GEBIT Solutions

Technologie beherrschen ▶ Prozesse gestalten ▶ Lösungen schaffen ▶▶

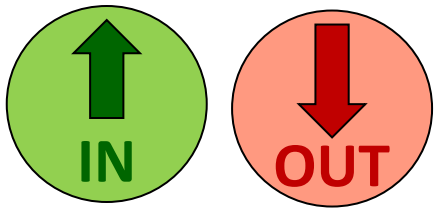
Test von Web-Anwendungen: Auf das Werkzeug kommt es an

Dr.-Ing. Dehla Sokenou, Ronald Brill

GEBIT Solutions GmbH

Rich-Client ist out - Webanwendung ist in

- Wechsel der Basis – Betriebssystem wird durch Browser ersetzt
  - wenn man es zu Ende denkt, landet man bei Electron
- im Gegensatz zum Betriebssystem war der Browser nicht als Plattform konzipiert – deshalb rasante Weiterentwicklung
- Versprechen der Plattformunabhängigkeit
  - dafür wieder Device abhängig



- Interaktionsdesign ist abhängig von der Größe des Bildschirms
  - Responsive Design - eine Applikation, die sich auf verschieden Geräten anders verhält

Was ändert sich NICHT

- Auch Rich Client Technologien bzw. Anwendungsframeworks waren/sind nicht testfreundlich
  - Testen bleibt das Stiefkind
- Testdaten sind das eigentliche Problem
  - aus unserer Sicht gibt es keine allgemeine Lösung
- Automatisierte Test bekommt man nicht geschenkt

Was ändert sich NICHT

- Überlegtes, vernünftiges Herangehen ist der Schlüssel
  - Aufteilung zwischen Automatisierung und manuellem Test
  - Testplanung
  - Testpflege
  - Design for Testability
    - e.g. think of **html-aria**

- Der richtige Werkzeugkasten
  - Testtools, CI Umgebung

Was ändert sich

Neue Herausforderungen für automatische Tests

- Dynamik & Asynchronität
  - Ajax
  - Single Page – der Inhalt des Browserfensters wird verändert, ohne die ganze Seite neu zu laden
  - asynchrones Nachladen als generelles Konzept und als Lösung für Performance Probleme
- Komplexer Aufbau der Webseiten
  - klassischer Aufbau von links/oben nach rechts/unten vs. Scrolling
  - komplexe UI Widgets (schwierige DOM-Struktur, Hidden-Elements, ...)

Was ändert sich

Neue Herausforderungen für automatische Tests

- Browserkompatibilität
  - wird besser, de facto nur noch 2 Browserengines (Webkit/Blink, Gecko)
  - deutliche Verbesserungen bei Standardisierung (und Dokumentation)
  - Einstellungen/Add-Blocker u.Ä. auf Nutzerseite (nicht kontrollierbar)
- Device-Adaption in der Anwendung (Responsive Design)
  - Layout und ScreenFlow vom Device abhängig
- Ständige Weiterentwicklung der Plattform (Browser)
- Ständige Weiterentwicklung der Pattern & Frameworks
  - jQuery, Node.js, Vue.js, React.js, Angular, Ember.js, ...

Worüber reden wir

Ideen und Anregungen, wie ihr das passende Tool finden könnt und was ihr 2019 von einem Testtool erwarten dürft

Tools für

- Funktionale Tests
- Integrationstests
- UI Tests
- Regressionstests

von Webanwendungen

Worüber reden wir NICHT

- Wille zum Testen, zum Automatisieren der Tests
- Testplanung, Teststrukturierung
- **Testdaten** – Notwendige Voraussetzung
  - Testautomatisierung nie ohne machbares Konzept für die Testdaten versuchen
  - verständlich, änderbar, Seiteneffekte vs. Reuse
  - technische und monetäre Restriktionen
- Verschiedene Sprachen, um Tests zu formulieren
  - Script, Gherkin, Modell, etc.
  - Tools bieten in der Regel mehrere Möglichkeiten an
  - wenn alles nicht passt, die Wunschsprache an die Tool-API anbinden

Dream On

Wenn ihr euch ein Testtool wünschen könntet, was würde es können?

- ROBUST bei technischen Änderungen
- SENSITIV bei fachlichen Änderungen
- WARTBAR
  - Warum schlägt dieser Test fehl?
  - Fehler im Programm oder Fehler im Test?
- LESBAR für die Fachseite, die Tester, die Entwickler
  - Sprache des Kunden verwenden
  - Qualität sichtbar machen
  - **Vertrauen**
  - Kommunikation (Was muss nicht mehr manuell getestet werden?)

Dream On

Wenn ihr euch ein Testtool wünschen könntet, was würde es können?

