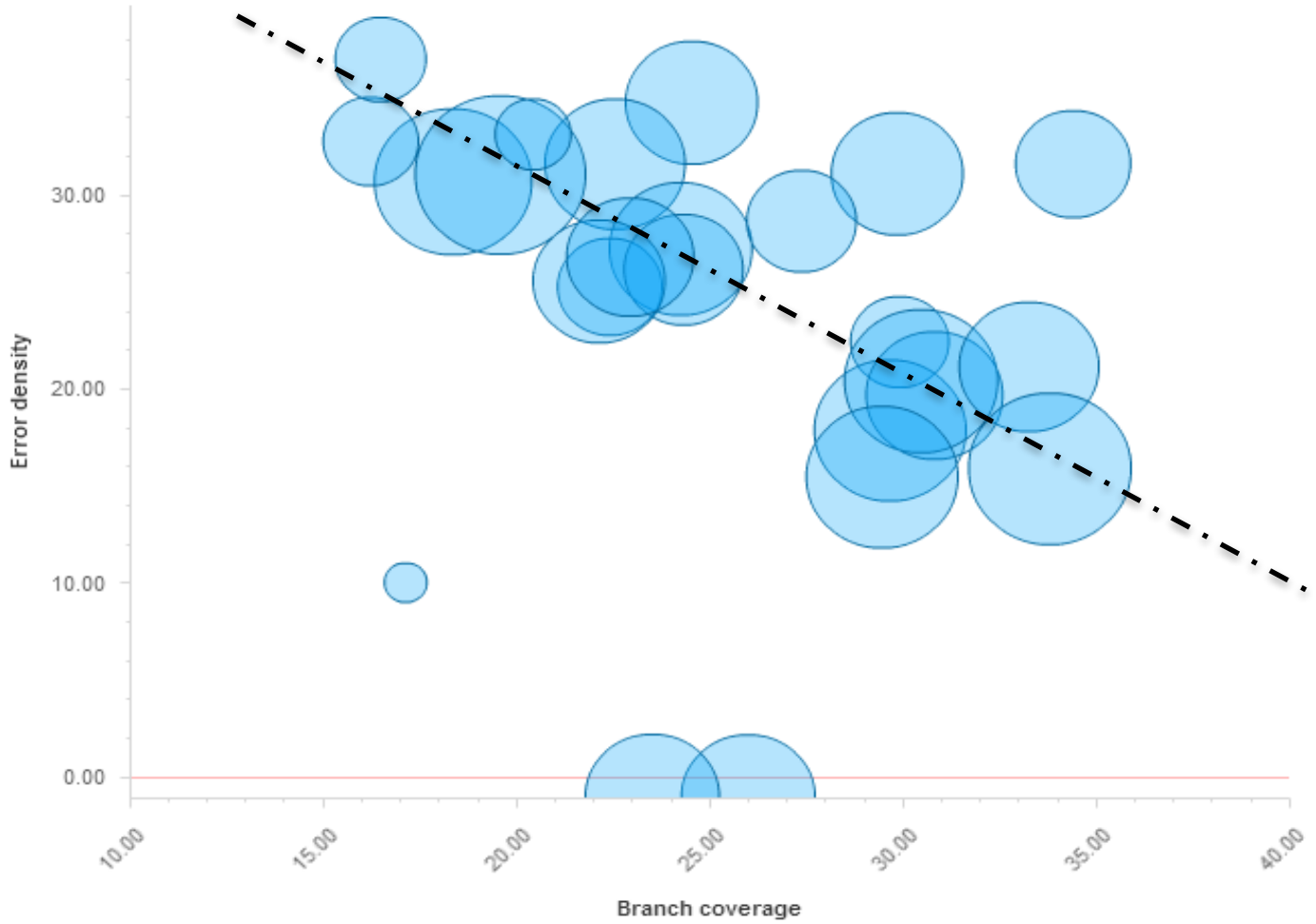


# Die Herausforderung: ein einfacher, durchgängiger, homogener Test-Ansatz trotz heterogener Clients und Technologien


- Vorgehensweise
- Frameworks & Tools
- Programmiermodell
- Spec Language
- Automatisierung
- Virtualisierung



# Testen zahlt sich aus: die statisch ermittelte Fehlerdichte sinkt mit steigender Testüberdeckung



# Gute Tests haben eine positive Wirkung auf die Qualität unserer Software und ihrer Architektur (1)

- Der Test als erster Nutzer unseres Codes liefert wertvollen Informationen
  - Macht mein Code was er soll?
  - Verhält sich eine Komponente wie erwartet? Auch im Fehlerfall?
  - Habe ich alle Randbedingungen berücksichtigt?
  - Ist eine Komponente gut entworfen? Lässt sie sich leicht benutzen?
- Der Test hilft uns unseren Code besser zu verstehen, zu formen und zu verbessern.
- Der Test ist die implizite Dokumentation des Codes.

# Gute Tests haben eine positive Wirkung auf die Qualität unserer Software und ihrer Architektur (2)

- Je höher die Testabdeckung desto höher ist die Zuversicht in unseren Code und das System als Ganzes
  - Oft gelten 50% Branch-Coverage als ausreichend.
  - Das reicht bei Weitem nicht!

Code coverage
**82,2%**
84,2% line coverage
73,6% branch coverage

- Der Software-Test als Garantie für die gute Wartbarkeit und Erweiterbarkeit unserer Software → Auch große Refactorings werden nicht zur Zitterpartie.
- Regressionen werden aufgedeckt und verhindert.
- Exploratives Testen beim Einsatz neuer Open Source Bausteine oder zur Umsetzung nicht klar definierter Features

