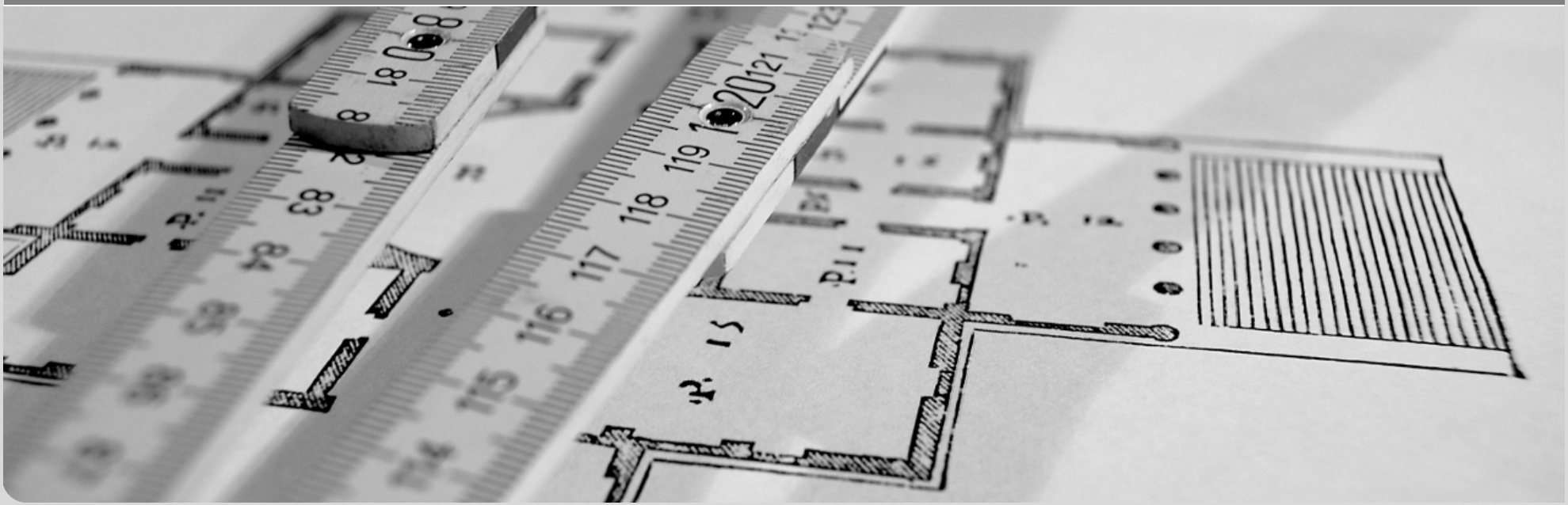
Karlsruher Institut für Technologie

Softwaretechnik 1 - 6. Tutorium

Tutorium 17

Felix Bachmann | 23.07.2019

KIT - INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION (IPD)



1. Orga
   - Feedback
2. Werbung
3. Testen
   - Definitionen
4. Wiederholung und Klausuraufgaben
   - Planung & Definition
   - Entwurf
   - Implementierung
   - Testen
   - Abnahme, Einsatz & Wartung
   - Rest
5. Ende
   - Feierabend

