Unit Testing & Continuous Integration

(basierend auf Folien von Marc Bux)

Patrick Schäfer

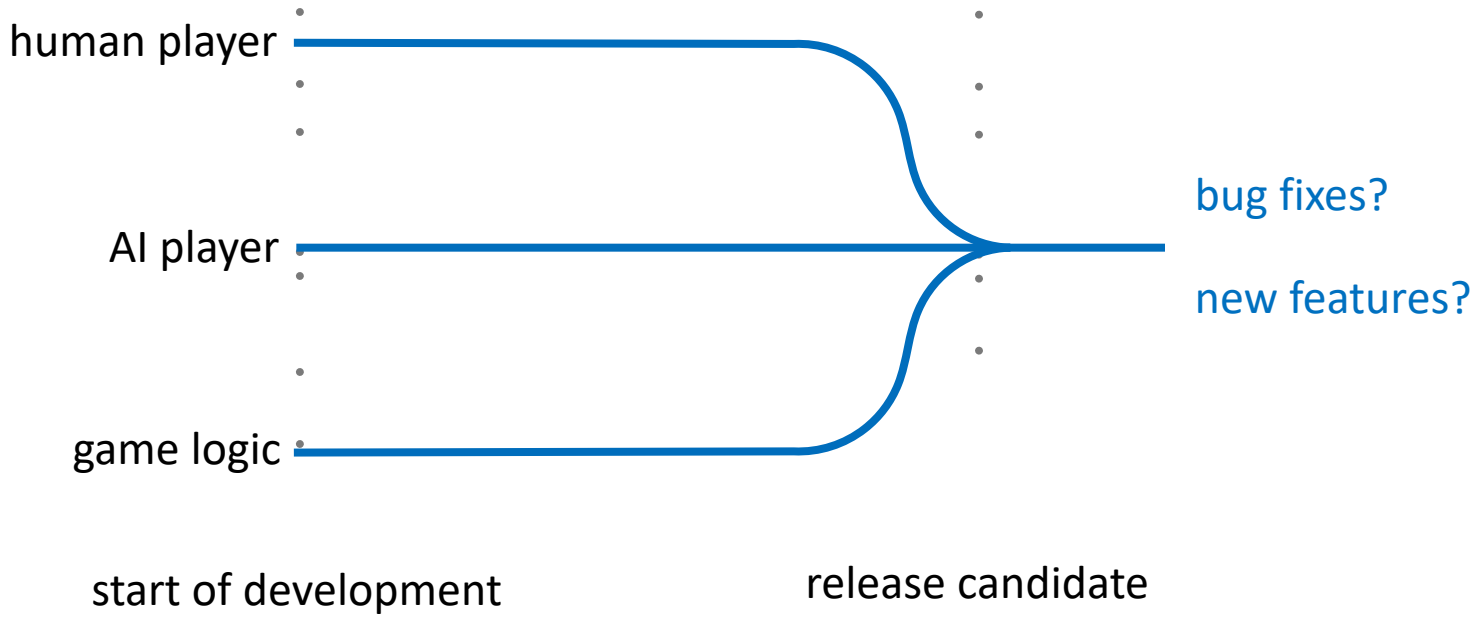
patrick.schaefer@hu-berlin.de

Today

- New User Story:
  - Observer-Mode
  - Backlog-Grooming
- Technical Talk: Unit tests & continuous integration
- This week:
  - technical refinement for the new user stories
  - finalize tasks in your sprint backlog (incl. tests, code review)
- Next Monday, 13 c.t.
  - Sprint #1 Review Meeting; bring a laptop & presentable prototype
  - Sprint #2 Sprint planning; kickoff; present your Sprint Backlog

How Software Used to Be Developed

- software developed in teams
- software is divided in modules
- modules are assigned to teams
- modules have to be integrated at some point
- integration is done manually at release time



What Are the Issues?