- Testumgebung
  - verschiedenen Browser
  - verschiedene Geräte
  - verschiedenen Bildschirmgrößen
- Output
  - Reports für verschiedenen Zielgruppen
  - Daumenkino
  - detaillierte Logs

Dream On

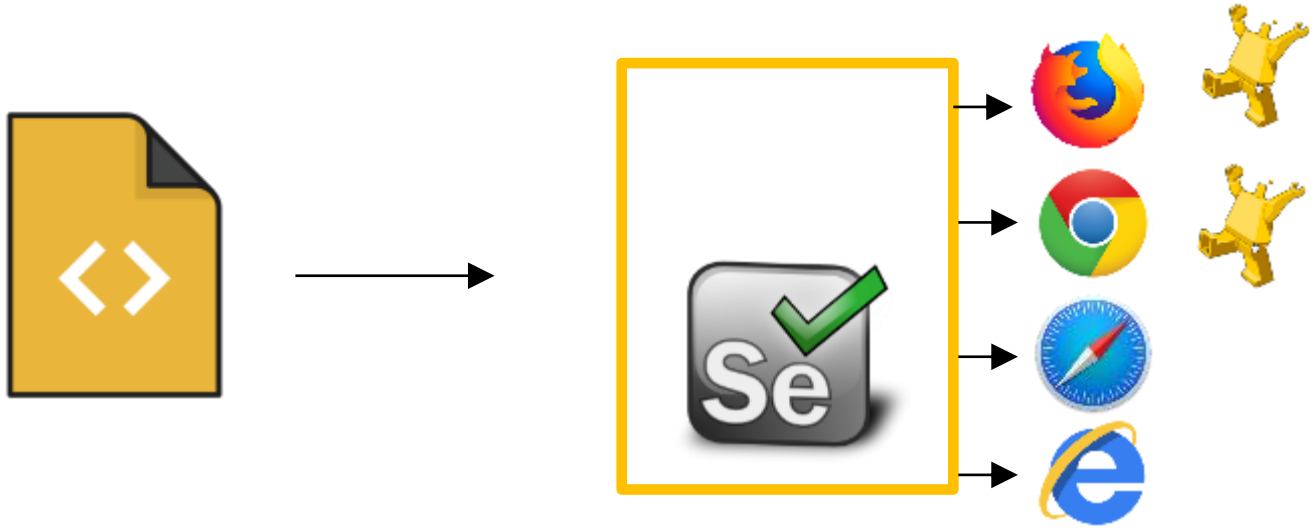
Wenn ihr euch ein Testtool wünschen könntet, was würde es können?

- Wie soll mit der Applikation interagiert werden?
  - URL öffnen und dann?
  - Finden von Widgets - Label benutzen oder DOM?
  - Interaktion – click & type
  - Scrolling automatisch oder Fehler
  - …
- Was soll der Test überprüfen können?
  - Layout, Positionen, Farben, verfügbare Optionen

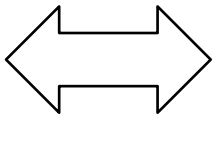
Was soll/muss das Tool NICHT können?

Und die Realität - Selenium

- DER Standard
- steuert echte Browser (Chrome, Firefox, Safari, IE)
- viele Language Bindings (Java, C#, Phyton, Ruby, …)
- eher ein Framework als ein Testtool



Und die Realität - Selenium



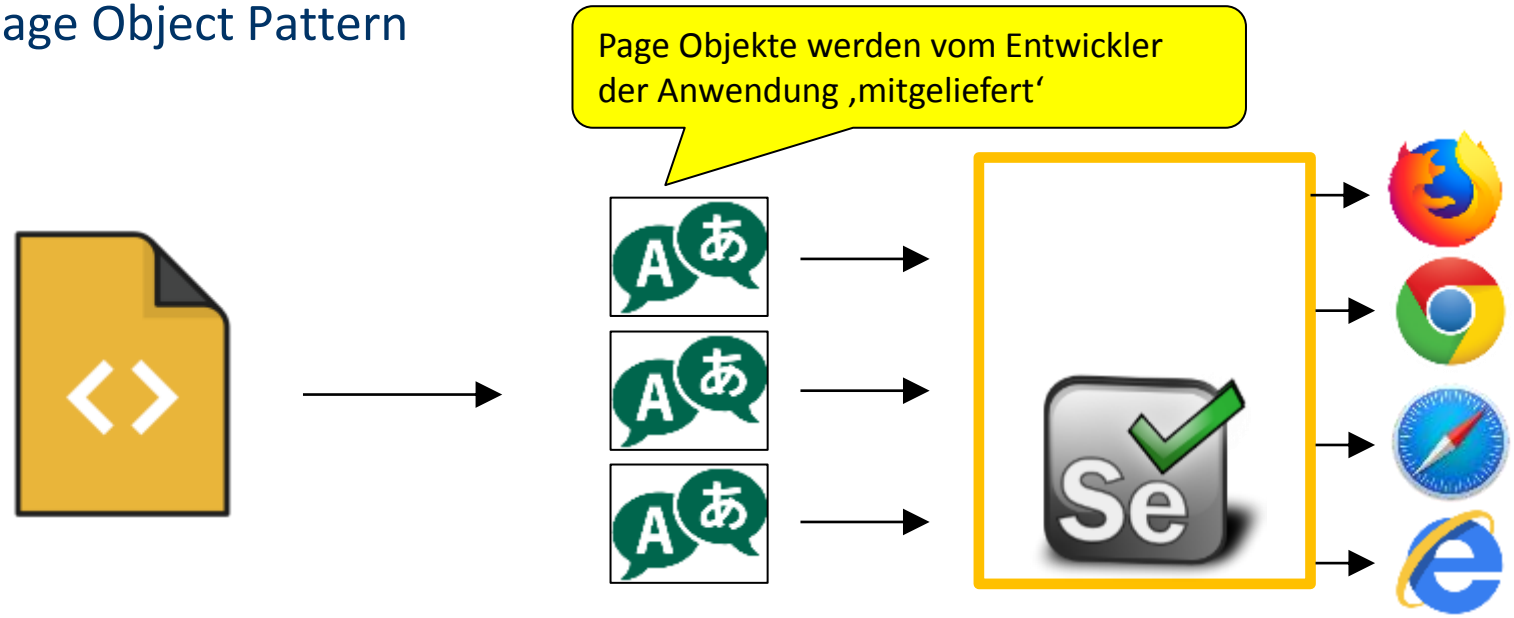
Die API

```
driver.get(URL)
WebElement elem = driver.findElement(By.id(„myID“));
elem.sendKeys(„Hello Selenium“);
```