Termine

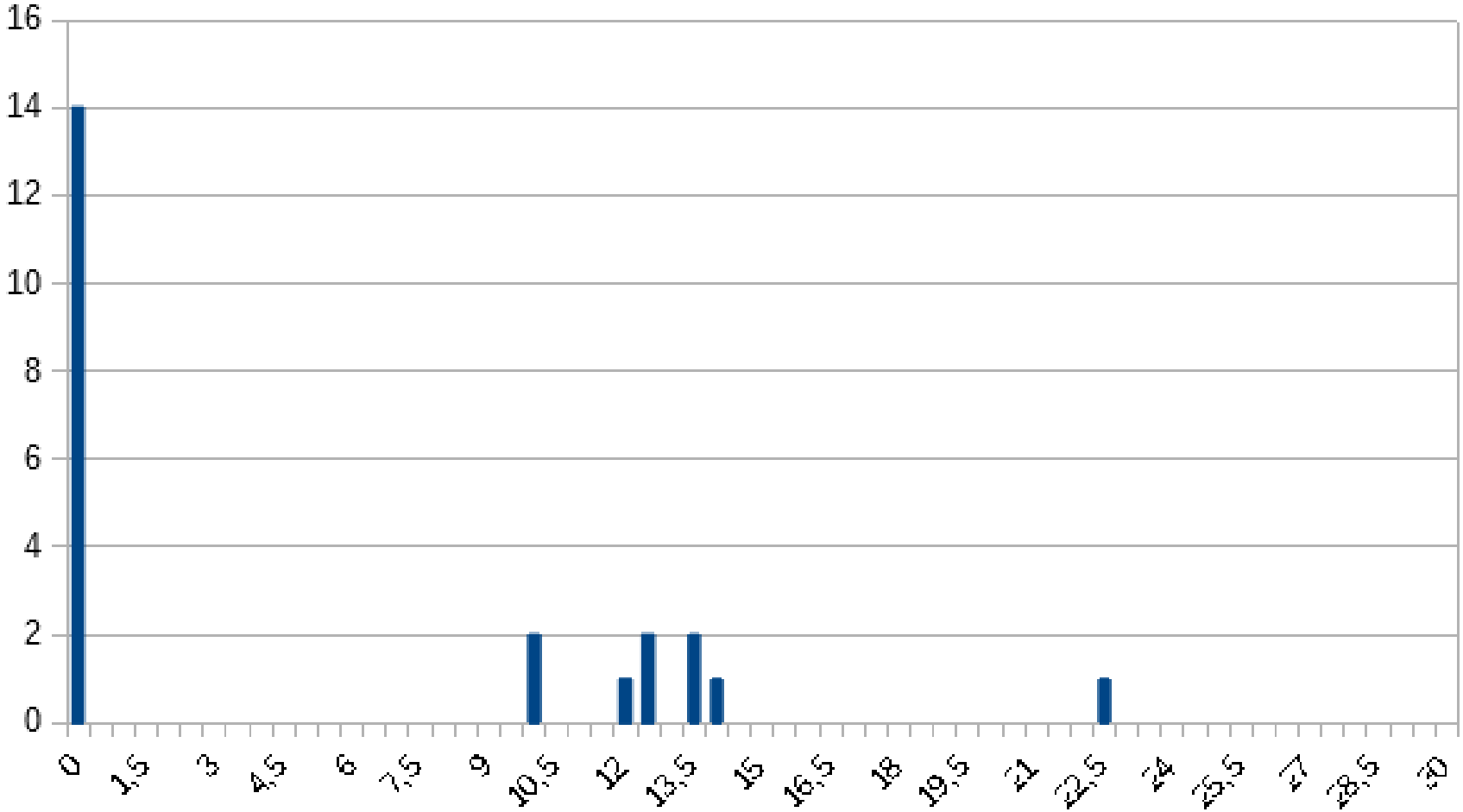
Klausur, Übungsschein

- Hauptklausur am 01.08.19, 11:00
- Nachklausur wahrscheinlich am 07.10.19
- Anmeldung sollte nun für alle möglich sein

Nicht abgeholte Übungsblätter

- bringe ich zu den Übungsleitern (R346, in der Nähe vom Holzkasten)
  - siehe `https://ps.ipd.kit.edu/3273.php`
- falls ihr die noch haben wollt, holt sie euch dort ab
- Übungsleiter beißen (meistens) nicht

6. Übungsblatt Statistik



Häufige Fehler

Aufgabe 1+2: Parallelisierung

Häufige Fehler

Aufgabe 1+2: Parallelisierung

- 4+1 Abgabe…

Häufige Fehler

Aufgabe 1+2: Parallelisierung

- 4+1 Abgabe...
- Anzahl Prozessoren (inkl. HT) berechnen
  - `int nproc = Runtime.getRuntime().availableProcessors();`

Häufige Fehler

Aufgabe 1+2: Parallelisierung

- 4+1 Abgabe...
- Anzahl Prozessoren (inkl. HT) berechnen
  - `int nproc = Runtime.getRuntime().availableProcessors();`

Aufgabe 3: Äquivalenzklassen-Tests

- Grenzwerte „möglichst nah“ an Grenze
- z.B. ungültiger Betrag: Grenzwert sowas wie -1 oder -0.01, nicht -20

Häufige Fehler

Aufgabe 4: Kontrollfluss-orientiertes Testen

Häufige Fehler

Aufgabe 4: Kontrollfluss-orientiertes Testen

- alles außer Kontrollfluss-Zeug so lassen wie es ist
  - insb. auch `return x;` (nicht zusammenfassen!)

Häufige Fehler

Aufgabe 4: Kontrollfluss-orientiertes Testen

- alles außer Kontrollfluss-Zeug so lassen wie es ist
  - insb. auch `return x;` (nicht zusammenfassen!)
  - ansonsten bei Überdeckung gemogelt
    - wo hört man beim Zusammenfassen auf? sind auch alle `c++` ein Block?
    - ein `return`-Block betreten reicht dann schon für Überdeckung aller `returns`
    - dann können wir es gleich lassen :D

Häufige Fehler

Aufgabe 4: Kontrollfluss-orientiertes Testen

- alles außer Kontrollfluss-Zeug so lassen wie es ist
  - insb. auch `return x;` (nicht zusammenfassen!)
  - ansonsten bei Überdeckung gemogelt
    - wo hört man beim Zusammenfassen auf? sind auch alle `c++` ein Block?
    - ein `return`-Block betreten reicht dann schon für Überdeckung aller `returns`
    - dann können wir es gleich lassen :D
  - Code 1:1 übernehmen, `;` nicht vergessen

Häufige Fehler

Aufgabe 5: Codeinspektion

Häufige Fehler

Aufgabe 5: Codeinspektion

- JavaDoc bei `@Override` nicht nötig
  - solange Unterklasse nicht etwas nennenswert anders macht
    - sollte sie aber eh nicht (Substitutionsprinzip undso)