- effects of changes (bugfixes, new features) are hard to predict
  → Unit Tests

- no feedback (by the product owner) possible before release
  → Continuous Integration

- unnecessarily complicated integration process
- bad software quality
- integration hell: stress and frustration towards the end of a project

Unit Tests

unit test: systematic, automated test of a software component

- unit: smallest testable part of an application
  - in object-oriented programming, this is usually a method (or a class)
- the unit is tested in isolation (of other units)

- usually written in the same language as the tested software
- tests should be written by the developer (of the unit)
- can be written prior or concurrent to unit development
- tests can succeed or fail

Advantages of Unit Testing

unit tests …

- … facilitate the design of robust code
  (bugs make it through only if the unit and its test are poorly designed)
- … provide immediate feedback on the effect of changes in the code base
- … serve as a to-do list subsequent to changes in the code base
- … help define what a piece of code is (and isn't) supposed to do

Best Practices

1. test the complete intended behavior of the unit, including
   a) expected cases (e.g., sort an unsorted array)
   b) special cases (e.g., sort already sorted array)
   c) boundary conditions (e.g., sort empty array)
2. test every behavior only once (no redundant test)
3. test only one unit at a time
4. design tests independent of the application‘s state
5. design tests independent of external resources
6. name unit tests clearly and consistently
7. whoever breaks a working unit is responsible for fixing it

A concrete Example

- We want to add the functionality to claim routes
- From the rule book:

  To claim a route, a player must play a set of cards equal to the number of spaces in the route. A set of cards must be of the same type. Most routes require a specific type of set. For example a Blue route must be claimed using blue-colored Passenger Car cards. Some routes – those that are Gray colored – can be claimed using a set of cards of any one color.

  [...] Locomotives are Multi-colored and act as a wild card that can be part of any set of cards when claiming a route. [...]

A naïve Example

```
package de.huberlin.wbi;

public class Player {
  public static enum PassengerColor {
    Blue, Black, Red, Rainbow
  }
  int[] playerCards = new int[4];

  /**
   * Claim a route between two adjacent cities using
   * the payByColor-passenger-cards in our hand
   */
  public boolean claimRoute(
        PassengerColor payByColor,
        int routeCost,
        PassengerColor routeColor
      ) {
    int currentCards = playerCards[payByColor.ordinal()];
    // Pay for a route between two adjacent cities on the map
    if ( (payByColor == routeColor
          || payByColor == PassengerColor.Rainbow
          || routeColor == PassengerColor.Rainbow)) {
       if (currentCards >= routeCost) {
        playerCards[payByColor.ordinal()] -= routeCost;
        return true;
       }
    }
    // we cannot buy the route
    return false;

  }
}
```

```
public class PlayerTest {

  @Test
  public void testContains() {
    Player p = new Player();
    Arrays.fill(p.playerCards, 3);

    assertTrue(p.claimRoute(Player.PassengerColor.Black, 1, Player.PassengerColor.Black));
    assertTrue(p.claimRoute(Player.PassengerColor.Red, 1, Player.PassengerColor.Rainbow));
    assertTrue(p.claimRoute(Player.PassengerColor.Rainbow, 1, Player.PassengerColor.Blue));

    assertFalse(p.claimRoute(Player.PassengerColor.Black, 2, Player.PassengerColor.Blue));
    assertFalse(p.claimRoute(Player.PassengerColor.Red, 10, Player.PassengerColor.Red));

    //    Blue, Black, Red, Rainbow
    System.out.println(Arrays.toString(p.playerCards));
    for (int c : p.playerCards) {
      assertTrue(c >= 0);
    }
  }
}
```

How to test this:
- Special-Cases?
- Expected Cases?
- Boundary Cases?

Rainbow Cards

- From the rule book:

  To claim a route, a player must play a set of cards equal to the number of spaces in the route. A set of cards must be of the same type. Most routes require a specific type of set. For example a Blue route must be claimed using blue-colored Passenger Car cards. Some routes – those that are Gray colored – can be claimed using a set of cards of any one color.

  [...] **Locomotives are Multi-colored and act as a wild card that can be part of any set of cards when claiming a route**. [...]