- steuert den Browser – perfekte Umgebung
- Basis ist das WebElement und die By – suchen
  - Name, ID etc.; CSS Selektor; XPath
  - KEIN Zugriff auf den gesamten DOM-Baum
- Das ändert sich auch nicht mit dem **WebDriver -** W3C First Public Working Draft 12 September 2019 bzw. Selenium 4

Und die Realität - Selenium

- Kein Zugriff auf die Struktur der Seite - um einen Test zu schreiben, muss man
  - die Seite analysieren (z.B. CSS Selektor mit Browser Tools) oder
  - den Test aufzeichnen
- Page Object Pattern



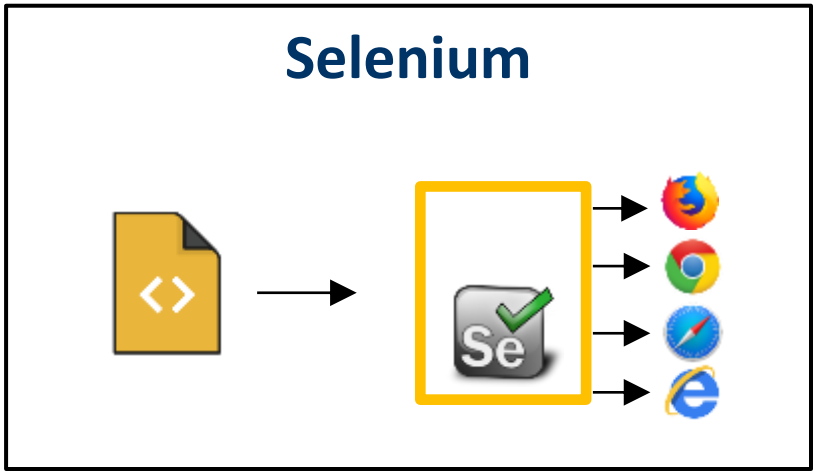
Und die Realität - Selenium

- ROBUST
  - echter Browser
  - Selektoren anfällig für technische Änderungen und ‚Async/Wait'-Probleme
- SENSITIV
  - eher übersensitiv - dem Autor des Tests überlassen
- WARTBAR
  - Page Objekte – arbeits- und pflegeaufwendig; Abhängigkeit vom App-Entwickler
- LESBAR
  - dem Autor des Tests überlassen

Kein direkter Zugriff auf den kompletten DOM

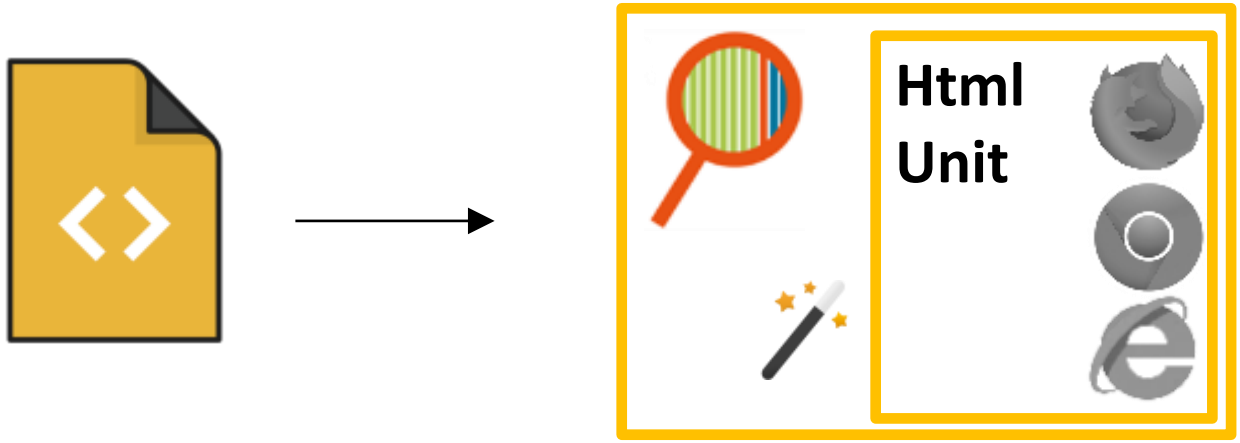
Viele Tools setzen auf Selenium auf und verbessern die Situation

Grundlegende Techniken hinter den Tools



Und die Realität - Wetator

- Basiert auf HtmlUnit
  - HtmlUnit komplett in Java geschriebene Browser Simulation incl. JavaScript
    - Chrome, Firefox, IE
  - HtmlUnit arbeitet Headless – kein Layouting der Seiten
  - HtmlUnit bietet kompletten Zugriff auf den DOM und auf die JS Engine
  - keine Abhängigkeit vom Betriebssystem (IE - Windows)
  - HtmlUnit ist auch Basis für andere Web Test Tools (z.B. Spring Test MVC)



Und die Realität - Wetator

- Kleines Befehlsset (open-url, set, select, click,…)
- Scripting aus Benutzersicht - „Intelligenz“
  - **set** | Your profession | Clean Coder