Häufige Fehler

Aufgabe 5: Codeinspektion

- JavaDoc bei `@Override` nicht nötig
  - solange Unterklasse nicht etwas nennenswert anders macht
    - sollte sie aber eh nicht (Substitutionsprinzip undso)
- `throws` in Signatur und JavaDoc
  - nur bei `CheckedExceptions` nötig
  - bei `UncheckedExceptions` = Unterklassen von `RuntimeException` **nicht**
  - z.B. `NullPointerException`, `IllegalArgumentException`, `ArrayIndexOutOfBoundsException`

Häufige Fehler

Aufgabe 5: Codeinspektion

- JavaDoc bei `@Override` nicht nötig
  - solange Unterklasse nicht etwas nennenswert anders macht
    - sollte sie aber eh nicht (Substitutionsprinzip undso)
- `throws` in Signatur und JavaDoc
  - nur bei `CheckedExceptions` nötig
  - bei `UncheckedExceptions` = Unterklassen von `RuntimeException` **nicht**
  - z.B. `NullPointerException`, `IllegalArgumentException`, `ArrayIndexOutOfBoundsException`
- `data = new double[rows][cols];` füllt das Array implizit mit Nullen
  - also keine Verletzung des Verhaltens wie in JavaDoc beschrieben

Häufige Fehler

Aufgabe 5: Codeinspektion

- JavaDoc bei `@Override` nicht nötig
  - solange Unterklasse nicht etwas nennenswert anders macht
    - sollte sie aber eh nicht (Substitutionsprinzip undso)
- `throws` in Signatur und JavaDoc
  - nur bei `CheckedExceptions` nötig
  - bei `UncheckedExceptions` = Unterklassen von `RuntimeException` **nicht**
  - z.B. `NullPointerException`, `IllegalArgumentException`, `ArrayIndexOutOfBoundsException`
- `data = new double[rows][cols];` füllt das Array implizit mit Nullen
  - also keine Verletzung des Verhaltens wie in JavaDoc beschrieben
- `final`-Variablen können erst deklariert werden und im Konstruktor initialisiert

Zusatztutorium

Was?

- Übergang von SWT1 zu PSE
- in kleinen Gruppen ein 2D-Jump'n'Run erweitern

Zusatztutorium

Was?

- Übergang von SWT1 zu PSE
- in kleinen Gruppen ein 2D-Jump'n'Run erweitern
- <zeige werbungs-slides>

Zusatztutorium

Was?

- Übergang von SWT1 zu PSE
- in kleinen Gruppen ein 2D-Jump'n'Run erweitern
- <zeige werbungs-slides>

Wann?

- steht noch nicht 100%ig fest
- vermutlich 1./2. VL-Woche vom nächsten Semester
- **bevor** das PSE losgeht

Zusatztutorium

Was?

- Übergang von SWT1 zu PSE
- in kleinen Gruppen ein 2D-Jump'n'Run erweitern
- <zeige werbungs-slides>

Wann?

- steht noch nicht 100%ig fest
- vermutlich 1./2. VL-Woche vom nächsten Semester
- **bevor** das PSE losgeht

Warum?

- SWT ist dann schon über 2 Monate her ;)
- und außerdem: warum nicht?

Arten von Fehlern

- Was verursacht was?
- Defekt, Irrtum, Versagen

Arten von Fehlern

- Was verursacht was?
- Defekt, Irrtum, Versagen
- Irrtum (Aktion) → Defekt (Bug) → Ausfall (failure)
- dumme Eselsbrücke: IDA

Arten von Fehlern

- Was verursacht was?
- Defekt, Irrtum, Versagen
- Irrtum (Aktion) → Defekt (Bug) → Ausfall (failure)
- dumme Eselsbrücke: IDA
- „Testing shows the presence of bugs, not their absence.“ (Edsger W. Dijkstra)

Arten von Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)

Arten von Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing

Arten von Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
  - kontrollflussorientiert
  - datenflussorientiert
- black box testing

Arten von Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
  - kontrollflussorientiert
  - datenflussorientiert
- black box testing
  - funktionale Tests

Arten von Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
  - kontrollflussorientiert
  - datenflussorientiert
- black box testing
  - funktionale Tests
  - Leistungstests

Arten von Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
  - kontrollflussorientiert
  - datenflussorientiert
- black box testing
  - funktionale Tests
  - Leistungstests

Statische Verfahren

- Inspektion („scharfes Hinsehen“, Code Review, etc.)

Arten von Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
  - kontrollflussorientiert
  - datenflussorientiert
- black box testing
  - funktionale Tests
  - Leistungstests

Statische Verfahren

- Inspektion („scharfes Hinsehen“, Code Review, etc.)
- statische Analyse mit Tools (formale Verifikation, SpotBugs, etc.)

Arten von Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
  - kontrollflussorientiert
  - datenflussorientiert
- black box testing
  - funktionale Tests
  - Leistungstests

Statische Verfahren

- Inspektion („scharfes Hinsehen“, Code Review, etc.)
- statische Analyse mit Tools (formale Verifikation, SpotBugs, etc.)
- Programm wird nicht ausgeführt!

Disclaimer

- Ich kenne die Klausur auch nicht!