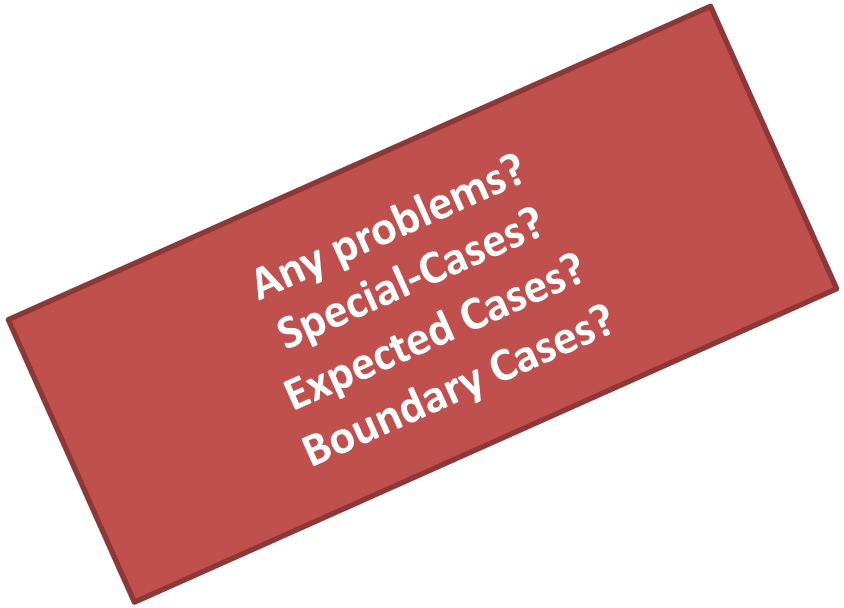
- So lets add this functionality

A naïve Example

```
public boolean claimRoute(
        PassengerColor payByColor,
        int routeCost,
        PassengerColor routeColor
    ) {
  int currentCards = playerCards[payByColor.ordinal()];
  int rainBowCards = playerCards[PassengerColor.Rainbow.ordinal()];

  // Pay for a route between two adjacent cities on the map
  if ( (payByColor == routeColor
        || payByColor == PassengerColor.Rainbow
        || routeColor == PassengerColor.Rainbow)) {
    // no rainbow cards needed
    if (currentCards >= routeCost) {
      playerCards[payByColor.ordinal()] -= routeCost;
      return true;
    }
    // rainbow cards are needed
    else if (currentCards + rainBowCards >= routeCost){
      playerCards[payByColor.ordinal()]= 0;
      playerCards[PassengerColor.Rainbow.ordinal()] -= (routeCost - currentCards);
      return true;
    }
  }

  // we cannot buy the route
  return false;
}
```



Demo: Unit Tests

Repository:
https://github.com/hu-berlin-semesterprojekte/cidemo

Continuous Integration (CI)

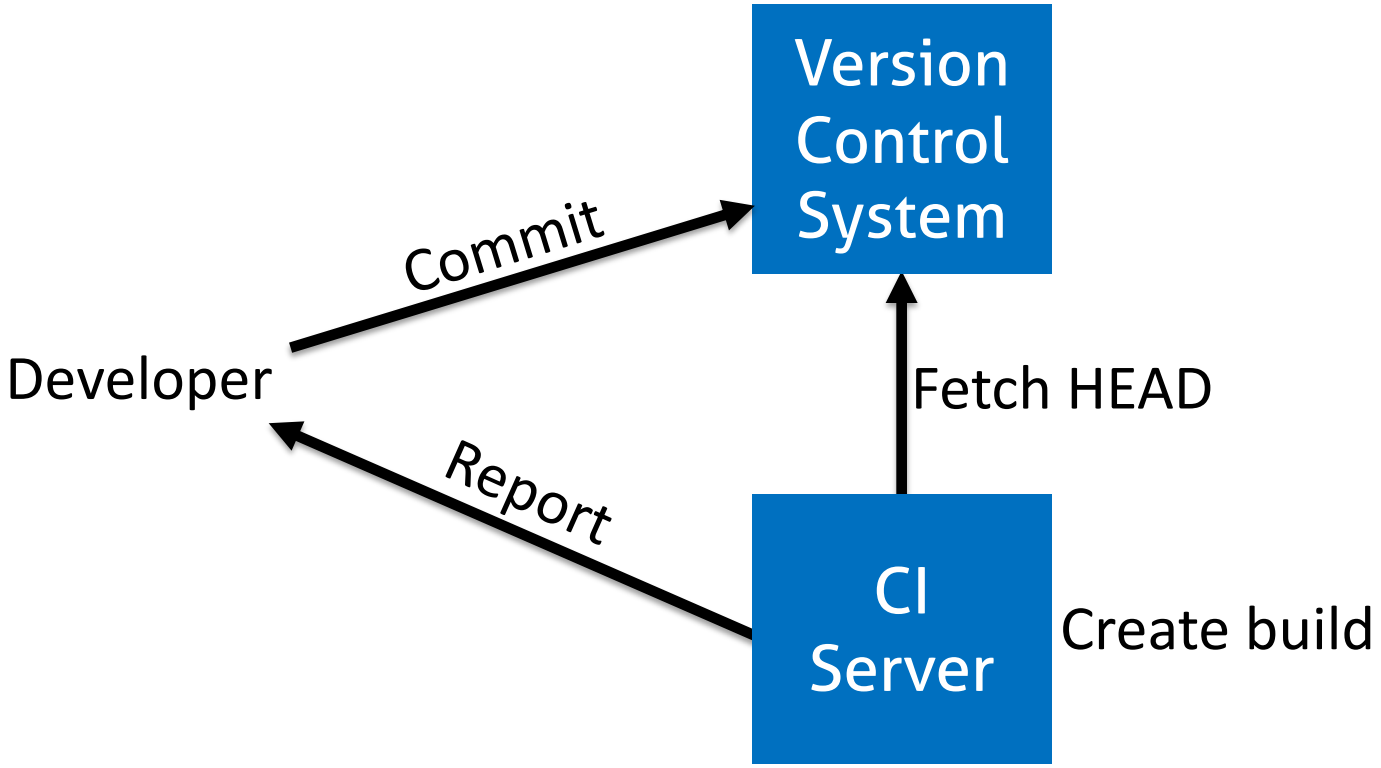
- automatically test and merge all units into an integrated software (multiple times a day)