# No broken windows! Tests sind kein Code zweiter Klasse!

Anwendungs-Code

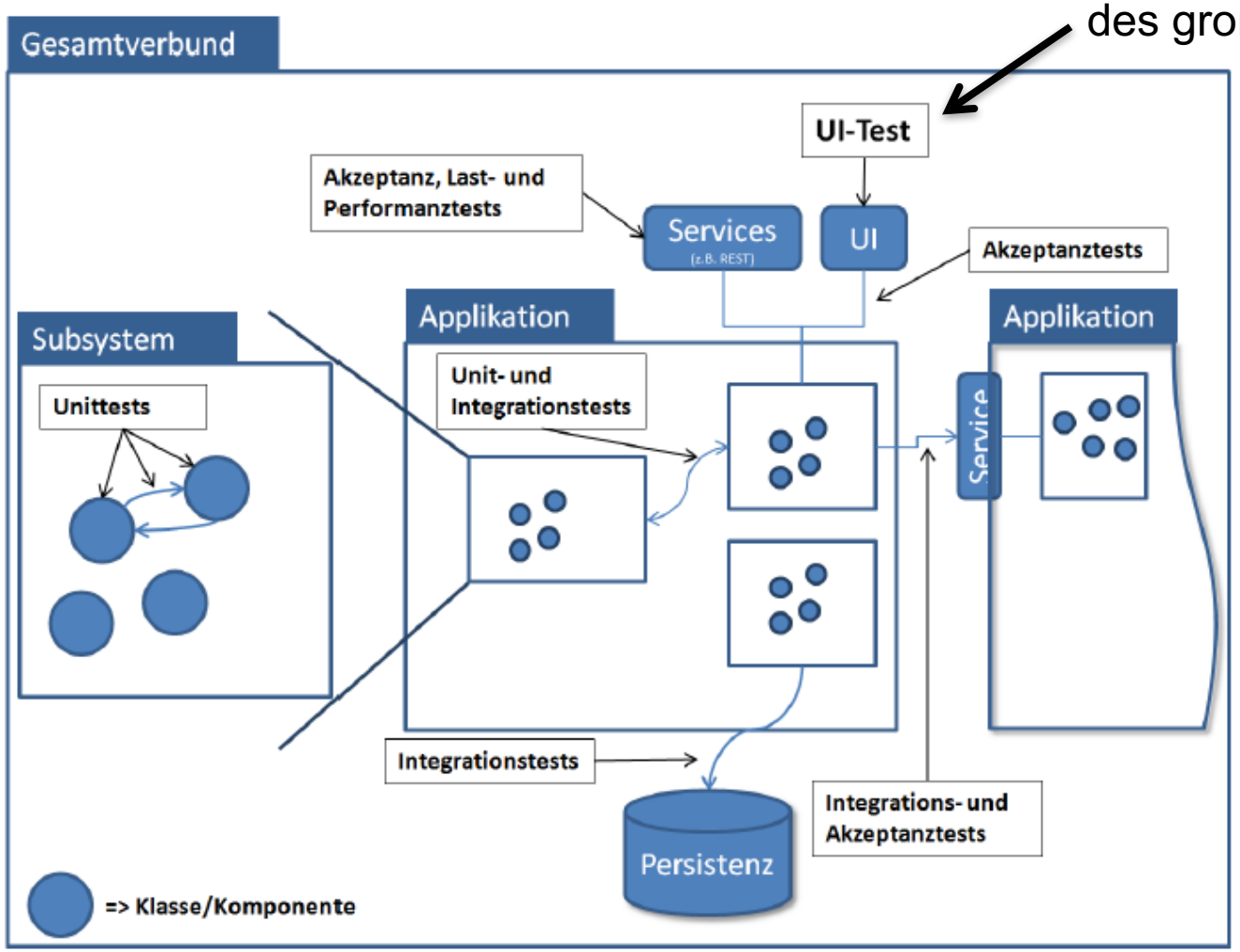
Test-Code

# Tests sind kein Code zweiter Klasse! Was genau heißt das?

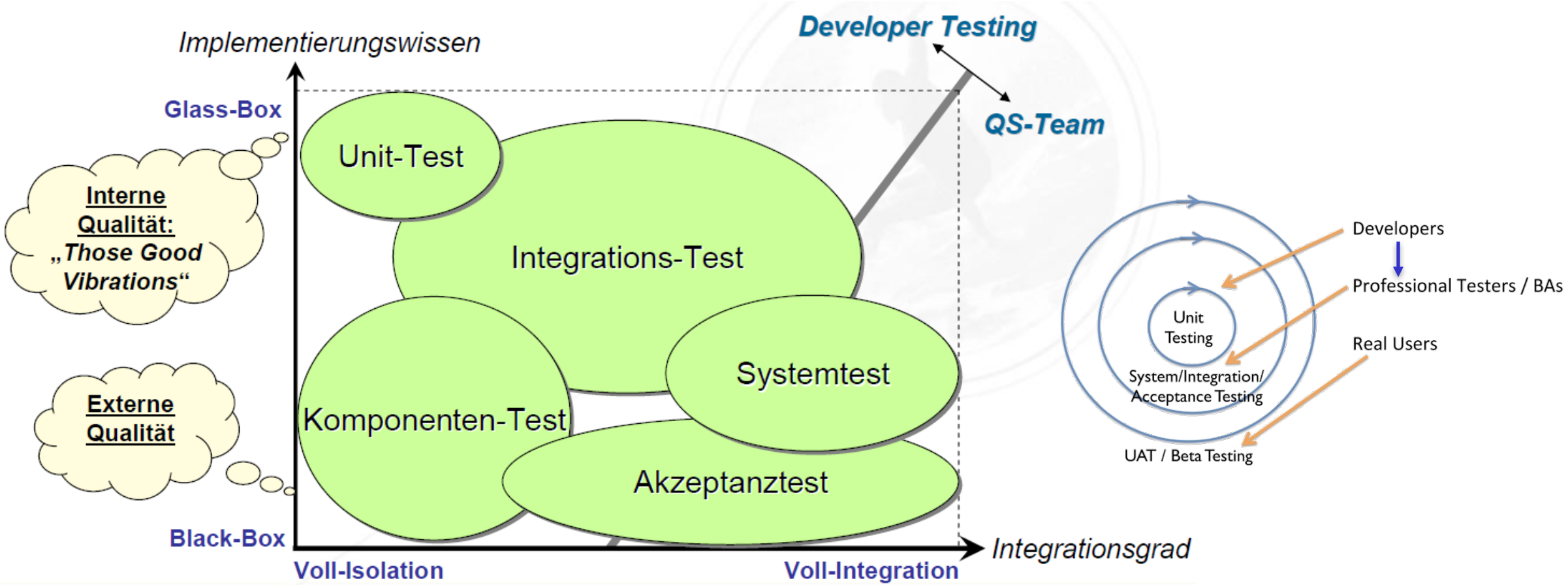
- Behandle den Test-Code mit der gleichen Sorgfalt wie den Anwendungs-Code!
  - Einhaltung der im Projekt geltenden Code-Konventionen (Header, Javadocs, Formatierung, …)
  - Eigentlich sollten fast die gleichen Regeln gelten. Naja, mit Ausnahmen!
  - Auch Sonar hat das bereits erkannt: https://jira.codehaus.org/browse/SONAR-1076
- Zudem gilt es Test-spezifische Constraints zu beachten
  - Klarer Aufbau: Setup - Call - Verify.
  - Mindestens ein Assert pro Test. Nur ein Assert pro Test?
  - Keine Abhängigkeiten zwischen Test

# Unterschiedliche Testautomatisierungsbereiche in einer Applikations-Landschaft

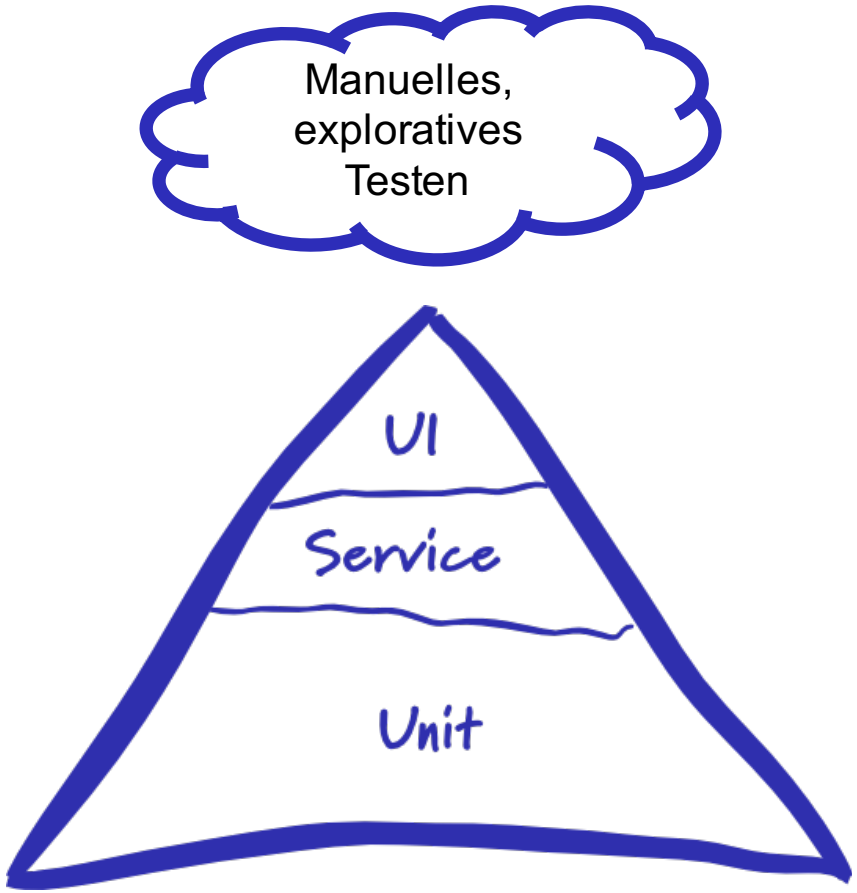
Nur ein kleiner Teil des großen Ganzen



# Testebenen und Ansätze unterscheiden sich in ihrem Wissen zur Implementierung und dem Grad ihrer Integration



# (Agile) Softwareentwicklung braucht eine Test-Strategie aus automatisierten und aber auch manuellen Tests.

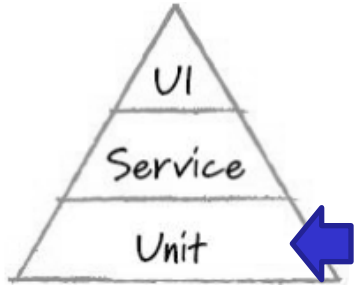



- Gute Testautomatisierung auf den unteren Ebenen mit allen Vorteilen.
- Eine Testautomatisierung auf den oberen Ebenen ist
  - sehr gut für Regressionstests geeignet und reduziert Aufwände für manuelles Testen,
  - entbindet nicht von der Pflicht manuell und explorativ zu testen
- Mythen (?) automatisierter UI-Tests:
  - Langsame Ausführungsgeschwindigkeit
  - Aufwändig in der Umsetzung
  - Fehleranfällig und schlechte Wartbarkeit

# Es gibt eine Vielzahl an Frameworks und Tools für das Testen in unterschiedliche Technologien

| # | Unit Tests | Integration Tests | Acceptance Tests | User Interface Tests |
|---|---|---|---|---|
| Java Entwicklung | JUnit<br>Hamcrest<br>Mockito | REST-assured<br>SoapUI | Cucumber-JVM | Selenium<br>Jemmy<br>MarvinFX |
| Groovy Entwicklung | JUnit<br>Spock | Spock | Spock<br>Cucumber-JVM | Geb Framework |
| JavaScript Entwicklung | Jasmine<br>Mocha | - | Cucumber + Capybara<br>Mocha | WebDriverJS<br>Protractor |
| .NET Entwicklung | MSTest<br>NSubstitute | - | SpecFlow | CodedUI |
| Android Entwicklung | JUnit<br>Robolectric | Robotium<br>AVD | Cucumber-JVM | Calabash<br>Robotium |
| iOS Entwicklung | OCTest / XCTest<br>OCMock<br>OCHamcrest | - | Frank Testing Framework<br>Cucumber | Frank Testing Framework<br>Calabash |

# Automatische Unit- und Komponenten-Tests als solide Basis für komplexe weiterführende Tests




- Die Entwicklung erfolgt Test getrieben (TDD), aber nicht unbedingt immer nach dem Test-First Ansatz.
- So gut wie jede Klasse hat einen Unit-Test, auch scheinbar einfache und banale POJOs
- Tests laufen bei jedem Build, vor jedem Commit und regelmäßig auf dem Build-Server
- Schon wenige Frameworks reichen aus:
  - Java Entwicklung: JUnit, TestNG, Hamcrest, Mockito
  - .NET Entwicklung: NUnit, MSTest, NSubstitute,

