LEHR- UND FORSCHUNGSEINHEIT
PROGRAMMIERUNG &
SOFTWARETECHNIK

Softwaretechnik 2015/2016

PST Lehrstuhl

Prof. Dr. Matthias Hölzl

Joschka Rinke

AGENDA

Übung 13:
28.01.2016

- **Fragen**
- **Besprechung Blatt12**
- **Klausur Tipps**

Aufgabe 1 – Verifizieren und Validieren

socrative by MasteryConnect

Was versteht man unter Validierung und was unter Verifizierung?

- **Validierung: Entspricht Programm den Wünschen**
- **Validierung: Entspricht Programm den Spezifikationen**
- **Verifizierung: Entspricht Programm den Wünschen**
- **Verifizierung: Entspricht Programm den Spezifikationen**

Aufgabe 1 – Verifizieren und Validieren

Validierung:
„Versuchen wir das Richtige zu machen?“…
…soll heißen, macht das Programm das, was es machen soll?
Entspricht das Programm den Wünschen des Kunden?

Verifikation:
„Machen wir das, was wir machen richtig?“…
…soll heißen, macht das Programm Fehler?
Entspricht das Programm den Spezifikationen?

Aufgabe 1 – Verifizieren und Validieren

Verifikations- & Validierungstechniken:

- **Statisch: Programm wird nicht ausgeführt**
  - **→ Code Inspektion**
  - **→ formale Methoden**
- **Dynamisch: Programm wird ausgeführt & Terminierungszustände oder Ergebnisse werden überprüft**
  - **→ Tests**
  - **→ Fuzzing**

(Validierung findet meist dynamisch statt)

Aufgabe 1 – Verifizieren und Validieren

Code Navigation:

Hilfreich für Code Inspektion

Code Navigation in Eclipse (kleine Auswahl):

Ctrl + Shift + R : open any resource

Ctrl + E : switch between open tabs; Ctrl + Bild < > : go to left/right tab

Ctrl + T : open type hierarchy of package

Ctrl + O : view all methods, press again to view all superclass methods, too

Ctrl + 1 : quick fix

Ctrl + Shift + O : organize imports

Ctrl + L : go to line

Ctrl + Shift + F : auto format

Ctrl + Alt + H : open call hierarchy

Alt + Shift + R : refactor

Hold Ctrl while clicking on a class/method/interface to open it

Ctrl + Q : go to last edited place

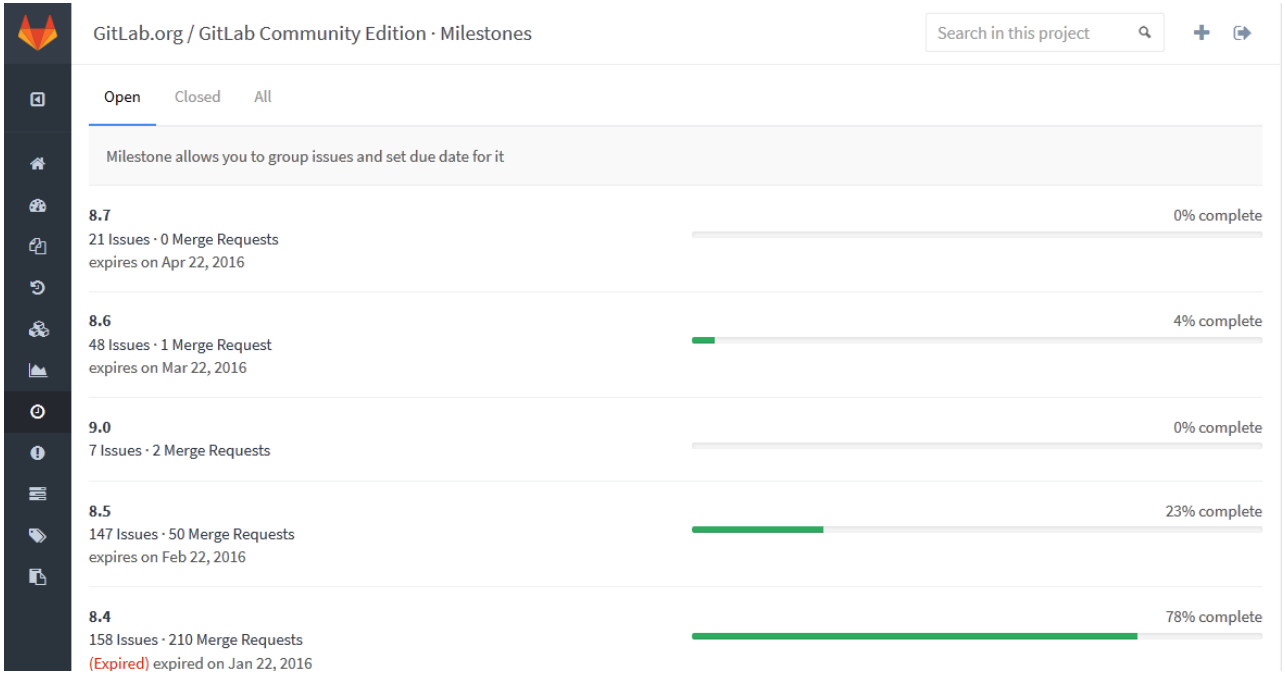
List of possible navigation in Eclispe: window → preferences → general → keys