Disclaimer

- Ich kenne die Klausur auch nicht!
  $\Longrightarrow$ alles, was ich zum Inhalt der Klausur sage ist Spekulation
  - basierend auf Altklausuren

Disclaimer

- Ich kenne die Klausur auch nicht!
  $\Longrightarrow$ alles, was ich zum Inhalt der Klausur sage ist Spekulation
  - basierend auf Altklausuren
- kein Anspruch auf Vollständigkeit der Wiederholung

Die typische SWT-Klausur

1. Aufgabe 1: Aufwärmen
   - Wahr-/Falsch-Fragen
     - ein paar gesammelt auf
       `www.github.com/malluce/swt1-tut/multiple_choice.txt`
   - Wissensfragen

Die typische SWT-Klausur

1. Aufgabe 1: Aufwärmen
   - Wahr-/Falsch-Fragen
     - ein paar gesammelt auf `www.github.com/malluce/swt1-tut/multiple_choice.txt`
   - Wissensfragen
2. meistens Aufgaben zu:
   - UML-Diagrammen

Die typische SWT-Klausur

1. Aufgabe 1: Aufwärmen
   - Wahr-/Falsch-Fragen
     - ein paar gesammelt auf `www.github.com/malluce/swt1-tut/multiple_choice.txt`
   - Wissensfragen
2. meistens Aufgaben zu:
   - UML-Diagrammen
   - Entwurfsmustern

Die typische SWT-Klausur

1. Aufgabe 1: Aufwärmen
   - Wahr-/Falsch-Fragen
     - ein paar gesammelt auf `www.github.com/malluce/swt1-tut/multiple_choice.txt`
   - Wissensfragen
2. meistens Aufgaben zu:
   - UML-Diagrammen
   - Entwurfsmustern
   - Parallelität

Die typische SWT-Klausur

1. Aufgabe 1: Aufwärmen
   - Wahr-/Falsch-Fragen
     - ein paar gesammelt auf `www.github.com/malluce/swt1-tut/multiple_choice.txt`
   - Wissensfragen
2. meistens Aufgaben zu:
   - UML-Diagrammen
   - Entwurfsmustern
   - Parallelität
   - Testen bzw. Qualitätssicherung (KFG!)

Die typische SWT-Klausur

1. Aufgabe 1: Aufwärmen
   - Wahr-/Falsch-Fragen
     - ein paar gesammelt auf `www.github.com/malluce/swt1-tut/multiple_choice.txt`
   - Wissensfragen
2. meistens Aufgaben zu:
   - UML-Diagrammen
   - Entwurfsmustern
   - Parallelität
   - Testen bzw. Qualitätssicherung (KFG!)
   - Rest (z.B. Objektorientierung, Git/Versionskontrolle, Prozessmodelle. . .)

Die typische SWT-Klausur

1. Aufgabe 1: Aufwärmen
   - Wahr-/Falsch-Fragen
     - ein paar gesammelt auf `www.github.com/malluce/swt1-tut/multiple_choice.txt`
   - Wissensfragen
2. meistens Aufgaben zu:
   - UML-Diagrammen
   - Entwurfsmustern
   - Parallelität
   - Testen bzw. Qualitätssicherung (KFG!)
   - Rest (z.B. Objektorientierung, Git/Versionskontrolle, Prozessmodelle...)

- bisher: $\frac{1}{3} \pm \epsilon$ der Punkte reichen zum Bestehen
  - kann sich eventuell ändern, wegen längerer Bearbeitungszeit

Aufwärmaufgabe

Karlsruher Institut für Technologie

Hauptklausur SS2011 A1

Planung & Definition

- Lastenheft, Pflichtenheft

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen
  - Beispiele geben

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen
  - Beispiele geben
- UML-Diagramme

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen
  - Beispiele geben
- UML-Diagramme
  - Klassendiagramm

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen
  - Beispiele geben
- UML-Diagramme
  - Klassendiagramm
  - Aktivitäts-, Sequenz-, Zustandsdiagramm

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen
  - Beispiele geben
- UML-Diagramme
  - Klassendiagramm
  - Aktivitäts-, Sequenz-, Zustandsdiagramm
  - Anwendungsfalldiagramm

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen
  - Beispiele geben
- UML-Diagramme
  - Klassendiagramm
  - Aktivitäts-, Sequenz-, Zustandsdiagramm
  - Anwendungsfalldiagramm
  - Syntax kennen!

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen
  - Beispiele geben
- UML-Diagramme
  - Klassendiagramm
  - Aktivitäts-, Sequenz-, Zustandsdiagramm
  - Anwendungsfalldiagramm
  - Syntax kennen!
  - gegebenen Text/Code in Diagramm umwandeln

Planung & Definition

- Lastenheft, Pflichtenheft
  - Phasen zuordnen
  - Gliederung kennen
  - Beispiele geben
- UML-Diagramme
  - Klassendiagramm
  - Aktivitäts-, Sequenz-, Zustandsdiagramm
  - Anwendungsfalldiagramm
  - Syntax kennen!
  - gegebenen Text/Code in Diagramm umwandeln
  - bei Zustandsdiagramm
    - Umwandeln hierarchisch $\Leftrightarrow$ nicht-hierarchisch
    - Umwandeln parallel $\Leftrightarrow$ nicht-parallel