# Acht Eigenschaften guter Unit-Tests

1. **Korrekt**: Die Test müssen in sich fehlerfrei sein und zu den Anforderungen passen.
2. **Schnell**: Die Tests müssen schnell laufen, um häufig durchgeführt werden zu können.
3. **Abgeschlossen**: Die Tests müssen ein klares Ergebnis liefern und dürfen keine Interpretation benötigen, z.B.: durch lesen von Log-Meldungen.
4. **Isoliert**: Die Tests müssen unabhängig von anderen Tests durchführbar sein und dürfen andere Tests nicht beeinflussen.
5. **Sprechend**: Die Tests sollten ihre Absicht durch sprechende Benennung kundtun (wie etwa beim BDD). Das gilt auch für die Prüfung der Annahmen.
6. **Wartbar**: Die Tests sollten den Regeln für sauberen Code folgen und sich leicht an den veränderten Produktivcode anpassen lassen.
7. **Begrenzt**: Die Tests sollten jeweils nur einen kleinen Bereich des Testobjekts prüfen (und nicht etwa mehrere Methoden gleichzeitig).
8. **Einfach durchführbar**: Die Tests müssen so einfach wie möglich (am besten auf Knopfdruck) durch einen beliebigen Entwickler durchführbar sein.

# Testen von Komponenten mit deren Interaktionen in Isolation am Beispiel von Mockito, JUnit und Hamcrest

```
@RunWith(MockitoJUnitRunner.class)
public class GenericSearchImplTest {

    @Mock
    private GenericSearchDao genericSearchDao;
    @Mock
    private ResultConverter resultConverter;
    @InjectMocks
    private GenericSearchImpl genericSearch;

    private SearchResult<GenericSearchDto> dtoSearchResult;
    private SearchResult<GenericSolrEt> etSearchResult;

    @Before
    public void setUp() throws Exception {...}

    @Test
    public void testSearchByID() throws Exception {
        when(genericSearchDao.searchByID(eq("4711"), (SearchParams) any())).thenReturn(etSearchResult);
        when(resultConverter.convert(etSearchResult)).thenReturn(dtoSearchResult);
        SearchResult<GenericSearchDto> result = genericSearch.searchByID("4711", new SearchParams());
        assertThat(result, is(dtoSearchResult));
        verifyNoMoreInteractions(genericSearch, resultConverter);
    }

    @Test
    public void testSearch() throws Exception {...}

    @Test
    public void testSearchByInfoTypes() throws Exception {...}
}
```

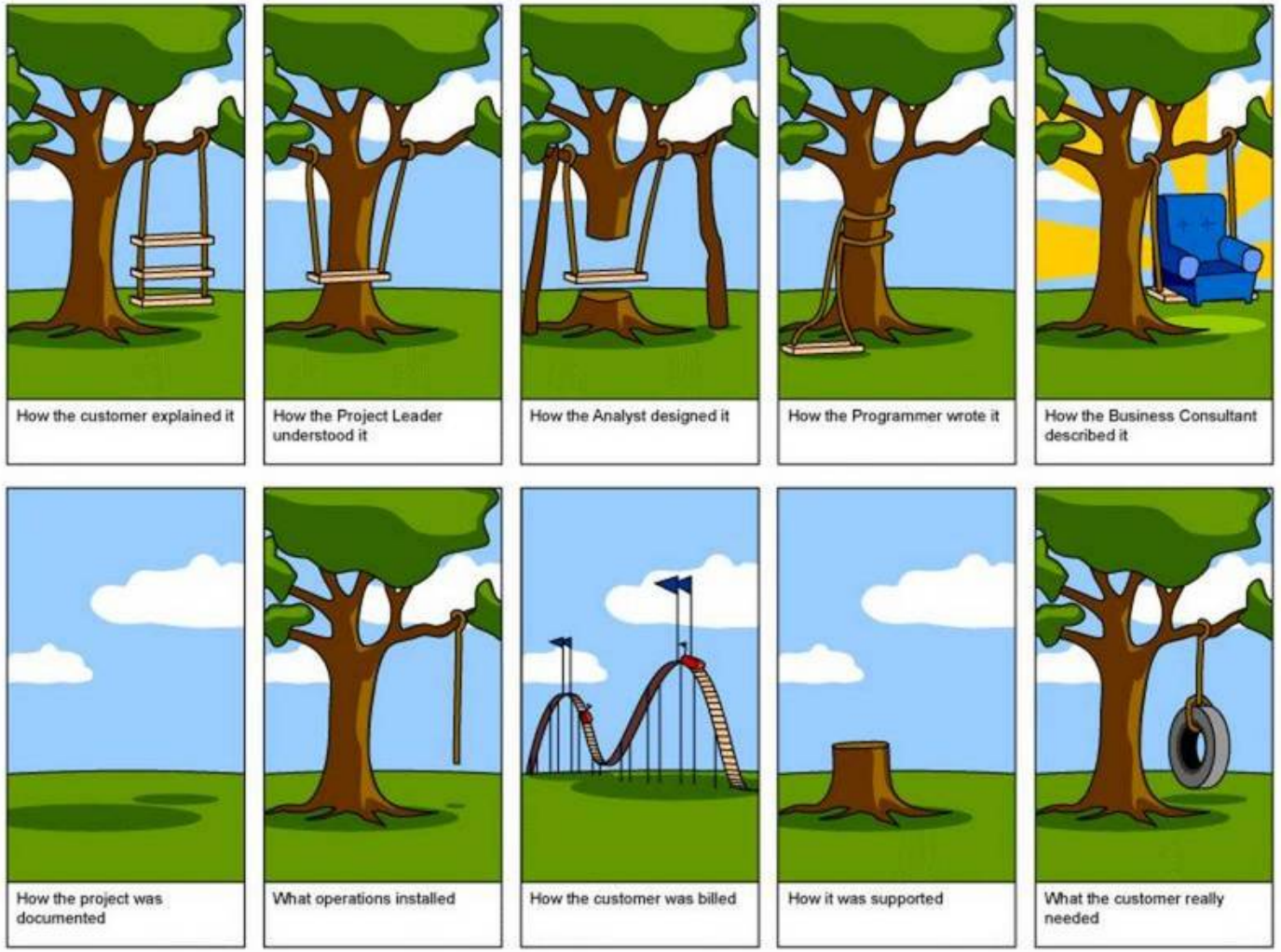
Erzeugen der Mocks und Injizieren in den Testling

Definition der Aufrufe und Parameter

Prüfung der Ergebnisse und Interaktionen

„Das habe ich mir aber anders vorgestellt.“

**Stakeholder**



# Steigende Komplexität und wachsende Funktionsumfänge erfordern fachliche Integrations- und Akzeptanztests

- Unit-Tests sind nicht geeignet um fachlich komplexe Sachverhalte und das korrekte Systemverhalten im Ganzen zu überprüfen.
- „Am I building the code right?“ → „Am I building the right code?“
- Wir benötigen bereits während der Entwicklung von neuen Features ein frühzeitiges und regelmäßiges Feedback
  - Werden alle benötigten Fremdsystem korrekte angebunden und integriert?
  - Sind die geforderten fachlichen Akzeptanzkriterien meiner Features erfüllt?
  - Funktionieren bereits realisierte Features nach Änderungen immer noch korrekt?
- **Lösung**: *Automatisierte Integrations- und Akzeptanztests*, als Teil des *Continuous Builds* in Kombination mit dem *Continuous Deployment* der gesamten Anwendung