- Simulation von Geduld – kein wait im Testscript
  - Zustand der JS Engine wird ausgewertet
- Assert über eine normalisierte Textform der Seite
  - implizierte Asserts bei den Aktionen
  - unterstützt auch PDF/Excel etc. downloads
- Typing in Benutzergeschwindigkeit incl. Event/Ajax Support

Und die Realität - Wetator

- Report mit vorher/nachher Screenshot zu jedem Schritt inkl. Markierung des Aktions-Controls

- Viele ‚Details' können nicht getestet werden
  - Layout, Styling
- Basiert auf einer Simulation
  - nicht 100% kompatibel zum Browser
  - kein Drag&Drop

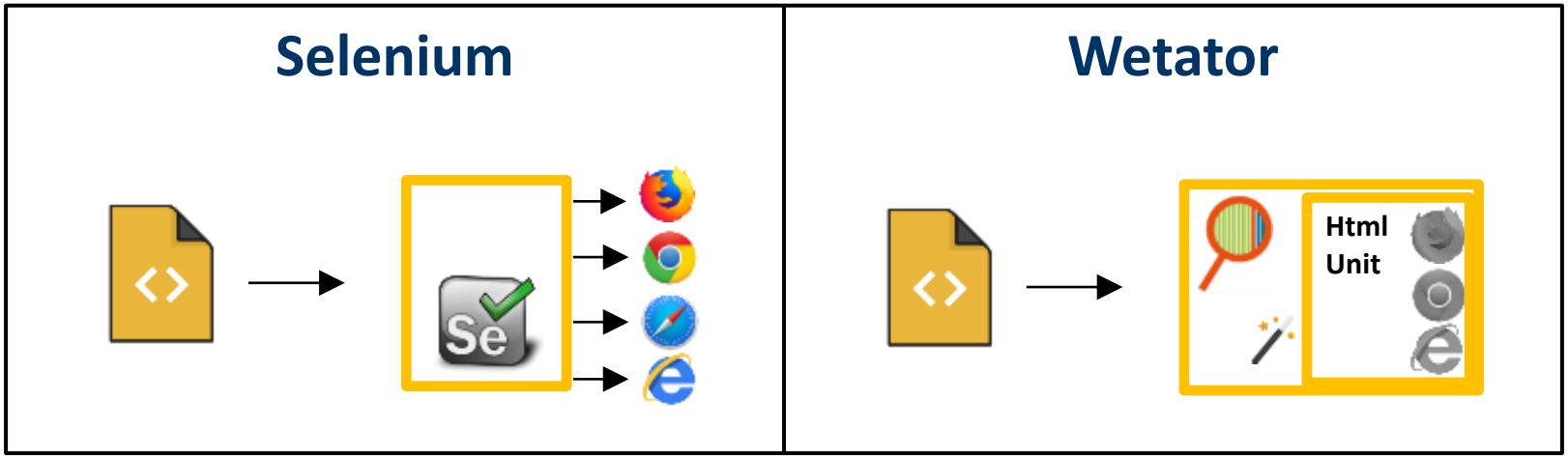
- Keine Aufnahmefunktion

Und die Realität - Wetator

- ROBUST
  - durch intelligente Suche unabhängig von technischen Änderungen
- SENSITIV
  - Normalisierung der Seite als Text – alles andere nicht testbar
- WARTBAR
  - wenige Befehle, intelligente Auswahl, durch Screenshots im Report keine Überraschungen
- LESBAR
  - detaillierter Report

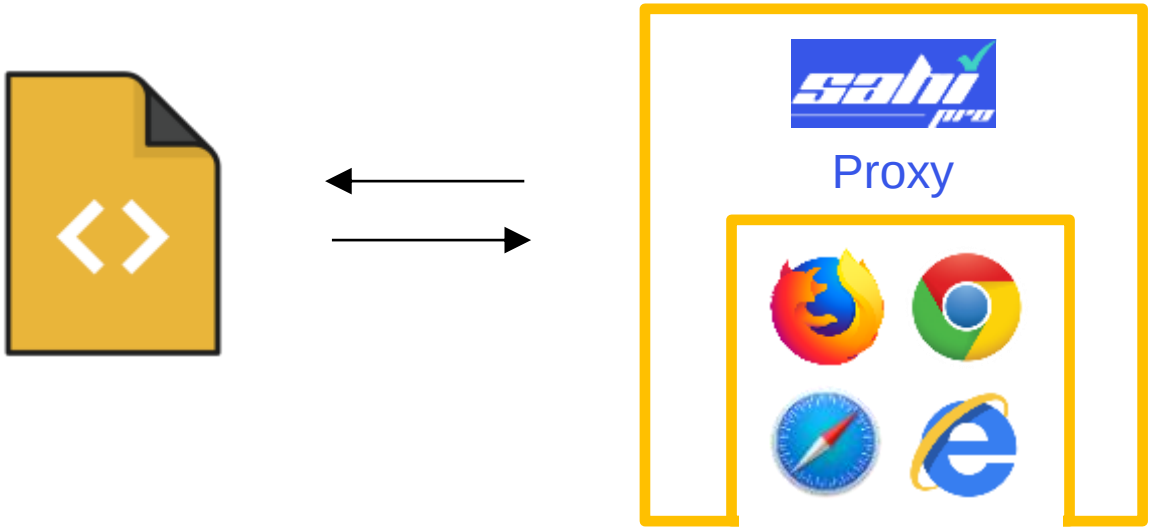
Problem sind die Defizite der Browser-Simulation

