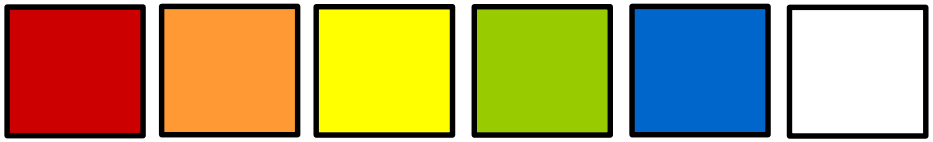


Stufe 3: Gelber Grad von CCD

Lutz Prechelt
Institut für Informatik, Freie Universität Berlin

Übersicht

Achtung:
Ein Mega-Pensum!

Prinzipien:

- Interface Segregation Prin. (ISP)
- Dependency Inversion Prin. (DIP)
- Liskov Substitution Prin. (LSP)
- Principle of Least Astonishment (LA)
- Information Hiding Prin. (IH)

Praktiken:

- Automatisierte Unit Tests
- Testattrappen (Mock-ups)
- Code Coverage Analyse
- Teilnahme an Fachveranstaltungen
- Komplexe Refaktorisierungen

Interface Segregation Principle (ISP)

- Was?
  - Gestalte Interfaces so schmal wie sinnvoll möglich
    - so dass Klienten einer Implementierungsklasse nur selten Abhängigkeiten aufbauen, die sie gar nicht benötigen
    - Implementierungsklassen realisieren oft mehrere Interfaces zugleich
- Wofür?
  - Evolvierbarkeit
  - Produktionseffizienz
- Hinderungsgrund:
  - Verlangt, sich die Bedürfnisse fast aller sinnvollen Klienten einer Gruppe von Klassen vorzustellen

Interface Segregation Principle (ISP)

- Folgen von Verletzungen:
  - Implementierende Klassen: müssen manchmal Methoden implementieren, die sie gar nicht benötigen
  - Klienten dieser Klassen: Programmierer müssen nach Interfaceauswahl noch irrelevante Methoden als solche erkennen

(bei DIP)

- Hilfreiche Techniken:
  - Rollen-Denkweise
    - Fowler: Role Interfaces
  - "-able" Benennungsweise
    - Java SE: Adjustable, Appendable, AutoClosable, Callable, Cloneable, Closable, Compilable usw. usf.
  - Bilde gedanklich Tupel von besonders eng zusammen-hängenden Methoden
    - Reichen die öfter mal aus? → bilden eigenes Interface
  - Duck Typing
    - dyn. Sprachen: gar keine separaten Interfaceobj'e; immer implementieren, was man braucht

Interface Segregation Principle (ISP): Beispiele

Freie Universität Berlin

Positivbeispiele:

- Die Mini-Interfaces in java.lang, java.util:
  - Comparable<T>
    - compareTo(T)
  - Iterable<T>
    - iterator()
  - Iterator<T>
    - hasNext(), next(), remove()
  - Readable
    - read(CharBuffer)
  - Runnable
    - run()

Negativbeispiel:

- java.sql.Statement
  - 42 Methoden!
  - Funktionsbereiche wie
    - Konfigurieren
    - Inhalt vorbereiten
    - Ausführen
    - Ergebnisse holen
    - Status abfragen
    - Schließen, Abbrechen
  - Da hätte man einiges Abspalten können

Dependency Inversion Principle (DIP)

- Was?
  - High-Level Klassen sollen nicht von Low-Level Klassen abhängig sein, sondern beide von Interfaces.
  - Vermeide statische Abhängigkeit einer Klasse von Kollaborationspartnern,
    - lege diese zur Laufzeit fest,
    - z.B. per Dependency Injection (DI): "Move Implementation Choices Up the Call Stack"
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - (Produktionseffizienz)
- Hinderungsgrund:
  - Ist erstmal umständlich
    - …zu denken (jedenfalls für Top-Down-Denker, die an Zerlegung gewöhnt sind)
      - daher kommt die Bezeichnung "Inversion"
    - …zu bauen (weil man mehr Einzelteile bekommt)
      - jedenfalls in statisch getypten Sprachen ("high ceremony")