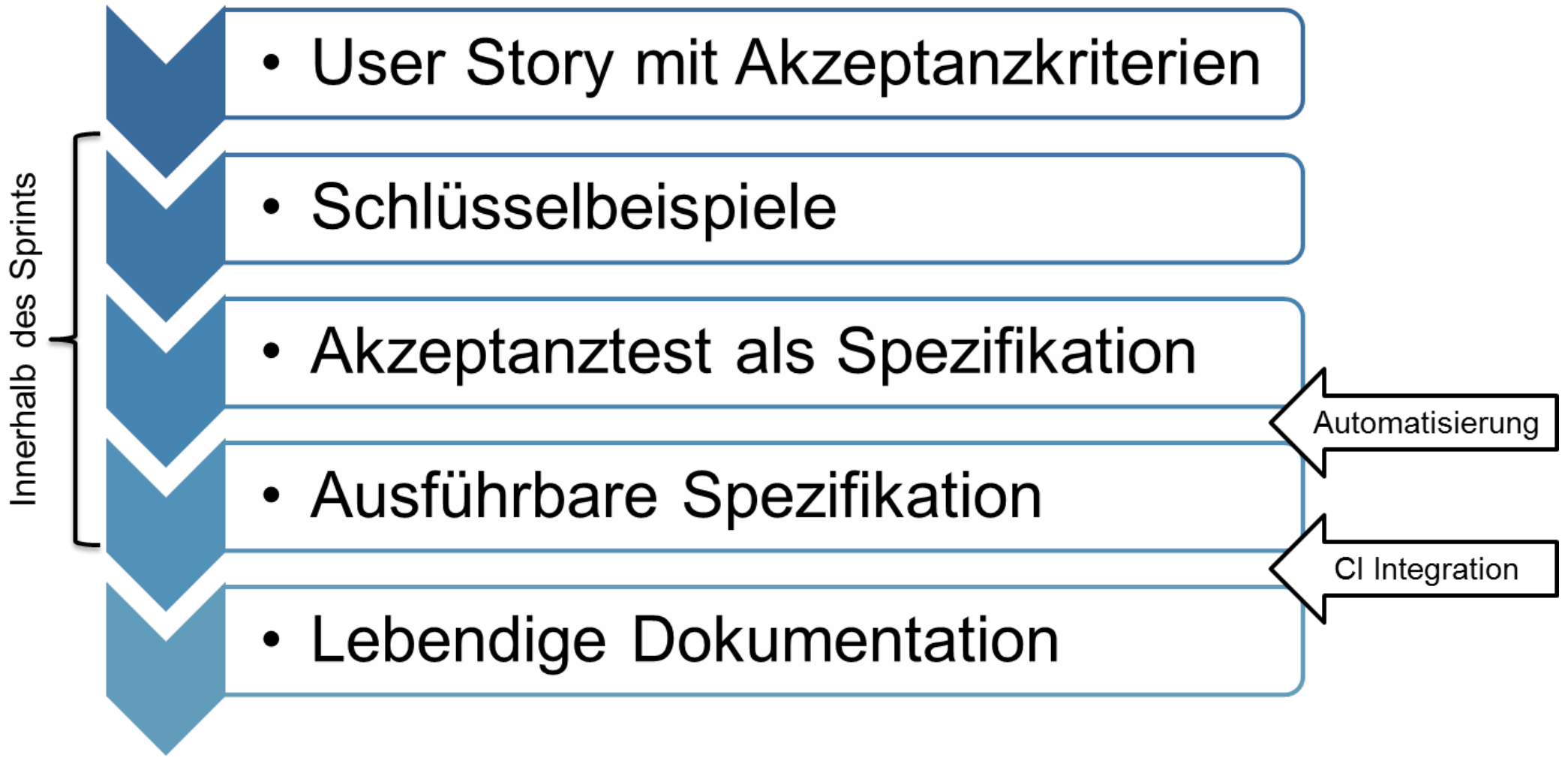
# Akzeptanztest getriebene Entwicklung (ATDD) auf einer Folie

- Agile Methode, die die Zusammenarbeit von Auftraggebern, Entwicklern und Testern unterstützt
  - Anforderungen und Akzeptanzkriterien werden von den Beteiligten gemeinsam erarbeitet und formuliert
  - Spezifikation erfolgt in natürlicher Sprache, ist einfach und für alle Projektbeteiligten verständlich
  - Kommunikation wird verbessert, man spricht gemeinsame Sprache
- Akzeptanztests überprüfen
  - die Systemfunktionalität aus Sicht der Anwender und Kunden,
  - funktionale und soweit möglich nicht-funktionale Eigenschaften
- Akzeptanztests sind automatisiert ausführbar

# „Wenn ich diesen Button klicke dann ...“

- Funktionen von Softwaresystemen werden häufig über das erwartete Verhalten der Benutzeroberfläche beschrieben
- Ein Akzeptanztest-getriebenes Vorgehen hilft die geforderten Features eindeutig zu spezifizieren, umzusetzen und automatisiert zu testen
- Dies gilt auch und ganz besonders für die Entwicklung von grafischen Benutzeroberflächen

# Das ATDD Phasenmodell



# Phase 1: User Story mit Akzeptanzkriterien

- User Stories sind eine kurze, einfache Beschreibung der geforderten Features (aka Anwendungsfall)
- Das in der User Story geforderte Verhalten beinhaltet oft bereits das erste Akzeptanzkriterium.
- Die Akzeptanzkriterien werden als Teil der Product Backlog Pflege vom PO / Team zu jeder User Story erarbeitet
- Mindestens ein Akzeptanzkriterium muss definiert sein bevor mit der Umsetzung begonnen wird (*Definition of Ready*)
- Die Definition kann bereits in einer späteren automatisierbaren Art formuliert werden, muss aber nicht.

# Phase 1: Beispiel

**User Story:**

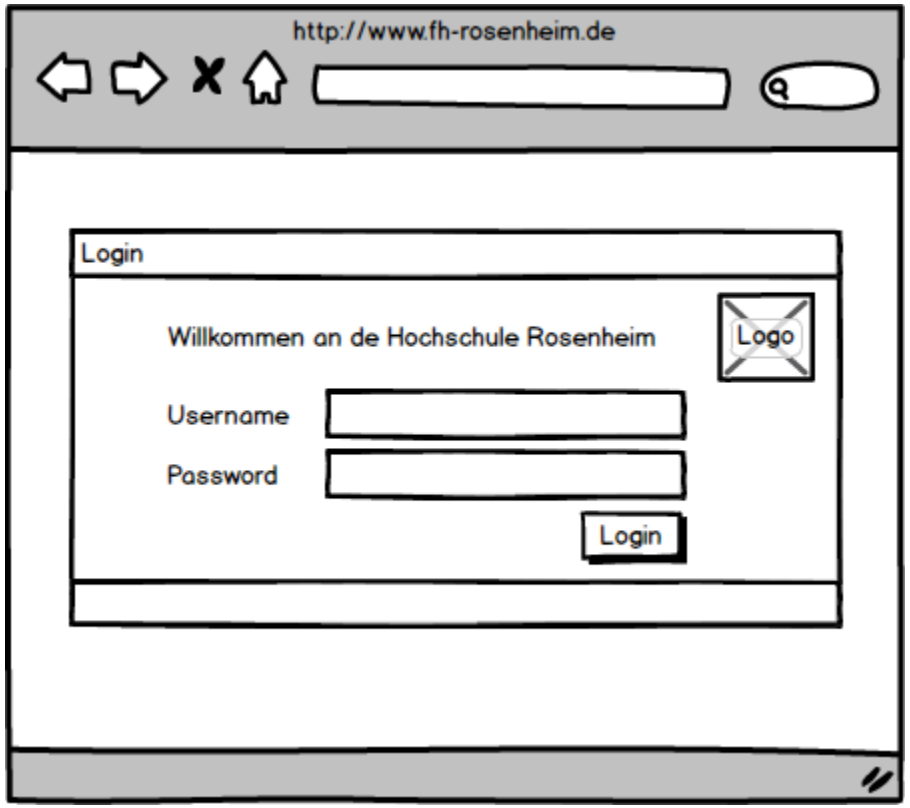
Für den Zugriff auf den geschützten Bereich muss sich der Benutzer am System mit einem Usernamen und Passwort anmelden.

**Akzeptanzkriterien:**

1. Der Benutzername und das Passwort dürfen nicht leer sein.
2. Der Benutzername muss eine E-Mail Adresse der FH-Rosenheim sein.
3. Das Passwort muss zwischen 6 und 16 Zeichen lang sein.
4. Bei fehlerhaftem Login soll dem Benutzer eine Fehlermeldung angezeigt werden.

# Für grafische Benutzeroberflächen werden zusätzlich zu den Akzeptanzkriterien auch Mockups oder Screen-Dummys erstellt

- Benutzeroberflächen als Mockup bzw. lauffähiger Screen-Dummy
- Beinhaltet alle wichtigen Elemente und Layout-Details
- Vereinfacht die Diskussion mit den Stakeholdern und Designern
- Akzeptanzkriterien können Elemente im Mockup verwenden
- **Empfehlung**: Eine Confluence-Seite pro User Story (Mini Spec)