Aufgabe 1 – Verifizieren und Validieren

Milestones: Legen Ziel für ein Projekt fest

- ➜ **Oft Zusammenfassung von Issues und Merge Requests**
- ➜ **Liste der Issues, Merge Requests & Zieldatum**
- ➜ **In SCRUM als mögliche Sprintziele**

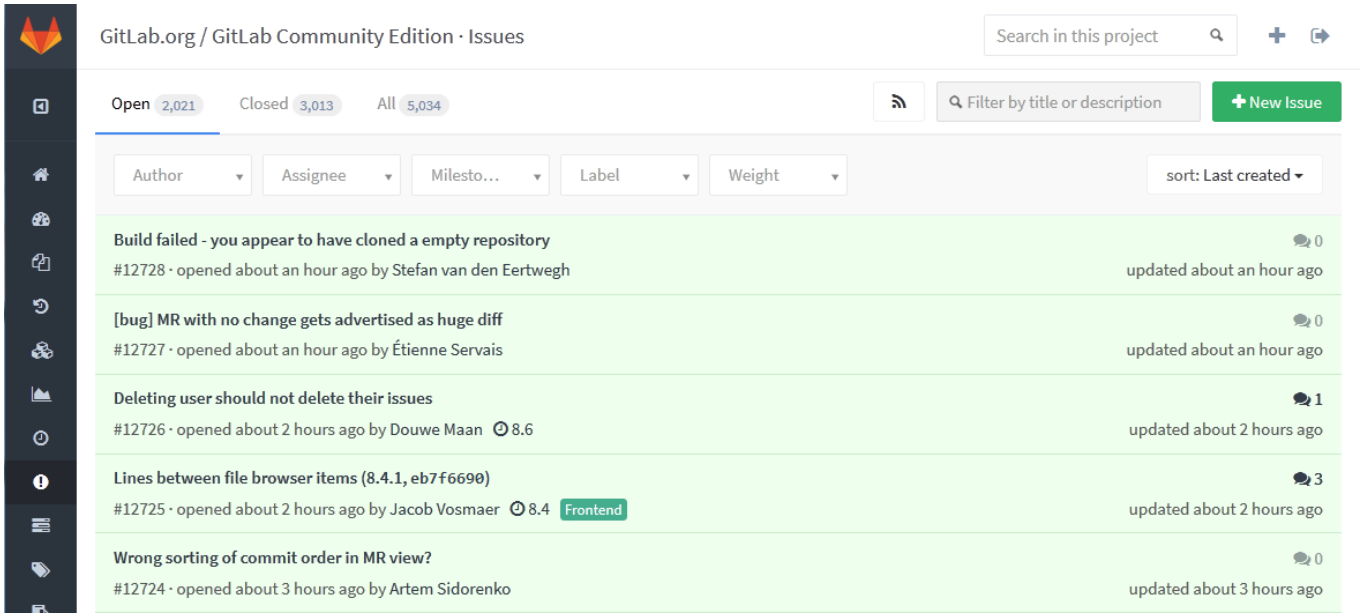
Liste der Milestones eines Projekts mit Issues, Merge Requests & due date:



Aufgabe 1 – Verifizieren und Validieren

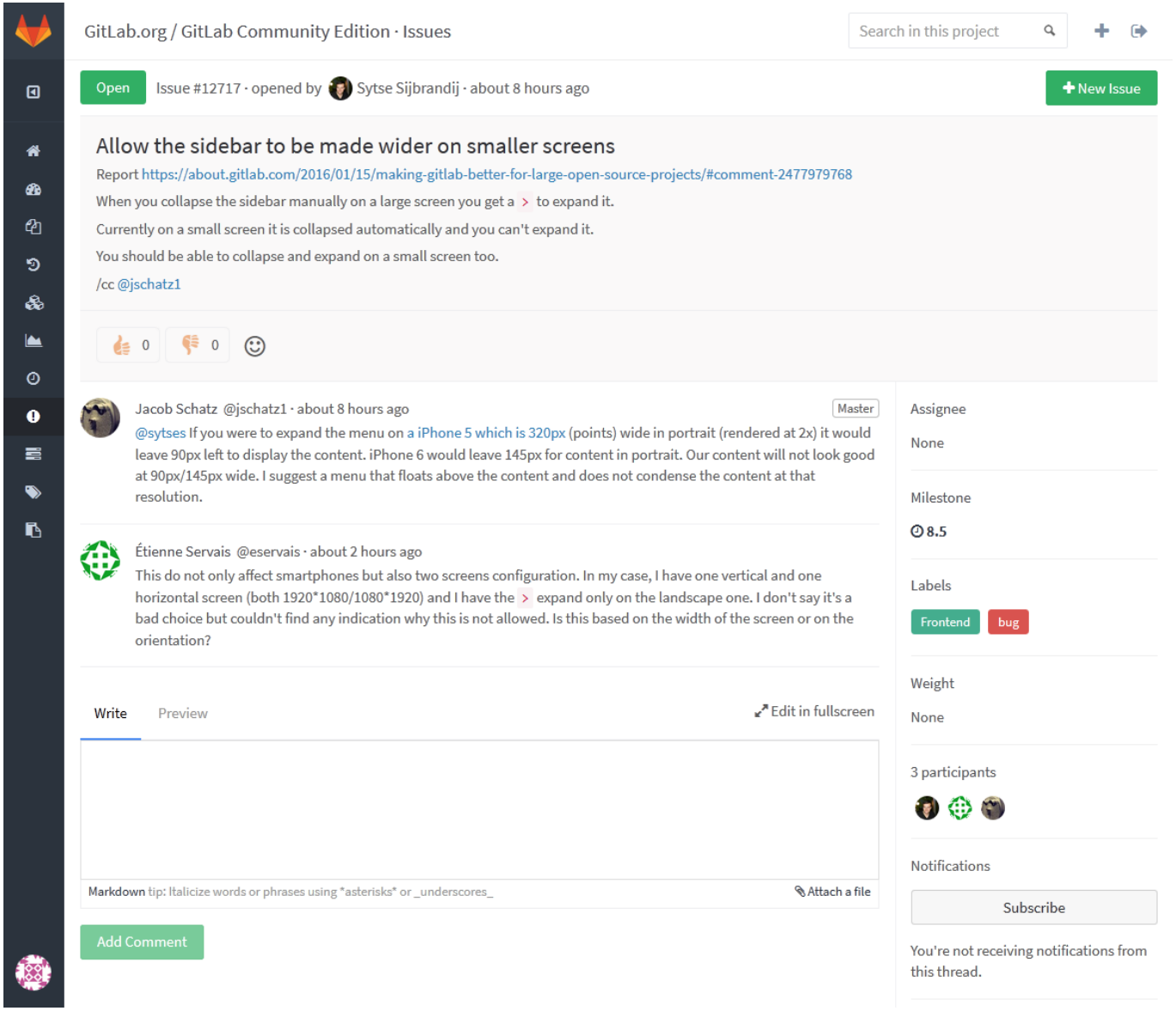
Issues: beschreiben konkrete Probleme/ Verbesserungsvorschläge

- ➜ **Beschreibung**
- ➜ **Reproduzierbarkeit**
- ➜ **Klassifikation**
- ➜ **Verantwortlichkeiten**
- ➜ **open unassigned/ open assigned/ closed**



Aufgabe 1 – Verifizieren und Validieren

Issue:



Aufgabe 1 – Verifizieren und Validieren

Verteilte Versionskontrolle:

- **Kein zentrales Repository, jeder Entwickler nutzt lokales Repository**
- **Es existiert aber für gewöhnlich ein offizielles Repository**

GitLab (Features):

- **Issue tracking: Issues werden kategorisiert, assigned, etc.**
- **Issue assignment: Issues sind assigned oder unassigned;**
  Entwickler assigned issue und bearbeitet issue dann (open assigned)
- **Merge request: mehrere Branches in Repository → pull;**
  Meldung an andere Entwickler, dass ein Issue
  abgeschlossen wurde; Anstoß zu Review und Diskussion

Aufgabe 1 – Verifizieren und Validieren

socrative by MasteryConnect

Warum ist es sinnvoll innerhalb eines Projektes einen einheitlichen Workflow zu verwenden?

- **Automatisierte Erzeugen von konsistentem Code**
- **Vermeiden von Missverständnissen im Team**
- **Eindeutige Fehlerzuweisung zu Entwickler**
- **Erhöhte Motivation der Entwickler**

Aufgabe 1 – Verifizieren und Validieren

Workflow: Richtlinie zur Verwendung von DVCS

- ➜ **einheitliches Vorgehen**
- ➜ **Vermeidung von Fehlern**
- ➜ **Zurückverfolgung von Changes etc.**

Aufgabe 1 – Verifizieren und Validieren

GitHub Workflow (vgl. GitLab Features):

- **Create own branch from master:**
  - **→ master stays deployable**
  - **→ descriptive name for own branch**
- **Add commits:**
  - **→ keep track of development and add commit message**
  - **→ transparent history**
  - **→ seperate unit → rollback possible**
- **Pull request (Merge request bei GitLab):**
  - **→ initiate discussion and reviewing**
- **Discussion:**
  - **→ questions can be answered**
  - **→ tests available?**
  - **→ according to code guidelines?**
- **Deploy:**
  - **→ discussion passed → deploy for verification under production**
- **Merge → into master branch**

Aufgabe 1 – Verifizieren und Validieren

Continuous Integration (CI):

- ➜ **Entwickler integrieren eigene Arbeitsversionen so oft wie möglich**
- ➜ **in das System um „integration hell“ zu verhindern**
- ➜ **Build und Tests laufen automatisiert auf einem Server**
- ➜ **hohe Toolunterstützung (Server-/ webbasiert)**
- ➜ **hohe Konfigurierbarkeit**

Ablauf:

1. **Entwickler nimmt Änderungen vor**
2. **Code funktioniert, Tests werfen keine Fehler**
3. **Code wird in VCS eingecheckt**
4. **CI Server bemerkt Änderungen**
   - **CI checks out Code**
   - **CI builds the whole system**
   - **CI runs all tests**
   - **Results are published**
5. **Entwickler werden über Probleme informiert**

Aufgabe 1 – Verifizieren und Validieren

socrative by MasteryConnect

Was könnte in Zusammenhang mit Continuous Integration ein Nightly Build sein?

- **Eine häufig verwendete Form von CI in der Entwickler vornehmlich nachts entwickeln**
- **Eine Vorstufe von CI in der Build Prozess nachts und automatisiert erfolgt**
- **Nightly Build hat eigentlich nichts mit CI zu tun, das ist ein häufig auftretendes Missverständnis**

Aufgabe 2 – Formale Methoden & Verifikation

Formale Verifikation: Nachweis der Korrektheit eines Programms

- **Formales Modell als Spezifikation**
- **Mathematische Analyse des Modells**
  - **➜ Modell (des Programms) in geeignetem Formalismus**
  - **➜ Aussagen über Modell (das das Programm repräsentiert)**
- **Je ausdrucksmächtiger die Logik, desto mehr manueller Beweisaufwand**

➜ Werden Eigenschaften von Modell erfüllt

(leichtgewichtige Methoden auf Programmcode möglich vgl. ACL2 aus VL oder C++ Checker, die problematische Stellen markieren)

Aufgabe 2 – Formale Methoden & Verifikation

Formale Techniken zur Verifikation:

- **+ erhöhte Systemsicherheit**
- **+ erhöhte Zuverlässigkeit des Systems**
- **+ Garantien können gegeben werden, die sonst nicht möglich sind**
- **– formale Spezifikation vs. tatsächliche Anforderungen**
- **– formale Beweise können Fehler enthalten**
- **– falsche Annahmen als Ausgangssituation**

Aufgabe 3 – Testen

Kontrollfluss-Graph (CFG): gerichteter Graph der Kontrollstruktur eines Programms repräsentiert

Testabdeckung: unterschiedlich, je nach Definition…

- **Zeilenabdeckung: Prozentsatz der getesteten ausführbaren Zeilen**
- **Anweisungsabdeckung: Prozentsatz der getesteten Anweisungen**
- **Zweigabdeckung: Prozentsatz der getesteten Kanten im CFG**
- **Pfadabdeckung: Prozentsatz der getesteten Pfade von Start- zu Endknoten**
- **Bedingungsabdeckung: Prozentsatz wird abhängig von Verfahren über Anzahl der getesteten Bedingungen ermittelt**

Es kann nie sichergestellt werden, dass ein System keine Fehler hat.
Man kann nur sicherstellen, dass getestete Szenarien funktionieren.

Aufgabe 3 – Testen

Klassifikation von Tests:

- **Klassifikation nach Zielen: Validierungs- vs. Defekttesten**
  - **Validierung (Happy Path): prüfe ob Test gewünschte Funktion erfüllt**
  - **Defekt (Unhappy Path): versuche Fehlerzustände zu finden**
- **Klassifikation nach relativer Größe des SUT:**
  Unit, Component, System (Integration)

Testarten:

- **Unit Test: isoliertes Testen einzelner Methoden/ Klassen**
- **Component Test: isoliertes Testen einzelner Systemkomponenten**
- **Integration Test: Testen des kompletten Systems (Entwickler)**
- **Akzeptanz Test: Testen ob Business Requirements erfüllt werden (Kunde)**

Aufgabe 3 – Testen

Weitere Testarten:

- **Regression Test: behobene Fehler testen**
- **Performance Test: Verhalten des Systems unter Belastung**
- **Usability Test: wie gut kommen User mit System zurecht**
- **Entwicklungstest: Tests während der Entwicklungsphase durchführen**
- **Release Test: Vollständige Systemversion wird vor Übergabe getestet**
- **Endbenutzertest: Benutzer testen System in eigener Systemumgebung**

Strategien:

- **White-Box: Zugriff auf Systeminterna (inklusive Code)**
- **Grey-Box: Zugriff auf Designdokumente aber nicht auf Code**
- **Black-Box: kein Zugriff auf Systeminterna**
- **Top-Down: beginne mit Tests auf hoher Integrationsebene**
- **Bottom-Up: beginne mit Tests auf niedrigster Integrationsebene**

Aufgabe 3 – Testen

socrative
by MasteryConnect

Klassisches Fuzzing ist immer eine Form des…

- **…White-Box Testing**
- **…Black-Box Testing**
- **…Top-Down Testing**
- **…Bottom-Up Testing**

Aufgabe 3 – Testen

„Mocks“, „Stubs“, „Dummy-“ und „Fake-Objects“ werden in erster Linie bei…

- **…Unit Tests…**
- **…Component Tests…**
- **…Integration Tests…**
- **…Akzeptant Tests…**

…verwendet.

Aufgabe 3 – Testen

Unit Test: isoliertes Testen einzelner Methoden/ Klassen

Objekte andere Klassen stehen nicht zur Verfügung, werden aber häufig benötigt

Test Doubles:

- **Mock: Platzhalter, der Schnittstellen simuliert und überprüft ob erwartete Aufrufe erfolgen**
- **Stub: stellt pseudo Antworten für Aufrufe innerhalb des Tests bereit**
- **Dummy Object: Werden nicht genutzt, füllen lediglich Parameterlisten**
- **Fake Object (drivers): enthalten funktionierende Implementierung, die aber so abgespeckt ist, dass sie im eigentlichen Programm nicht verwendet werden**

Aufgabe 3 – Testen

Was ist der Unterschied zwischen Mocks und Stubs?

- **Mocks verifizieren simulierten Programmablauf, Stubs nicht**
- **Stubs verifizieren simulierten Programmablauf, Mocks nicht**
- **Ein Mock ist immer auch ein Stub**
- **Ein Stub ist immer auch ein Mock**