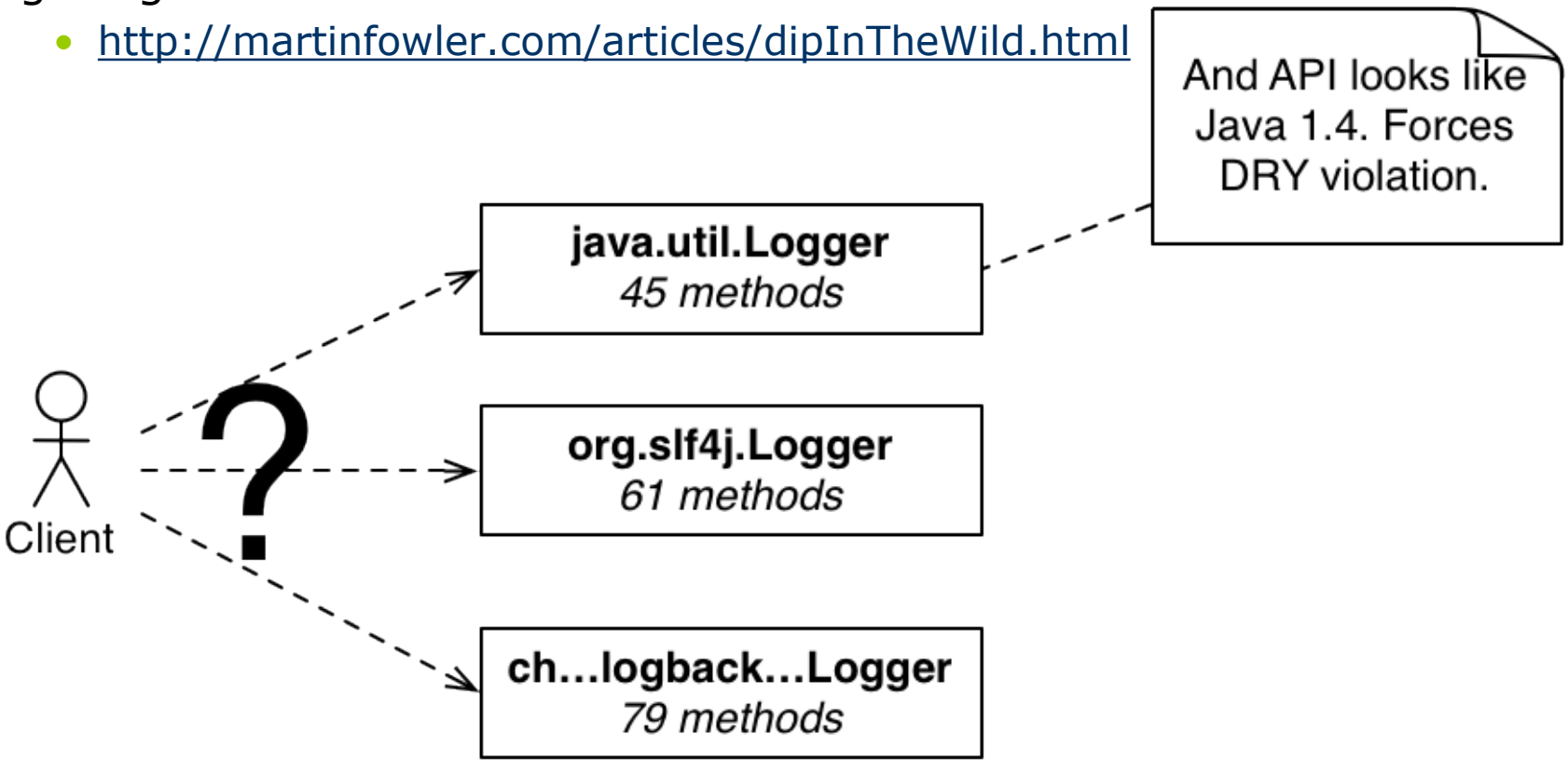
Dependency Inversion Principle (DIP): **Häh?**