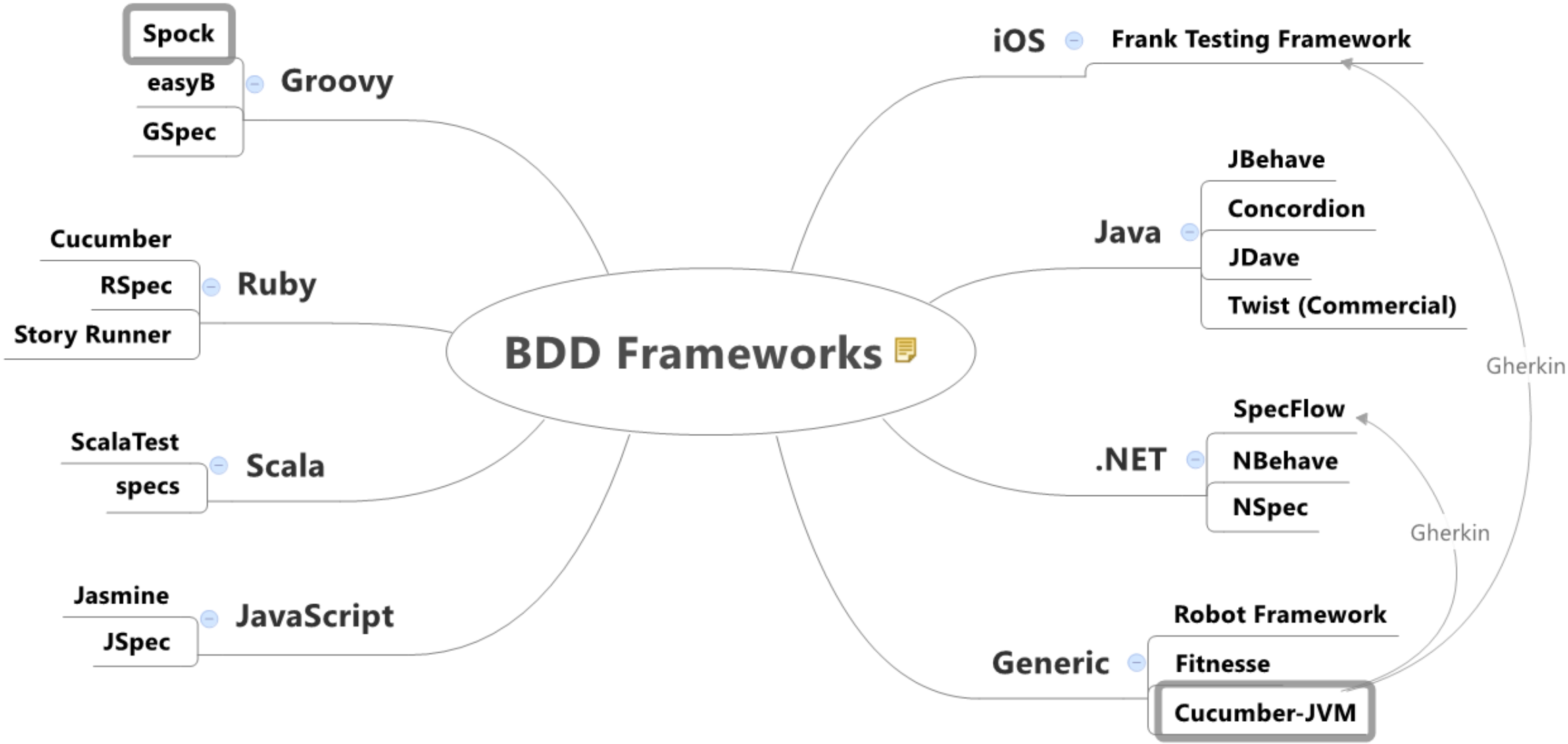
# Phase 2: Schlüsselbeispiele

- Definieren von möglichen Eingabe- und Ausgabedaten
- Grenzbereiche, gültige und ungültige Beispiele müssen abgedeckt sein
- Definition kann ausformuliert oder auch tabellarisch erfolgen

- **Beispiele**:
  - Eine gültige E-Mail Adresse ist mario-leander.reimer@hf-rosenheim.de
  - Eine ungültige E-Mail Adresse ist mario-leander.reimer@qaware.de
  - Ein minimal langes Passwort ist gültig, z.B. 1234a$
  - Ein maximal langes Passwort ist gültig, z.B. 1234567890123 4a$
  - Ein um 1 Zeichen zu kurzes Passwort ist ungültig, z.B. 123a$

# Phase 3: Akzeptanztest als Spezifikation

- Umsetzung der Akzeptanztests als automatisierbare und meist formalisierte Spezifikation
- Framework sollte in dieser Phase ausgewählt sein bzw. werden
- Spezifikation erfolgt oft im BDD-Style: Given / When / Then
- Spezifikation ist trotz der Formalisierung noch gut verständlich, kann aber maschinell verarbeitet werden
- Kann schon ausgeführt werden, schlägt aber fehl.




* Quelle: http://behaviordrivendevelopment.wikispaces.com/MoreTools

# Phase 3: Beispiele

```
# language: de
@Example
Funktionalität: Login-Maske

  Jeder Student muss sich für den Zugriff auf den geschützten Bereich am System anmelden.

  Szenario: Leerer Username und leeres Passwort
    Gegeben sei ein leeres Passwort und Username
    Dann ist der Login-Button deaktiviert

  Szenariogrundriss: Prüfung des korrekten Loginverhaltens
    Gegeben sei der Username "<USERNAME>"
    Und das Passwort "<PASSWORT>"
    Dann ist der Login-Button aktiviert
    Wenn der Login-Button angeklickt wird
    Dann ist der Login <RESULT>

  Beispiele: Erfolgreiche Login-Daten
    | USERNAME                  | PASSWORT          | RESULT |
    | m.l.reimer@fh-rosenheim.de | 1234a$           | OK     |
    | j.weigend@fh-rosenheim.de  | 12345678901234a$ | OK     |

  Beispiele: Falsche Login-Daten
    | USERNAME                  | PASSWORT | RESULT |
    | m.l.reimer@gaware.de      | abc      | NOK    |
    | j.weigend@fh-rosenheim.de |          | NOK    |
```

Szenarios beschreiben die Benutzeraktionen, Eingabedaten und das erwartete Verhalten

Verwendung unserer Schlüsselbeispiele

# Phase 4: Ausführbare Spezifikation

```
@Gegebensei("^ein leeres Passwort und Username$")
public void ein_leeres_Passwort_und_Username() throws Throwable {
    this.username = "";
    this.password = "";
}

@Gegebensei("^der Username \"([^\"]*)\"$")
public void der_Username(String username) throws Throwable {
    this.username = username;
}

@Wenn("^der Login-Button angeklickt wird$")
public void der_Login_Button_angeklickt_wird() throws Throwable {
    // interact with UI and click button
    throw new PendingException();
}

@Dann("^ist der Login \"([^\"]*)\"$")
public void ist_der_Login(String expectedLoginResult) throws Throwable {
    assertThat(actualLoginResult, is(expectedLoginResult));
}
```

- Initialisierung der Testdaten
- Einsatz von regulären Ausdrücken

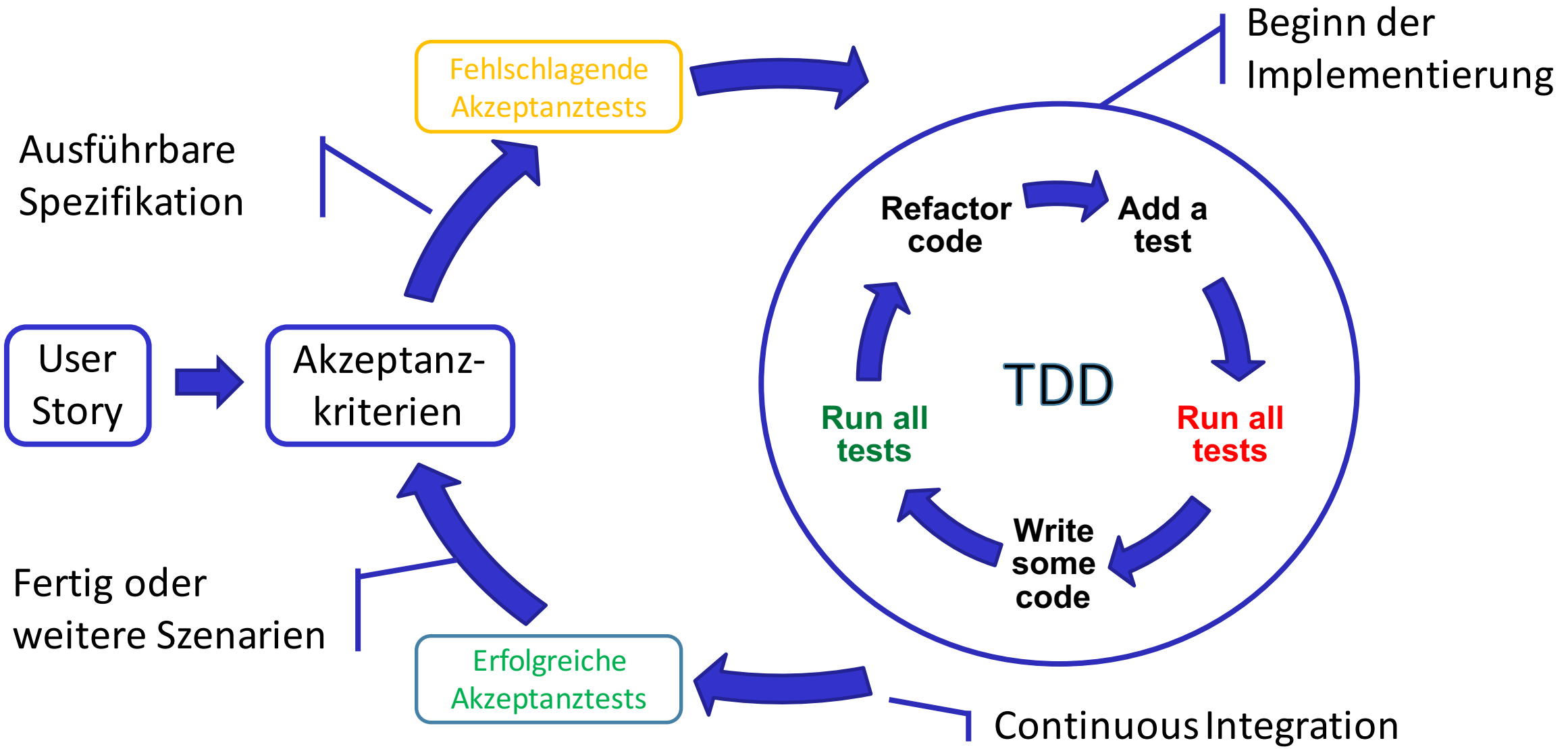
- Anfangs nur eine unvollständige Implementierung
- Nun startet der TDD Zyklus und die eigentliche Entwicklung