Zustandsdiagramm-Aufgabe

Karlsruher Institut für Technologie

Hauptklausur SS2012 A2

Entwurf

- Architekturstile

Entwurf

- Architekturstile
- **Entwurfsmuster**

Entwurf

- Architekturstile
- **Entwurfsmuster**
  - möglichst viele, bestenfalls alle kennen und verstehen

Entwurf

- Architekturstile
- **Entwurfsmuster**
  - möglichst viele, bestenfalls alle kennen und verstehen
  - Kategorien kennen

Entwurf

- Architekturstile
- **Entwurfsmuster**
  - möglichst viele, bestenfalls alle kennen und verstehen
  - Kategorien kennen
  - Klassendiagramm hinzeichnen (auch auf Szenario angepasst)

Entwurf

- Architekturstile
- **Entwurfsmuster**
  - möglichst viele, bestenfalls alle kennen und verstehen
  - Kategorien kennen
  - Klassendiagramm hinzeichnen (auch auf Szenario angepasst)
  - aus Klassendiagrammen/Code Entwurfsmuster erkennen

Entwurf

- Architekturstile
- **Entwurfsmuster**
  - möglichst viele, bestenfalls alle kennen und verstehen
  - Kategorien kennen
  - Klassendiagramm hinzeichnen (auch auf Szenario angepasst)
  - aus Klassendiagrammen/Code Entwurfsmuster erkennen
  - Code für Muster schreiben

Entwurf

- Architekturstile
- **Entwurfsmuster**
  - möglichst viele, bestenfalls alle kennen und verstehen
  - Kategorien kennen
  - Klassendiagramm hinzeichnen (auch auf Szenario angepasst)
  - aus Klassendiagrammen/Code Entwurfsmuster erkennen
  - Code für Muster schreiben
  - Code-Schnipsel auf mögliche Verbesserung durch EM untersuchen

Entwurfsmuster-Aufgabe

Karlsruher Institut für Technologie

Hauptklausur SS2010 A3

Implementierung

- UML-Abbildung

Implementierung

- UML-Abbildung
- **Parallelität**

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java
  - critical sections/ race conditions

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java
  - critical sections/ race conditions
  - deadlock

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java
  - critical sections/ race conditions
  - deadlock
  - Monitore, wait & notify

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java
  - critical sections/ race conditions
  - deadlock
  - Monitore, wait & notify
  - Semaphore

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java
  - critical sections/ race conditions
  - deadlock
  - Monitore, wait & notify
  - Semaphore
- Rechnungen können (Speedup, Amdahls Law (**Tafel**))

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java
  - critical sections/ race conditions
  - deadlock
  - Monitore, wait & notify
  - Semaphore
- Rechnungen können (Speedup, Amdahls Law (**Tafel**))
- gegebenen Code thread-safe machen

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java
  - critical sections/ race conditions
  - deadlock
  - Monitore, wait & notify
  - Semaphore
- Rechnungen können (Speedup, Amdahls Law (**Tafel**))
- gegebenen Code thread-safe machen
- Lösungsvorschläge zur Synchronisation bewerten

Implementierung

- UML-Abbildung
- **Parallelität**
  - grundlegendes Prinzip
  - in Java
  - critical sections/ race conditions
  - deadlock
  - Monitore, wait & notify
  - Semaphore
- Rechnungen können (Speedup, Amdahls Law (**Tafel**))
- gegebenen Code thread-safe machen
- Lösungsvorschläge zur Synchronisation bewerten
- eigenen Code schreiben

Parallelität-Aufgabe

Karlsruher Institut für Technologie

Hauptklausur SS2011 A3

java.util.concurrent

- thread-safe Datenstrukturen („Concurrent Collections“)
  - kann man in parallelen Programmen verwenden, falls mehrere Threads drauf zugreifen
  - „normale“ Collections sind nicht thread-safe
    - was soll passieren, wenn ein Thread über Daten iteriert, ein anderer sie gleichzeit modifiziert?
    - sind „fail-fast“: werfen `ConcurrentModificationException`

java.util.concurrent

- thread-safe Datenstrukturen („Concurrent Collections“)
  - kann man in parallelen Programmen verwenden, falls mehrere Threads drauf zugreifen
  - „normale“ Collections sind nicht thread-safe
    - was soll passieren, wenn ein Thread über Daten iteriert, ein anderer sie gleichzeit modifiziert?
    - sind „fail-fast“: werfen `ConcurrentModificationException`
  - statt `HashMap`, verwende `ConcurrentHashMap`
  - statt `ArrayList`, verwende `CopyOnWriteArrayList`
    - Achtung: kopiert bei jeder Veränderung das gesamte Array
    - daher nur benutzen, wenn öfter gelesen/iteriert wird als geschrieben
  - usw.