Freie Universität Berlin

- Ohne Inversion:
  - Eine High-level-Klasse H kennt die zugehörige Low-Level-Klasse L namentlich und benutzt sie statisch
  - Folge: Kopplung
  - Folge: Geringe Flexibilität
    - z.B. kein Austauschen von L für Testzwecke möglich (fixer Datensatz statt DB-Abfrage u.ä.)
- Mit Inversion:
  - Hohe Flexibilität
  - Code wird weniger mit "unbefugtem Wissen" verschmutzt
    - Implementierungsklasse ist ja unbekannt!
- High-Ceremony-Techniken:
  - Abstraktionen meist durch Interfaces beschreiben
    - und bei Verwendung über Abstraktionen nachdenken, nicht über Klassen
    - siehe [DIP]
  - Implementierungsklassen injizieren (DI)
    - per Konstruktor oder Setter
    - evtl. beim Methodenaufruf
    - von Hand oder per IoC-Behälter
      - (siehe Grüner Grad)
- Low-Ceremony-Techniken:
  - Injizieren, aber per Duck Typing
    - Erklärung

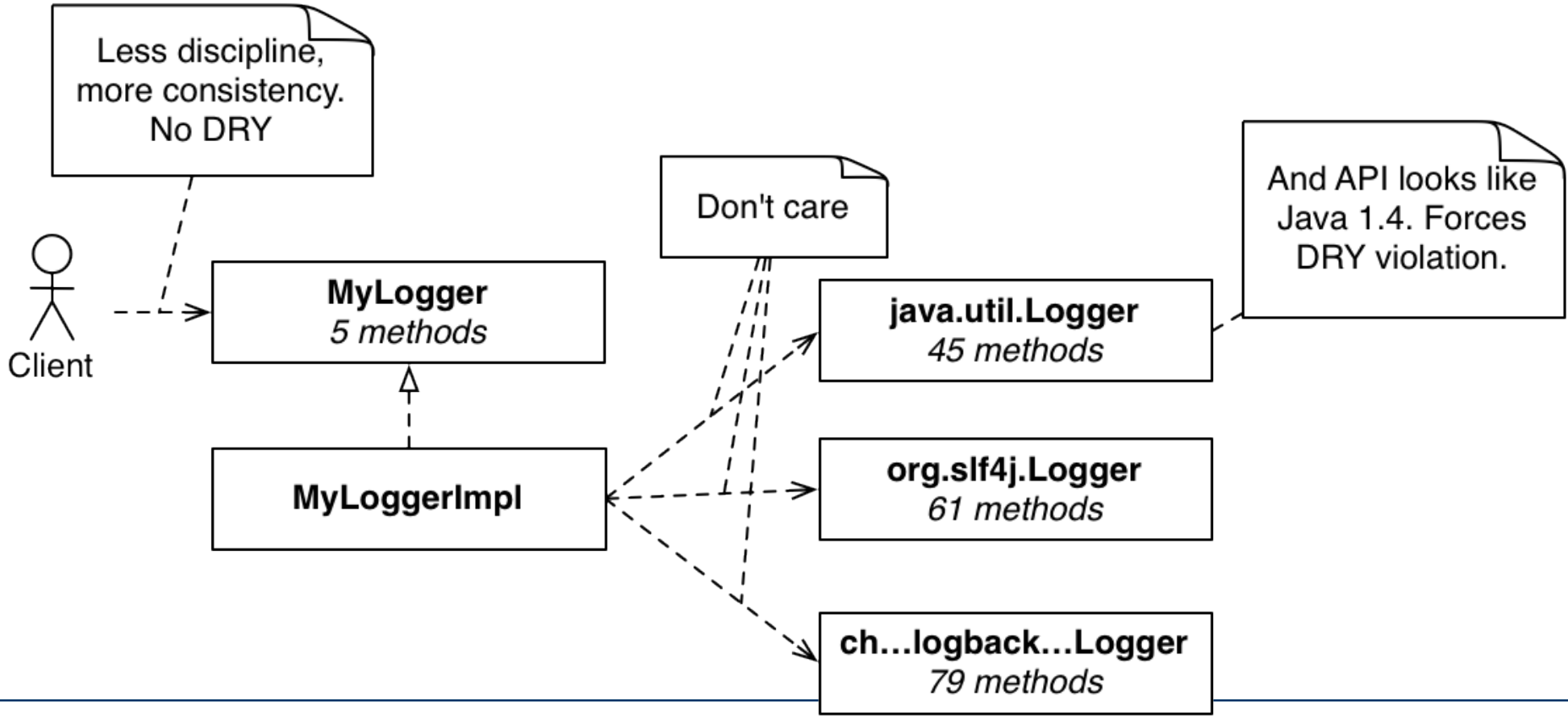
Dependency Inversion Principle (DIP): Negativbeispiel (zu wenig)