Grundlegende Techniken hinter den Tools



Und die Realität – Sahi

- Klinkt sich als Proxy zwischen Webbrowser und Webserver ein
- Proxy ist gleichzeitig eine JavaScript Engine
  - in Java geschrieben, deshalb zusätzlich zum Web-Test Ausführung beliebigen Java-Codes möglich



Und die Realität – Sahi

- Reichert die Web-App mit JavaScript-Eventhandlern an, um Tests aufzuzeichnen und wieder abzuspielen
- Deshalb (weitgehend) browserunabhängig
- Nur relevante Script-Anteile werden im Browser ausgeführt

```
<!--?xml version="1.0" encoding="UTF-8" ?-->
<!doctype html>
<html xmlns="http://www.w3.org/1999/xhtml">
▼<head id="j_idt2">
...   <!--SAHI_INJECT_START--> == $0
    <script src="/_s_/sprc/concat.js,assert.js,listen.js,async.js,actions.js,language_pack.js" id="_sahi_concat">
    </script>
    <script src="//sahi.example.com/_s_/dyn/SessionState_config/sahiconfig.js"></script>
    <!--SAHI_INJECT_END-->
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta http-equiv="content-style-type" content="text/css">
    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="expires" content="-1">
    <meta http-equiv="cache-control" content="no-cache,no-store,must-revalidate,private">
    <title>Web Beans Demo Application
    </title>
```

Und die Realität – Sahi

- Sehr einfach aufgebaute Scripts
  - JavaScript als Sprache, deshalb Fallunterscheidungen, Funktionen etc. möglich
  - nutzt „pseudo-intelligente“ Object Identification

```
_click(_button("Login"))
_assertExists(_span("Customer Browser..."));
_assertVisible(_span("Customer Browser..."));
_click(_span("Customer Browser..."))
_setValue(_textbox("cQueryEditor:firstname"), "Jane")
_setValue(_textbox("cQueryEditor:lastname"), "Doe")
_setSelected(_select("cQueryEditor:gender"), "female")
```

Probleme mit JavaScript-lastigen Webframeworks wie Vaadin / Angular

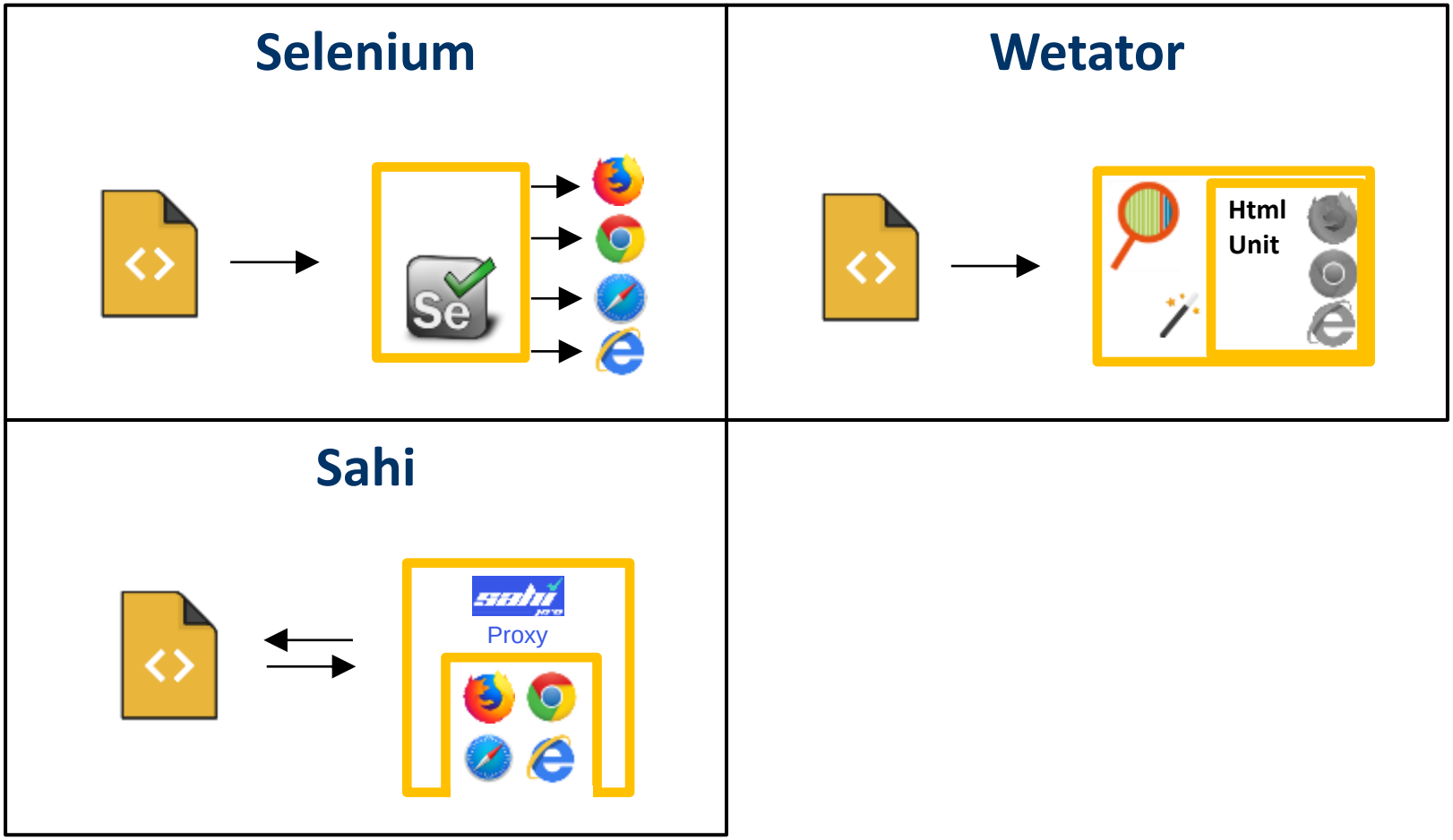
- sowohl bei Aufzeichnung als auch bei Playback