Prüfung der Akzeptanzkriterien erfolgt wie üblich mit Assertions, z.B. Hamcrest Matchern

# Phase 5: Lebendige Dokumentation

- Durch die CI Integration wird die ausführbare Spezifikation zur lebendigen Dokumentation (keine Schrankware)
  - Laufen alle neuen Akzeptanztests ohne Fehler
  - Laufen alle bisherigen Akzeptanztests noch ohne Fehler
- Vollständige, stets gültige Spezifikation des Systems. Wächst in jedem Sprint mit jeder neuen User Story.
- Basis für Systemspezifikation / Systemhandbuch (→ Relish)
  - Zielpersonen: Tester, Business Analysts, Support
  - Systemstruktur und fachliche Bereiche finden sich in der Strukturierung der Akzeptanztests / Feature-Dateien wieder
  - Fachbegriffe finden sich im Glossar wieder, können extrahiert werden

# ATDD ist kein Ersatz von TDD, sondern eine Ergänzung



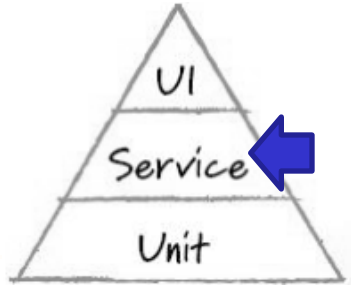
# ATDD ist keine Silver Bullet

- TDD wird weiterhin zwingend benötigt.
- Nicht alle Akzeptanzkriterien lassen sich automatisiert testen, z.B. die Usability oder Look&Feel eines Systems.
- ATDD muss um andere Testansätze ergänzt werden.
  - Last und Performance-Tests
  - Penetration Tests
  - User Acceptance Tests
- Akzeptanztests ersetzen nicht kluge und aufmerksame Tester. Exploratives Testen ist weiterhin sinnvoll und nötig.

# ATDD und seine Fallstricke

- Akzeptanztests dürfen genau wie alle anderen Tests nicht zum Wartungsrisiko werden
  - Duplikation von Step Definition muss vermieden werden.
  - Instabile und fragile Tests können schnell nerven (Leaky Szenarios).
  - Zu viele beiläufige Details blähen Tests unnötig auf (Imperative Steps).
  - Ausführungsgeschwindigkeit sinkt je größer das System und mit steigender Anzahl an Akzeptanztests
  - Abhängigkeiten zwischen Tests durch Test Fixtures sind problematisch.
- Die Formulierung guter, stabiler Akzeptanztests ist schwierig
  - Einheitliches Vokabular und Grammatik wird benötigt.
  - Konstantes Refactoring der Akzeptanztests ist unabdingbar.

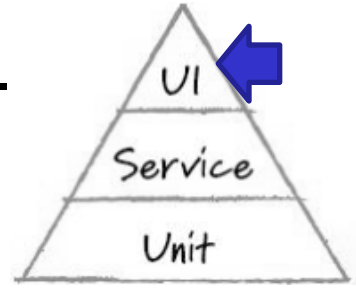
# Technische Akzeptanztests für REST API am Beispiel von REST-assured und Cucumber




```
Szenario: Lösche ein vorhandenes Backup
  Angenommen ich habe ein leeres Adressbuch
  Und ich mache 1 Backups
  Wenn ich einen JSON Request per DELETE nach "/contacts/v1/backups/{{lastResponse.backup.id}}" schicke
  Dann muss der HTTP-Code der Antwort 200 sein
  Und der Inhalt der Antwort muss dieser JSON Struktur entsprechen
  """
      {"value":true}
  """
```

```
@Angenommen("^ich habe ein leeres Adressbuch$")
public void ich_habe_ein_leeres_Adressbuch() throws Throwable {
    // delete
    restSupport.getRequestSpecification().delete("/contacts/v1/addressbook").
            then().assertThat().statusCode(anyOf(is(200), is(400)));

    // create user
    restSupport.getRequestSpecification().contentType(ContentType.JSON).header("Accept", ContentType.JSON.getAcceptHeader())
            .body("{\"last\":\"ContextConfiguration\",\"first\":\"Cucumber\"}".getBytes("utf-8"))
            .put("/contacts/v1/").then().assertThat().statusCode(200);

    // clear address book
    restSupport.getRequestSpecification().delete("/contacts/v1/all").then().assertThat().statusCode(200);
}
```

# Die Akzeptanztest getriebene Entwicklung von Benutzeroberflächen funktioniert Technologie übergreifend




| # | Acceptance Tests | User Interface Tests |
|---|---|---|
| Java Entwicklung | Cucumber-JVM | Selenium<br>Jemmy<br>MarvinFX |
| Groovy Entwicklung | Spock<br>Cucumber-JVM | Geb Framework |
| JavaScript Entwicklung | Cucumber + Capybara<br>Mocha | WebDriverJS<br>Protractor |
| .NET Entwicklung | SpecFlow | CodedUI |
| Android Entwicklung | Cucumber-JVM | Calabash<br>Robotium |
| iOS Entwicklung | Frank Testing Framework<br>Cucumber | Frank Testing Framework<br>Calabash |

Technologie-spezifische Frameworks für die UI Automatisierung

Unterstützt das *PageObject Pattern* bereits out-of-the-box

Akzeptanztest-getriebene Entwicklung von Benutzeroberflächen durch Kombination beider Bereiche

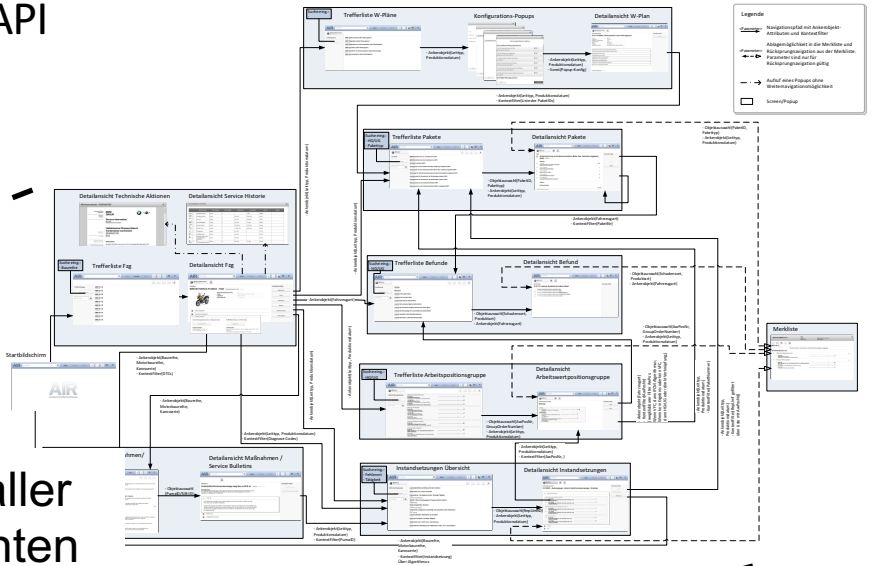
Gherkin als gemeinsame Spec-Language aller Akzeptanztests

# Page Object API anstatt Record & Replay stellen die optimale Wartbarkeit der Test sicher

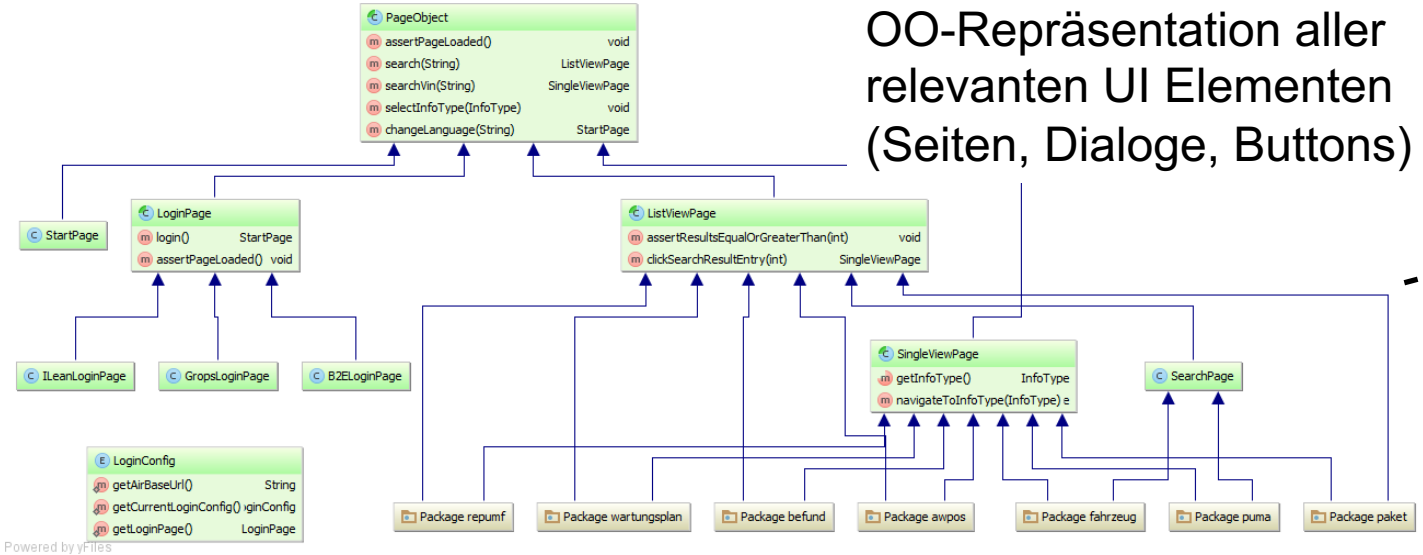
```
public void selectInfoType(final InfoType infoType) {
    if (InfoType.PUMA_MASSNAHME == infoType) {
        getWebDriver().findElement(MORE_BUTTON).click();
    }
    getWebDriver().findElement(FILTER_INFO_TYPE_ID.get(infoType)).click();
    waitFor(ONE_SECOND);
    currentFilterSelection = infoType;
}
```