java.util.concurrent

- thread-safe Datenstrukturen („Concurrent Collections“)
  - kann man in parallelen Programmen verwenden, falls mehrere Threads drauf zugreifen
  - „normale“ Collections sind nicht thread-safe
    - was soll passieren, wenn ein Thread über Daten iteriert, ein anderer sie gleichzeit modifiziert?
    - sind „fail-fast“: werfen `ConcurrentModificationException`
  - statt `HashMap`, verwende `ConcurrentHashMap`
  - statt `ArrayList`, verwende `CopyOnWriteArrayList`
    - Achtung: kopiert bei jeder Veränderung das gesamte Array
    - daher nur benutzen, wenn öfter gelesen/iteriert wird als geschrieben
  - usw.
  - Achtung: `Concurrent != Synchronized`
    - Datenstrukturen lassen ggf. mehrere Threads zu (JavaDoc lesen)

java.util.concurrent

- thread-safe Datenstrukturen („Concurrent Collections“)
  - kann man in parallelen Programmen verwenden, falls mehrere Threads drauf zugreifen
  - „normale“ Collections sind nicht thread-safe
    - was soll passieren, wenn ein Thread über Daten iteriert, ein anderer sie gleichzeit modifiziert?
    - sind „fail-fast“: werfen `ConcurrentModificationException`
  - statt `HashMap`, verwende `ConcurrentHashMap`
  - statt `ArrayList`, verwende `CopyOnWriteArrayList`
    - Achtung: kopiert bei jeder Veränderung das gesamte Array
    - daher nur benutzen, wenn öfter gelesen/iteriert wird als geschrieben
  - usw.
  - Achtung: `Concurrent != Synchronized`
    - Datenstrukturen lassen ggf. mehrere Threads zu (JavaDoc lesen)
  - Code-Beispiel

java.util.concurrent

- `Executor`, `ExecutorService`, `Executors`
  - abstrahiert von Thread-Management
  - `Executor` bietet Methode `void execute(Runnable runnable)`

java.util.concurrent

- `Executor`, `ExecutorService`, `Executors`
  - abstrahiert von Thread-Management
  - `Executor` bietet Methode `void execute(Runnable runnable)`
  - `ExecutorService extends Executor` bietet weitere Methoden

java.util.concurrent

- `Executor`, `ExecutorService`, `Executors`
  - abstrahiert von Thread-Management
  - `Executor` bietet Methode `void execute(Runnable runnable)`
  - `ExecutorService extends Executor` bietet weitere Methoden
  - `Executors` bietet Methoden zum Erstellen von `ExecutorService`-Impl.
    - `newFixedThreadPool(int nThreads)`
    - `newCachedThreadPool()`

java.util.concurrent

- `Executor`, `ExecutorService`, `Executors`
  - abstrahiert von Thread-Management
  - `Executor` bietet Methode `void execute(Runnable runnable)`
  - `ExecutorService extends Executor` bietet weitere Methoden
  - `Executors` bietet Methoden zum Erstellen von `ExecutorService`-Impl.
    - `newFixedThreadPool(int nThreads)`
    - `newCachedThreadPool()`
- `Callable<T>`, `Future<T>`
  - Problem an `Runnable`: keine Rückgabe möglich (`void`)

java.util.concurrent

- `Executor`, `ExecutorService`, `Executors`
  - abstrahiert von Thread-Management
  - `Executor` bietet Methode `void execute(Runnable runnable)`
  - `ExecutorService extends Executor` bietet weitere Methoden
  - `Executors` bietet Methoden zum Erstellen von `ExecutorService`-Impl.
    - `newFixedThreadPool(int nThreads)`
    - `newCachedThreadPool()`
- `Callable<T>`, `Future<T>`
  - Problem an `Runnable`: keine Rückgabe möglich (`void`)
  - `Callable<T>#call()` gibt `T` zurück

java.util.concurrent

- `Executor`, `ExecutorService`, `Executors`
  - abstrahiert von Thread-Management
  - `Executor` bietet Methode `void execute(Runnable runnable)`
  - `ExecutorService extends Executor` bietet weitere Methoden
  - `Executors` bietet Methoden zum Erstellen von `ExecutorService`-Impl.
    - `newFixedThreadPool(int nThreads)`
    - `newCachedThreadPool()`
- `Callable<T>`, `Future<T>`
  - Problem an `Runnable`: keine Rückgabe möglich (`void`)
  - `Callable<T>#call()` gibt `T` zurück
  - Problem: „man weiß nicht wann `call()` fertig“
  - daher: `Future<T>` ist Abstraktion der Rückgabe
    - mittel `Future<T>#get()` kann man auf Ergebnis warten

java.util.concurrent

- `Executor`, `ExecutorService`, `Executors`
  - abstrahiert von Thread-Management
  - `Executor` bietet Methode `void execute(Runnable runnable)`
  - `ExecutorService extends Executor` bietet weitere Methoden
  - `Executors` bietet Methoden zum Erstellen von `ExecutorService`-Impl.
    - `newFixedThreadPool(int nThreads)`
    - `newCachedThreadPool()`
- `Callable<T>`, `Future<T>`
  - Problem an `Runnable`: keine Rückgabe möglich (`void`)
  - `Callable<T>#call()` gibt `T` zurück
  - Problem: „man weiß nicht wann `call()` fertig“
  - daher: `Future<T>` ist Abstraktion der Rückgabe
    - mittel `Future<T>#get()` kann man auf Ergebnis warten
  - z.B.: `ExecutorService#submit(Callable<T> task)` gibt `Future<T>` zurück