Und die Realität – Sahi

- ROBUST
  - quasi intelligente Suche, trotzdem nur bedingt unabhängig von technischen Änderungen
- SENSITIV
  - flexibel, dem Autor des Tests überlassen
- WARTBAR
  - Fehleranalyse wenig unterstützt, Reports eher für Manager
- LESBAR
  - konfigurierbarer Report, viel Handarbeit für Übersichtlichkeit
    - bspw. Screenshot im Fehlerfall möglich, aber selbst zu implementieren

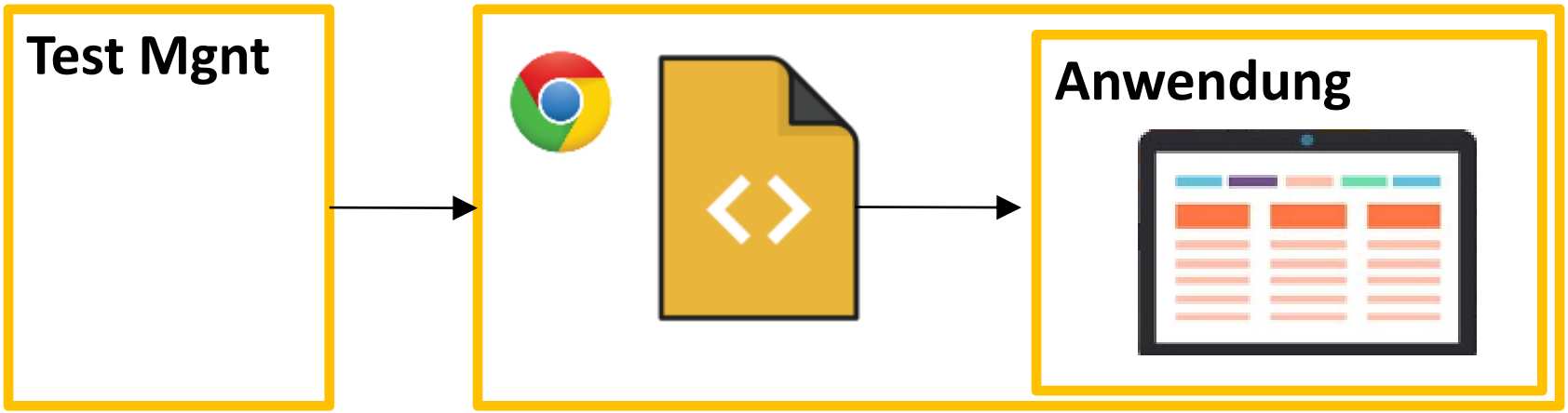
Steuerung per JavaScript nicht immer anwendbar

Grundlegende Techniken hinter den Tools



Und die Realität - cypress

- Läuft im Browser – neben der Anwendung
- Ähnlich zu den WebDev Tools der Browser
  - **Selector Playground**
- Report mit vorher/nachher Screenshot zu jedem Schritt incl. Markierung des Aktions-Controls



Und die Realität - cypress

- Ansprechen der Controls über jQuery Selektoren oder über Text Content des Elements
- Suchen haben einen wait/retry eingebaut
  - Timeout mit default und per Befehl überschreibbar
- Interaktion mit den Elementen über simple Befehle
- Viele detaillierte Asserts möglich
  - implizierte Asserts bei den Aktionen
- Unterstützung für Test verschiedener Bildschirmgrößen

Und die Realität - cypress

- Testscript in Javascript
- Zugriff auf DOM aus dem Testscript möglich
- CI Integration
- Github Integration