Page Objects sind das Application Level API für die UI Tests

Interaktion mit technischen APIs wird gekapselt



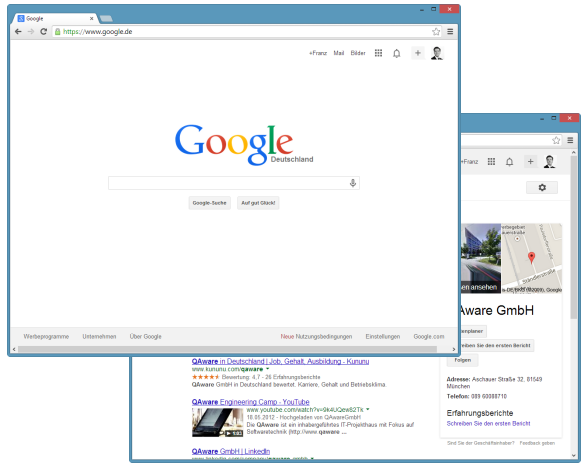
OO-Repräsentation aller relevanten UI Elementen (Seiten, Dialoge, Buttons)

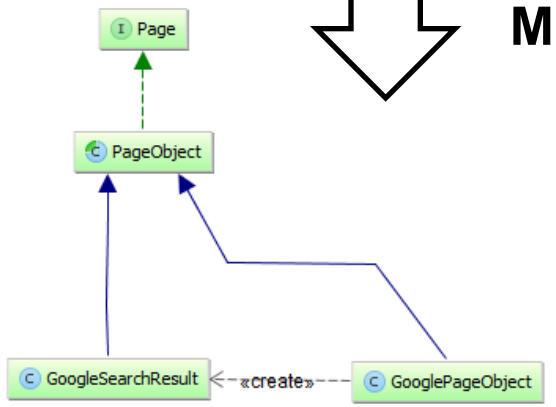


# Zusammenspiel von Akzeptanztest, Test und Glue-Code, Page Objects und der Oberfläche

```
# language: de
Funktionalität: Google Suche

  Szenario: Suche nach QAware
    Angenommen ich bin auf der Google Startseite
    Wenn ich nach "qaware" suche
    Dann sehe ich die Ergebnisseite
    Und der erste Link soll "QAware" enthalten
```

```
@Und("^ich nach \"([^\"]*)\" suche$")
public void ich_den_Text_eingebe(String searchValue) throws Throwable {
    googlePage = new GooglePageObject();
    googleSearchResult = googlePage.searchText(searchValue);
}

@Dann("^der erste Link soll \"([^\"]*)\" enthalten$")
public void ist_das_Suchergebnis_leer(String searchValue) throws Throwable {
    assertThat(googleSearchResult.getResultHeadlines(), is(empty()));
    assertThat(googleSearchResult.getResultHeadlines().get(0), containsString(searchValue));
}

@Dann("^sehe ich die Ergebnisseite$")
public void muss_die_Todo_Liste_ein_Todo_mit_dem_Text_enthalten() throws Throwable {
    assertThat(googleSearchResult, is(notNullValue()));
    googleSearchResult.assertPageLoaded();
}
```




**Magic happens here!**

# Beispiel für Java PageObject mit Selenium WebDriver zur Automatisierung der Google Suche

```
public class GooglePageObject extends PageObject {

    private static final By GOOGLE_SEARCH_INPUT = By.id("gbqfq");

    public GoogleSearchResult searchText(String arg1) {
        WebElement inputField = getWebDriver().findElement(GOOGLE_SEARCH_INPUT);
        inputField.sendKeys(arg1);
        return new GoogleSearchResult();
    }
}
```

Findet WebElements über deren ID
`<input id=„gbqfq“></input>`

Keyboard Interaktion und Navigation zu neuer Seite

```
public class GoogleSearchResult extends PageObject {

    private static final By SEARCHRESULT_HEADLINES =
            // XPath Expression for all H3 headlines
            By.xpath("//div[@id='ires']/ol/li/div/h3");

    public List<String> getResultHeadlines() {
        List<String> headlines = new ArrayList<~>();
        List<WebElement> resultElements = getWebDriver().findElements(SEARCHRESULT_HEADLINES);
        for (WebElement element : resultElements) {
            headlines.add(element.getText());
        }
        return headlines;
    }
}
```

Findet WebElements mittels XPath im aktuellen HTML Document

# Selenium WebDriver API zur Browser Automatisierung

- Technisches API zur Steuerung und Interaktion mit Browsern und für den Zugriff auf die angezeigten HTML Inhalte
- Unterstützt alles gängigen Browser
  - Firefox Unterstützung out-of-the-box
  - Anderen Browser über zusätzliche Executables
- Unterstützt auch Headless Browser, wie z.B. PhantomJS oder HtmlUnit
- Es gibt WebDriver API Bindings in so gut wie allen Sprachen

```
DesiredCapabilities caps = DesiredCapabilities.firefox();
WebDriver webDriver = new FirefoxDriver(caps);
webDriver.manage().timeouts().pageLoadTimeout(30, TimeUnit.SECONDS);
webDriver.manage().window().setSize(new Dimension(1024, 768));
```

# Very Groovy Browser Automation mit dem Geb Framework

- Groovy basiertes Framework mit DSL für UI Automatisierung
- Cross Browser Automation per Selenium WebDriver
- jQuery-like API zur Content Navigation
- Native Unterstützung des Page Object Patterns
- Unterstützung für asynchrones Laden von Seiten und Inhalten
- Gute Test-Framework Integration: JUnit, Spock, Cucumber, …
- Einfache Integration in Build-Tools: Maven, Gradle, …

# Einfaches Beispiel für Geb DSL

```
Browser.drive {
    to LoginPage
    assert at(LoginPage)
    loginForm.with {
        username = "admin"
        password = "password"
    }
    loginButton.click()
    assert at(AdminPage)
}
```

- Navigiert zur URL vom Page Object
- Prüft erfolgreiche Navigation

Interaktion mit den Elementen des aktuellen PageObjects

Klick und Navigation auf Folgeseite

# Umsetzung des Page Object Pattern in Geb

```
class LoginPage extends Page {
  static url = "http://www.fh-rosenheim.de/protected"
  static at = { heading.text() == "Please Login" }
  static content = {
    heading { $("h1") }
    loginForm { $("form.login") }
    loginButton(to: AdminPage) { $("input", type:"submit")}
  }
}
```

Navigation zur URL mit to LoginPage

Prüfung mit at LoginPage

Definition der Seiten Elemente erfolgt über jQuery-like Closures

# Geb bietet gute Integration in BDD und ATDD Test-Frameworks

Spock + Geb DSL

```
@Stepwise
class SpockSpeck extends GebSpec {
    def "Go to login"() {
        when: to LoginPage
        then: waitFor { at LoginPage }
    }

    def „Invalid login"() {
        when: loginButton.click()
        then: waitFor { at LoginPage }
    }
}
```

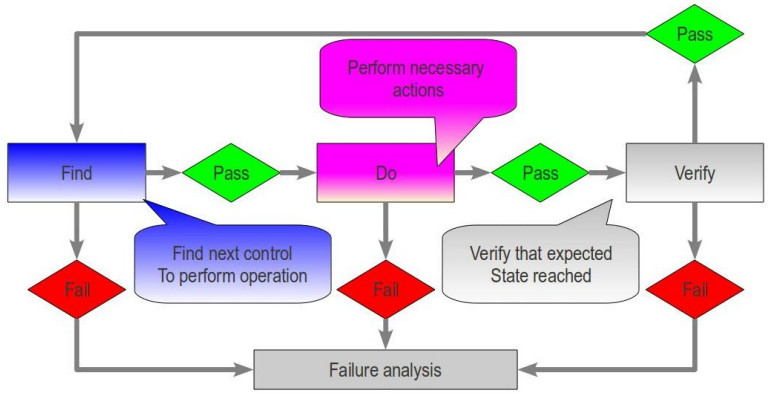
VS.