- every change (e.g., git push) in the software triggers a new build
- unit tests are executed to determine the success of a build
- gives feedback in form of reports

- requires version control and build automation for downloading dependencies, compiling code, and running tests
  - build automation tools for Java: Maven, Ant, Gradle

- builds can succeed or fail

CI in Practice



Advantages of Continuous Integreation

with continuous integration, we …

- prevent “integration hell” early
- always know the latest stable version of our software
- instant feedback if a developer‘s work in progress breaks the stable version
- can automatically test different setups
  - different databases
  - multiple versions of 3rd party libraries
  - different configurations

Best Practices

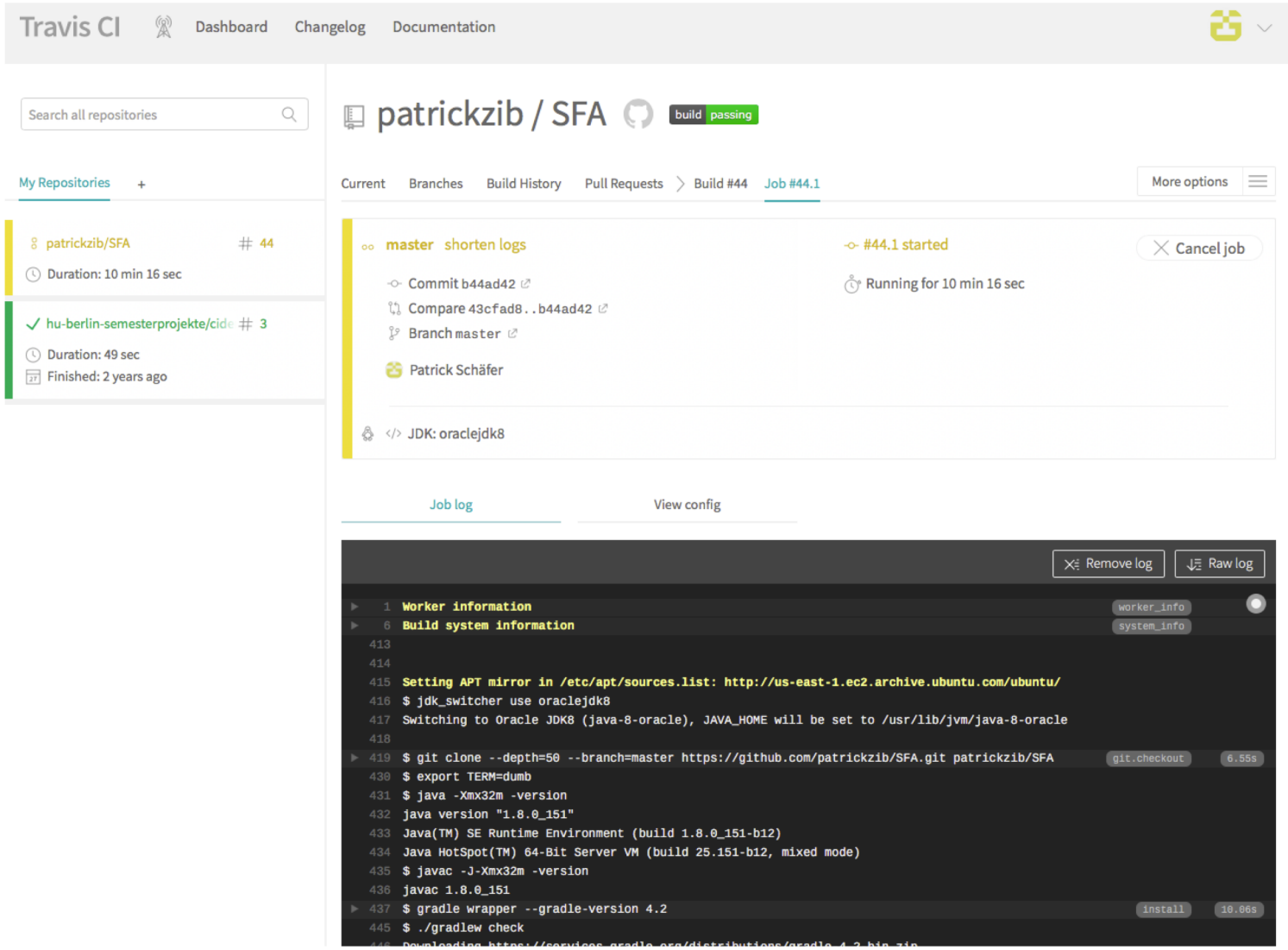
1. design meaningful unit tests for your software modules
2. commit frequently; keep iterations small
3. keep your tests fast; keep the build fast
4. don’t (ever) commit into a stable branch when the build is broken