`java.util.concurrent.atomic`

- bietet atomare Datentypen
- „lock-free thread-safe programming on single variables“
- erlaubt z.B. atomares `i++;`
  - normalerweise: `tmp = i; tmp = tmp + 1; i = tmp;`
  - problematisch bei mehreren Threads (s. Bsp. letztes Tut)
- Bsp: `AtomicInteger#incrementAndGet()`

`java.util.concurrent.atomic`

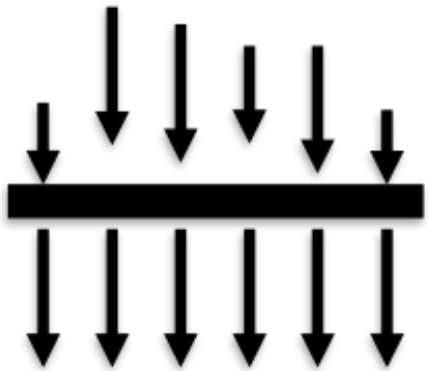
- bietet atomare Datentypen
- „lock-free thread-safe programming on single variables“
- erlaubt z.B. atomares `i++;`
  - normalerweise: `tmp = i; tmp = tmp + 1; i = tmp;`
  - problematisch bei mehreren Threads (s. Bsp. letztes Tut)
- Bsp: `AtomicInteger#incrementAndGet()`
- Code-Beispiel

Semaphoren

- `Semaphore(int n)`
  - kritische Abschnitte mit Kapazität n statt 1
    - ein kritischer Abschnitt kann von n Threads betreten werden
    - nützlich, wenn z.B. n Resourcen verarbeitet werden sollen
  - `aquire()`: versuchen reinzukommen, blockieren falls nicht möglich
  - `tryAquire()` wie `aquire()`, aber nicht blockierend
  - `release()` gibt einen „Platz“ wieder frei

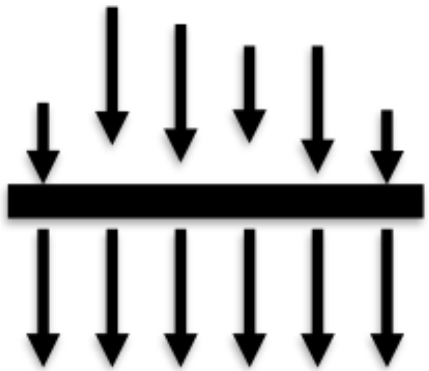
CyclicBarrier

- `CyclicBarrier(int n)`
  - `await()` blockiert aufrufenden Thread
  - falls `await()` n-mal aufgerufen, dürfen alle Threads weitermachen
  - `Cyclic` = Barriere kann danach wiederbenutzt werden



CyclicBarrier

- `CyclicBarrier(int n)`
  - `await()` blockiert aufrufenden Thread
  - falls `await()` n-mal aufgerufen, dürfen alle Threads weitermachen
  - `Cyclic` = Barriere kann danach wiederbenutzt werden



Code-Beispiel!

Testen

- Testphasen
  - Komponententest, Integrationstest, Systemtest, Abnahmetest

Testen

- Testphasen
  - Komponententest, Integrationstest, Systemtest, Abnahmetest
- Testverfahren
  - **Kontrollflussgraph**

Testen

- Testphasen
  - Komponententest, Integrationstest, Systemtest, Abnahmetest
- Testverfahren
  - **Kontrollflussgraph !!!**
    - absolute Standardaufgabe

Testen

- Testphasen
  - Komponententest, Integrationstest, Systemtest, Abnahmetest
- Testverfahren
  - **Kontrollflussgraph !!!**
    - absolute Standardaufgabe
    - in 33 von 36 der letzten Altklausuren
    - in allen Klausuren seit SS14
    - im Schlaf können (“Schema F“, lässt sich sehr gut üben. . .)

Testen

- Testphasen
  - Komponententest, Integrationstest, Systemtest, Abnahmetest
- Testverfahren
  - **Kontrollflussgraph !!!**
    - absolute Standardaufgabe
    - in 33 von 36 der letzten Altklausuren
    - in allen Klausuren seit SS14
    - im Schlaf können (“Schema F“, lässt sich sehr gut üben. . .)
- Testhelfer
- Definitionen kennen (Fehlerarten. . .)

KFG: Klausuraufgabe

Karlsruher Institut für Technologie

Hauptklausur SS2016 A6

Abnahme, Einsatz & Wartung

- Aufgaben der verschiedenen “Subphasen“ kennen

Abnahme, Einsatz & Wartung

- Aufgaben der verschiedenen “Subphasen“ kennen
- viel Text zum Lernen, aber nicht schwierig. . .

Abnahme, Einsatz & Wartung

- Aufgaben der verschiedenen “Subphasen“ kennen
- viel Text zum Lernen, aber nicht schwierig. . .
- Wartung vs. Pflege

Abnahme, Einsatz & Wartung

- Aufgaben der verschiedenen “Subphasen“ kennen
- viel Text zum Lernen, aber nicht schwierig. . .
- Wartung vs. Pflege
- selten eigene Aufgabe dazu, eher Ankreuzaufgaben in A1

Rest

- Schätzmethoden

Rest

- Schätzmethoden
- Prozessmodelle

Rest

- Schätzmethoden
- Prozessmodelle
- Agile Prozesse

Rest

- Schätzmethoden
- Prozessmodelle
- Agile Prozesse
- verschiedene Modelle kennen (XP, Scrum,. . .)