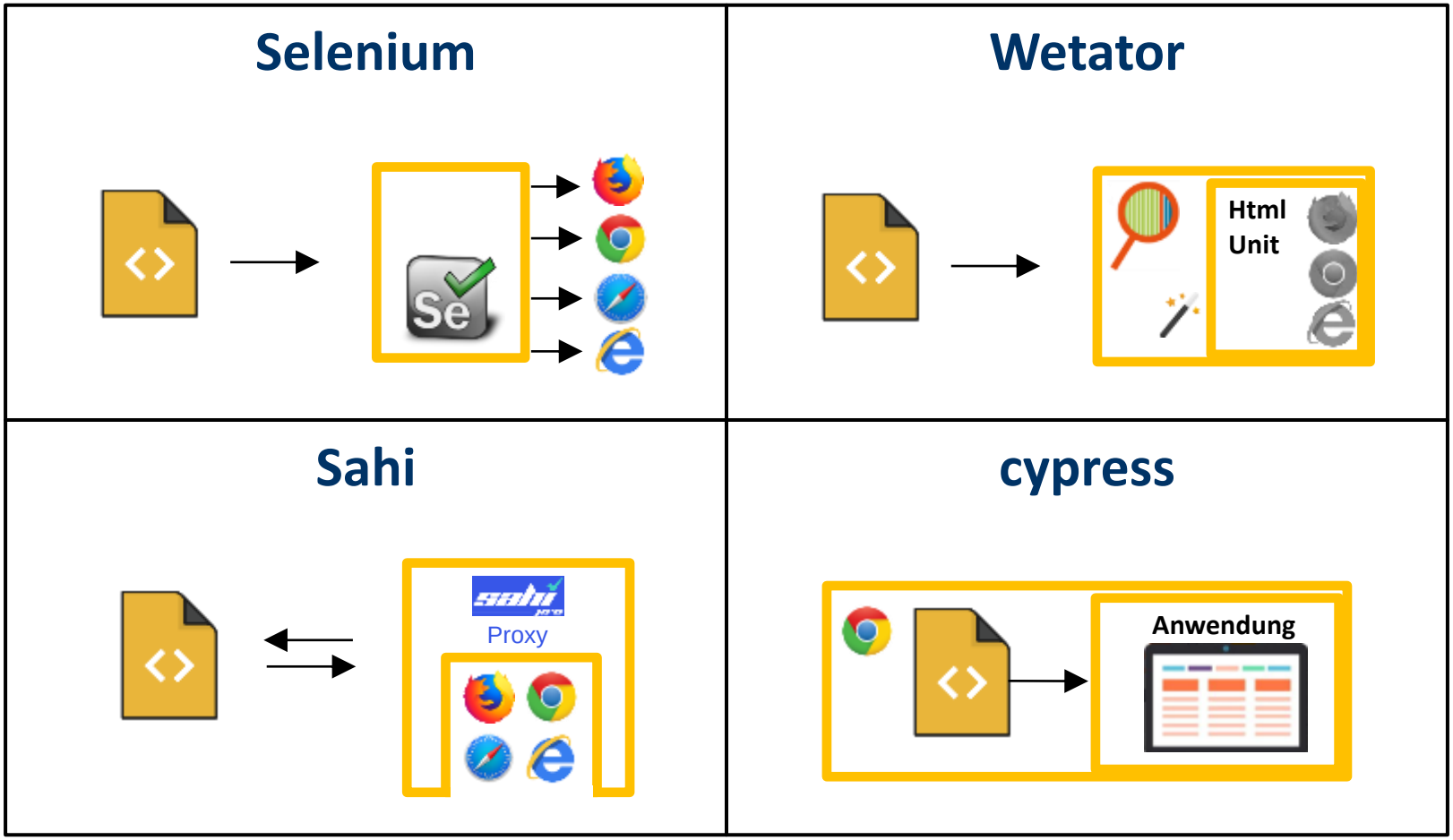
```
cy.visit(myUrl)
cy.get('#languageSelector', { timeout: 10000 }).should('exist')
cy.get('#languageSelector').select('Englisch')
cy.get('#loginButton').click()
cy.get('#Customer_Browser').click()
cy.get('#cQueryEditor\\:firstname').type('Jane')
cy.get('#cQueryEditor\\:lastname').type('Doe')
cy.get('#cQueryEditor\\:refreshButton').click()
cy.get('#cBrowser\\:0\\:r').should('contain', 'Doe')
```

Und die Realität - cypress

- ROBUST
  - jQuery oder Textsuche – nicht unabhängig von technischen Änderungen
- SENSITIV
  - sehr viele Assert Möglichkeiten – flexibel an die Aufgabe anpassbar
- WARTBAR
  - modern / nah an der Entwicklung von JS basierten Anwendungen
  - Selector Playground und durch Screenshots im Report keine Überraschungen
- LESBAR
  - detaillierter, interaktiver Report

Aktuell nur Chrome; Ziel eher die App Entwicklung

Grundlegende Techniken hinter den Tools



Test your Testtool

Drum prüfe wer sich ewig bindet ….

Eigene Applikationen

Seiten von Providern der verwendeten Web-Frameworks

- Angular, Vaadin, PrimeFaces, ...

Freie Beispielapplikation, z.B. auf github / gitlab

- z.B. umfangreiche Sammlung: http://the-internet.herokuapp.com/

Spezielle Testverfahren und deren Werkzeuge

Nur der Vollständigkeit halber erwähnt:
Werkzeuge für den nichtfunktionalen Test (sofern automatisierbar...)

Last- und Performancetest, Stresstest, Speed-Test

- Kann man hier die funktionalen Tests wiederverwenden?

Security-Test

- eine Übersicht der Werkzeuge findet sich u.a. im
  OWASP Testing Guide – Appendix A:
  https://www.owasp.org/index.php/Appendix_A:_Testing_Tools

Summa Summarum

Werkzeuge sind sehr unterschiedlich in

- Architektur, Technik
- Alter (!) und Anpassung an neue Anforderungen
  - **Inversion** of Control:
    - Früher: Input → Check Result
    - Heute: Input → Wait for Result (inklusive Simulation von *Geduld*)
- Zielgruppe