Travis CI

- open-source continuous integration service / server
- website: https://travis-ci.org
- coupled with GitHub

- easy to set up:
  1. sign in using your GitHub account
  2. select repositories that Travis should build
- build is configurable via .travis.yml file
  - YAML is a popular data serialization file format, similar to XML or JSON

Travis CI


Travis CI
Dashboard
Changelog
Documentation
Search all repositories
My Repositories
patrickzib/SFA
44
Duration: 10 min 16 sec
hu-berlin-semesterprojekte/cid
3
Duration: 49 sec
Finished: 2 years ago
patrickzib / SFA
build passing
Current
Branches
Build History
Pull Requests
Build #44
Job #44.1
More options
master shorten logs
#44.1 started
Cancel job
Commit b44ad42
Compare 43cfad8..b44ad42
Branch master
Running for 10 min 16 sec
Patrick Schäfer
JDK: oraclejdk8
Job log
View config
Remove log
Raw log
1 Worker information worker_info
6 Build system information system_info
413
414
415 Setting APT mirror in /etc/apt/sources.list: http://us-east-1.ec2.archive.ubuntu.com/ubuntu/
416 $ jdk_switcher use oraclejdk8
417 Switching to Oracle JDK8 (java-8-oracle), JAVA_HOME will be set to /usr/lib/jvm/java-8-oracle
418
419 $ git clone --depth=50 --branch=master https://github.com/patrickzib/SFA.git patrickzib/SFA git.checkout 6.55s
430 $ export TERM=dumb
431 $ java -Xmx32m -version
432 java version "1.8.0_151"
433 Java(TM) SE Runtime Environment (build 1.8.0_151-b12)
434 Java HotSpot(TM) 64-Bit Server VM (build 25.151-b12, mixed mode)
435 $ javac -J-Xmx32m -version
436 javac 1.8.0_151
437 $ gradle wrapper --gradle-version 4.2 install 10.06s
445 $ ./gradlew check

Travis CI: Build History

Using Travis YML

1. Write Unit Tests

2. Configure build automation

   pom.xml for Maven (Java)

   build.gradle for Gradle (Java)

   ...

3. Configure travis to use build automation

   .travis.yml:

   ```
   language: java
   jdk: - oraclejdk8
   sudo: false
   script: mvn clean verify
   ```

Demo: Continuous Integration

Repository:
https://github.com/hu-berlin-semesterprojekte/cidemo

Further Reading

- unit tests in Java using JUnit:
  http://www.frankwestphal.de/UnitTestingmitJUnit.html
- build automation in Java using Maven:
  https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

- unit tests in Unity using Unity Test Tools:
  https://unity3d.com/learn/tutorials/topics/production/unity-test-tools
- continuous integration in Unity using GitHub and Travis CI:
  https://stablekernel.com/continuous-integration-for-unity-5-using-travisci/

Next steps

- familiarize yourself with unit tests & continuous integration
  - further reading (→ last slide)
  - start testing and integrating (→ user story “Continuous Integration”)
- this week (w/o POs)
  - finalize tasks in your sprint backlog (incl. tests, code review)
  - mid-week: technical refinement for the new user stories
- Next Monday, 13:30
  - Sprint #1 Review Meeting; bring a laptop & presentable prototype
  - Sprint #2 Sprint planning; kickoff; present your Sprint Backlog
- Further technical talks…
- Questions?