Rest

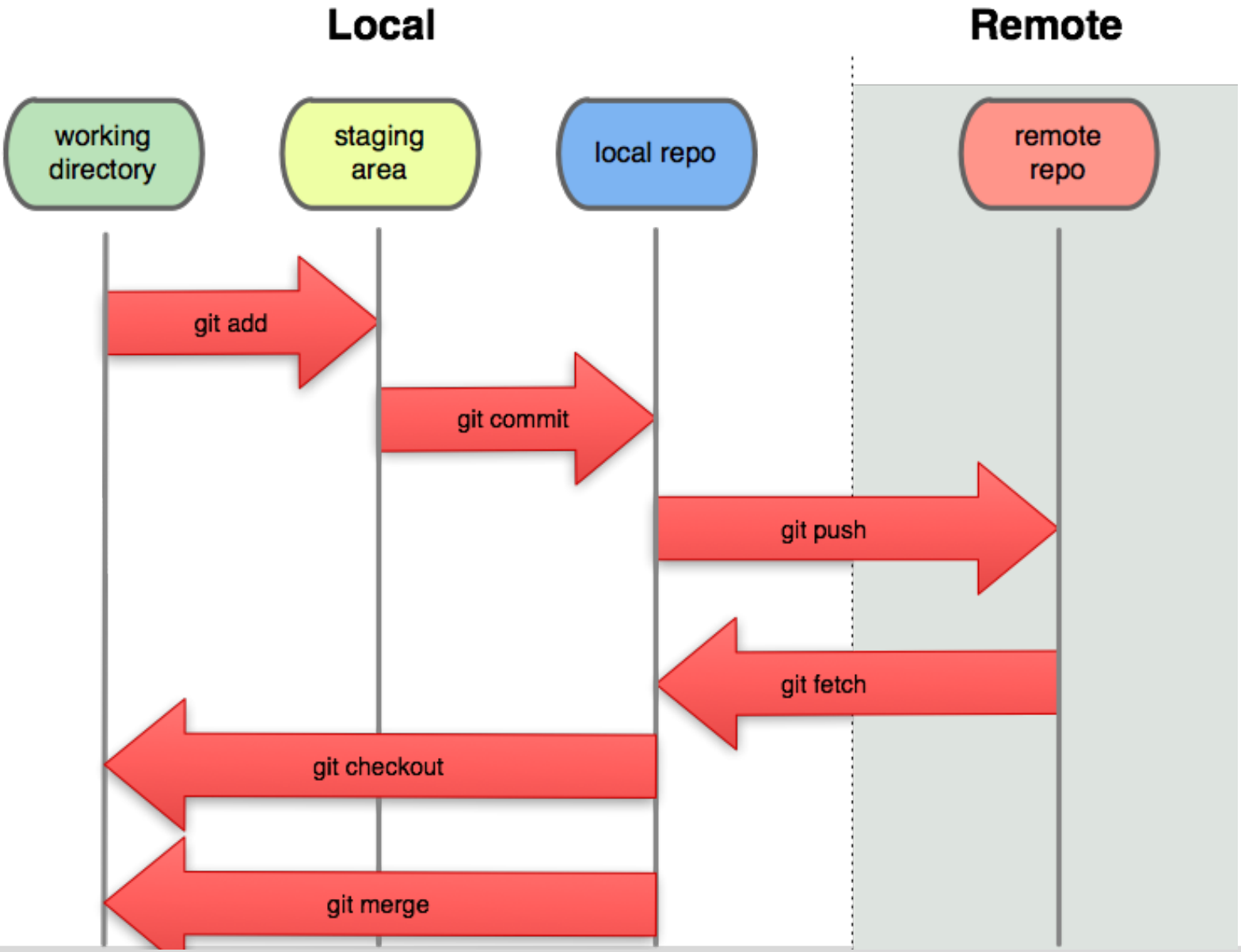
- Schätzmethoden
- Prozessmodelle
- Agile Prozesse
- verschiedene Modelle kennen (XP, Scrum,. . .)
- auch eher Ankreuzaufgaben, Wissensfragen

Git: Klausuraufgabe

Karlsruher Institut für Technologie

Hauptklausur SS18 A4

Git: Klausuraufgabe



Backup: Klassendiagramm-Aufgabe

Nachklausur SS2011 A5

Lernen

- Klausuren rechnen && Folien anschauen
  - Übung nötig, für sehr gute Note aber auch viel Wissen

Lernen

- Klausuren rechnen && Folien anschauen
  - Übung nötig, für sehr gute Note aber auch viel Wissen
- diesmal habt ihr mehr Zeit für den gleichen Umfang an Aufgaben wie in früheren Klausuren
  - vermutlich wird Bestehensgrenze hochgesetzt

Das war's dann wohl...

- danke für Eure Anwesenheit und Mitarbeit!
- Viel Erfolg bei der Klausur und im weiteren Studium! :)

SIMPLY EXPLAINED