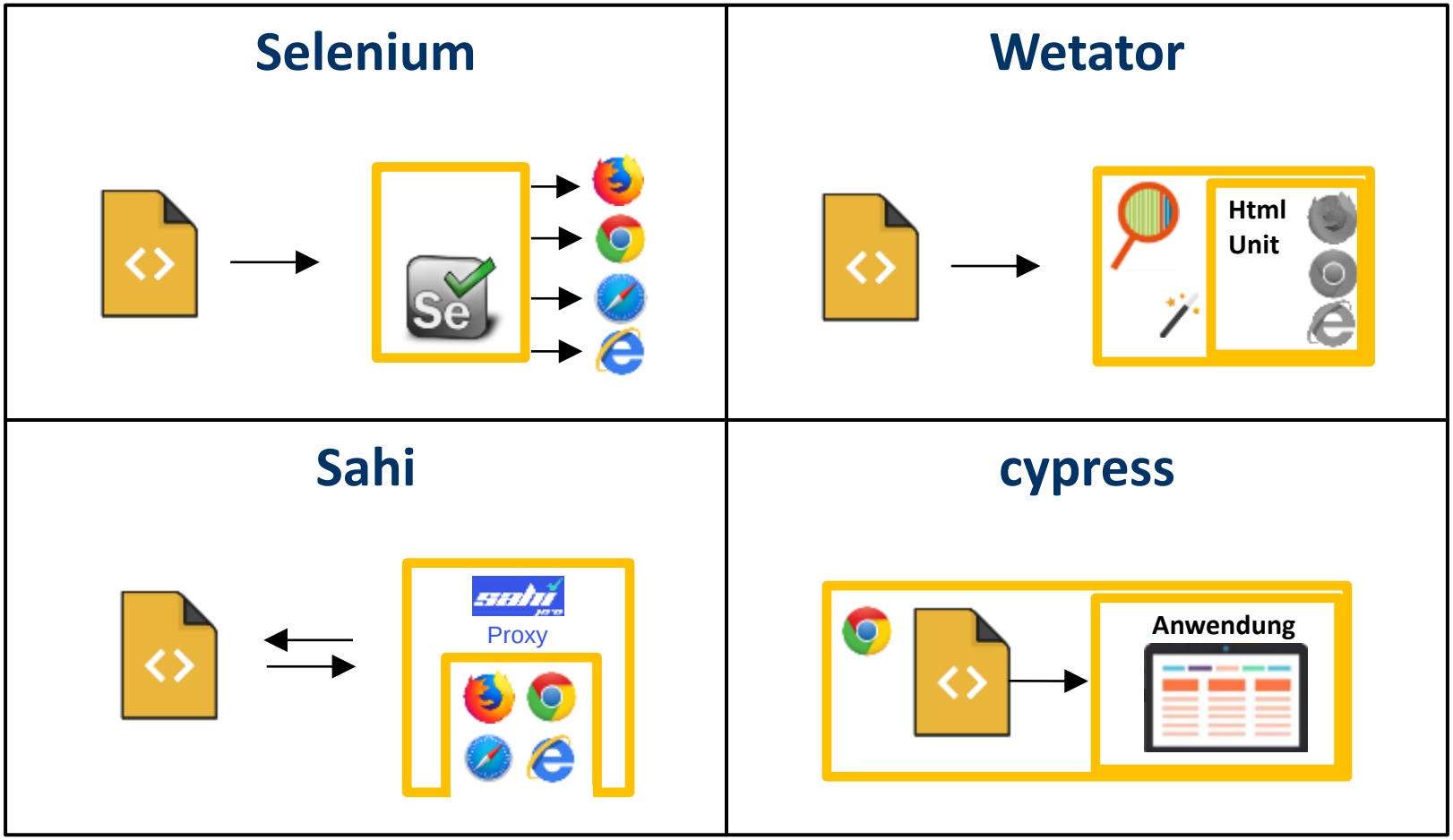
Auswahl sollte unbedingt an die eigenen Bedürfnisse angepasst sein

- Auch zukunftsweisend – will man wirklich alle Tests neu schreiben?

Eine Frage sollte immer im Vordergrund stehen:

Was will ich eigentlich testen?

Grundlegende Techniken hinter den Tools



Besucht uns am Stand!

Gewinnspiel mit Bon-Shooter

jobs.gebit.de

Anhang: Referenzen

Selenium https://www.seleniumhq.org/

- Sehr viele andere (sowohl OpenSource und freie als auch kommerzielle) Tools setzen darauf auf

W3C WebDriver (Draft) https://www.w3.org/TR/webdriver/

Sahi

- Freie Version (etwas älter) https://sourceforge.net/projects/sahi/
- Eingeschränkt freie Version https://sahipro.com/

Wetator https://www.wetator.org/

cypress https://www.cypress.io/

HtmlUnit http://htmlunit.sourceforge.net/

HtmlUnit – WebDriver https://github.com/SeleniumHQ/htmlunit-driver

Anhang: Grundlegende Techniken hinter den Tools

(komprimiert)

Selenium:

- Selenium Webdriver in Kommunikation mit spezifischen Driver-Implementierungen für einzelne Browser (browserabhängig, trotzdem große Unterstützung), um Webbrowser fernsteuern zu können – demnächst W3C Webdriver Spec

Wetator:

- Basiert auf HtmlUnit (Browser Simulation); dadurch volle Kontrolle über den DOM und den Zustand der Javascript Engine

Sahi:

- Klinkt sich als Proxy zwischen Webbrowser und Webserver ein; reichert die Web-App mit JavaScript-Eventhandlern an, um Tests aufzuzeichnen und wieder abzuspielen; browserunabhängig

Cypress:

- Klinkt sich als Proxy zwischen Webbrowser und Webserver ein; läuft im gleichen Prozess wie die Web-App und kann deshalb auf alles zu greifen; aktuell nur Chrome

☝ Wer weiß, wie das Tool funktioniert, kann Problemen leichter vorbeugen