Cucumber + Geb DSL

```
Given(~'^I go to the login page$') { ->
    to LoginPage
    waitFor { at LoginPage }
}

When(~'^I perform an invalid login') { ->
    page.loginButton.click()
}

Then(~'^I stay on the login page$') { ->
    waitFor { at LoginPage }
}
```

# JavaFX Automatisierung mit JemmyFX v3

- https://jemmy.java.net/
- Wird vom JavaFX Build verwendet um die FX Controls zu testen.
- Bietet API für
  - den Lookup von FX Controls im Scene Graph
  - die Interaktion mit FX Controls mittels Wrapper
- Keine offiziellen, aktuellen Downloads. Stattdessen selber bauen.
- Für weitere JavaFX Test Frameworks siehe aktuelles Java Magazin 6.14




```
public class LoginDemoTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        LoginSceneObject.executeNoBlock();
    }

    @Test
    public void testLoginOK() {
        LoginSceneObject loginScene = new LoginSceneObject();
        loginScene.setUsername("mlr");
        loginScene.setPassword("123");

        String result = loginScene.doLogin();
        assertThat(result, is("Login erfolgreich."));
    }

    @Test
    public void testLoginNOK() {...}
}
```

# JavaFX Automatisierung und Scene Objects mit JemmyFX v3

```
public class LoginSceneObject {

    private final SceneDock sceneDock;

    public LoginSceneObject() { sceneDock = new SceneDock(); }

    public void setUsername(final String username) {
        TextInputControlDock userInput = new TextInputControlDock(sceneDock.asParent(), "userInput");
        userInput.clear();
        userInput.type(username);
    }

    public void setPassword(final String password) {
        TextInputControlDock pwdInput = new TextInputControlDock(sceneDock.asParent(), "pwdInput");
        pwdInput.clear();
        pwdInput.type(password);
    }

    public String doLogin() {
        ControlDock loginButton = new ControlDock(sceneDock.asParent(), "loginButton");
        loginButton.mouse().click(1);

        LabeledDock resultLabel = new LabeledDock(sceneDock.asParent(), "resultLabel");
        return resultLabel.getText();
    }

    public static void executeNoBlock() {
        AppExecutor.executeNoBlock(LoginDemo.class);
    }
}
```

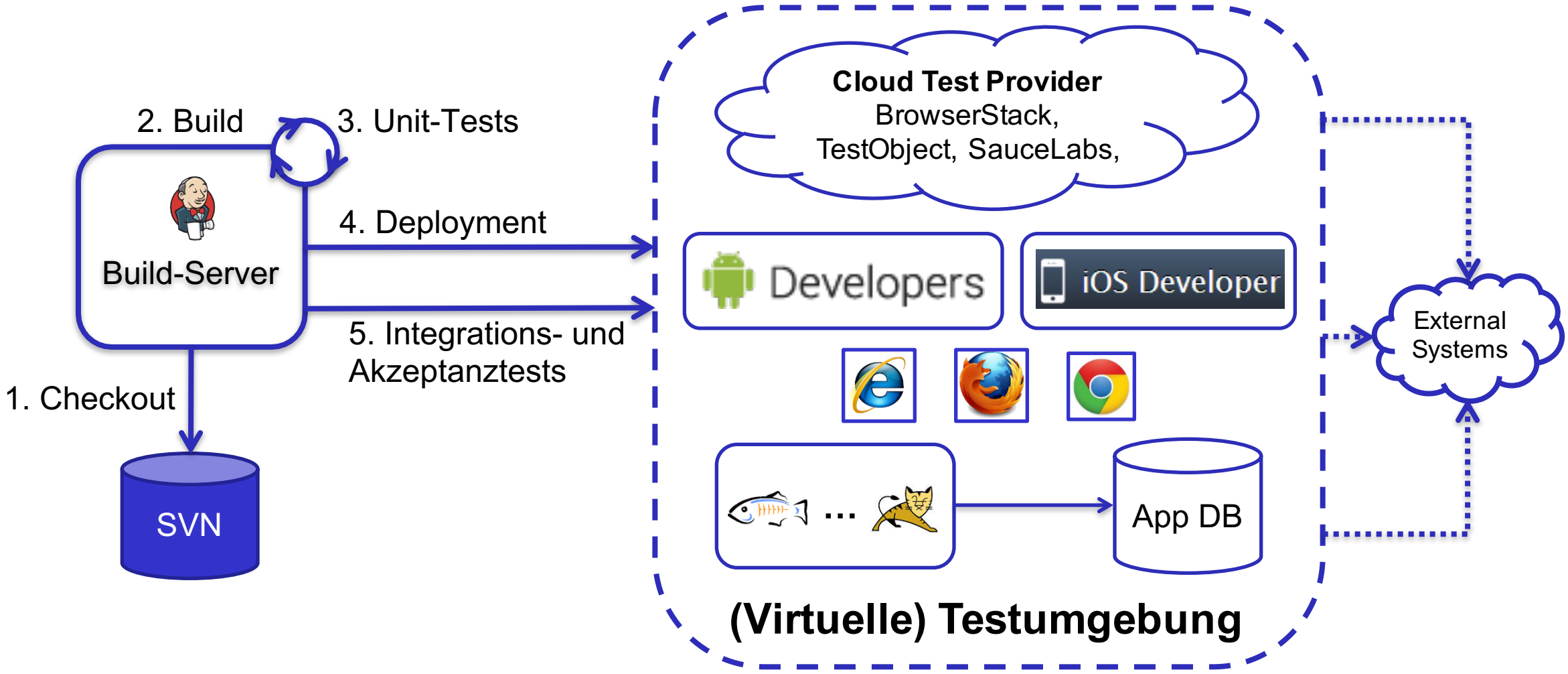
Wrap Scene als Root Node

Control Lookup, Wrap, Keyboard Interaktion

Control Lookup, Wrap, Mouse Interaktion

Ausführen der JavaFX App im Hintergrund

# Ablauf eines Continuous Builds in Kombination mit dem Continuous Deployment der Anwendung



# GUI Testing Using Computer Vision

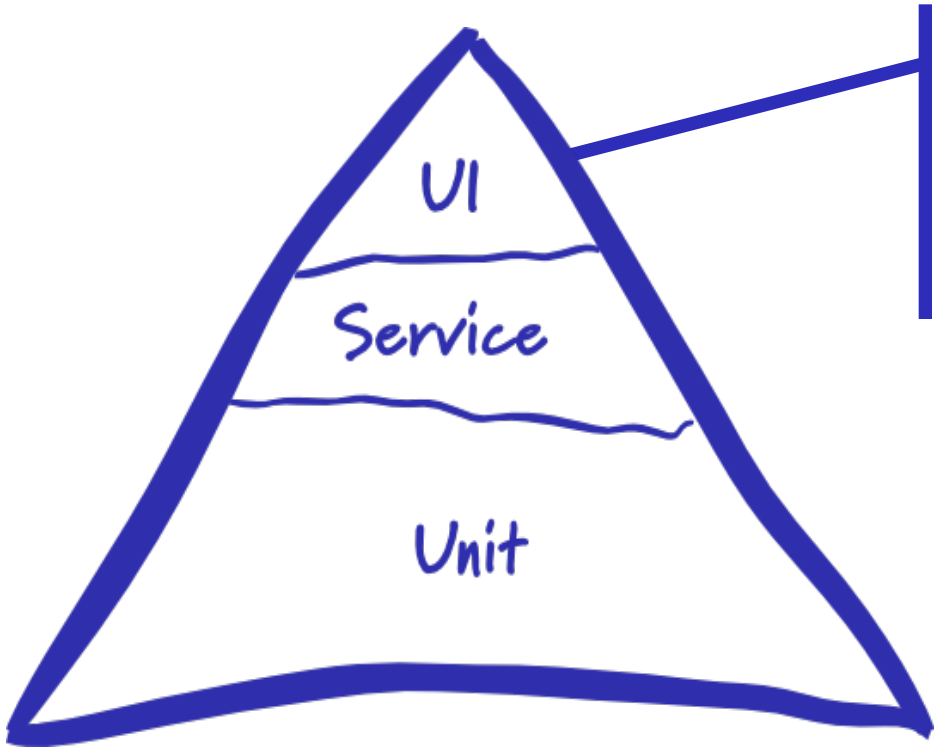
- GUI Automatisierung erfolgt über Scripting (kann oft aufgezeichnet werden)
- Vergleich von Referenz-Screenshot mit einem „frischen“ Screenshot
- Stabilität solcher Tests kann problematisch sein
  - Animationen müssen deaktiviert werden
  - Möglichst statische Inhalte und Daten anzeigen
  - Test-Regionen sollten eingeschränkt werden


http://www.sikuli.org/

http://de.slideshare.net/vgod/practical-sikuli-using-screenshots-for-gui-automation-and-testing

http://groups.csail.mit.edu/uid/projects/sikuli/sikuli-chi2010.pdf

# Das ganzheitliche Testen einer Anwendung mit grafischer Benutzeroberfläche ist aufwändig aber nicht schwer.




**Mythen automatisierter UI-Tests:**

- ~~*Langsame Ausführungsgeschwindigkeit*~~
- ~~Aufwändig in der Umsetzung~~
- ~~Fehleranfällig und schlechte Wartbarkeit~~