- Größere Projekte tun sich schwer, Logging konsistent zu benutzen, wenn sie direkt eine Standard-API verwenden.
  - Außerdem kommen durch verwendete Open-Source-Produkte gern gleich mehrere solche APIs vor
    - http://martinfowler.com/articles/dipInTheWild.html



Dependency Inversion Principle (DIP): Positivbeispiel (gelungen)

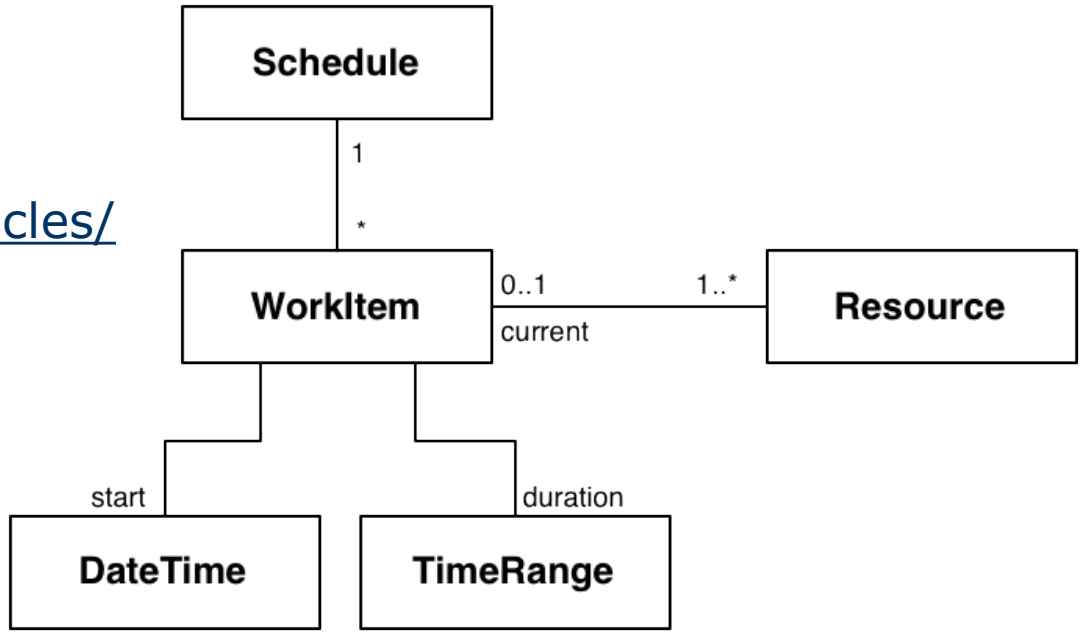
- Es ist deshalb sinnvoll, sich lieber eine eigene, schlanke API zu definieren – und zwar abstrakt
  - Und flugs sieht man auch die mehreren fremden APIs nicht mehr, selbst falls man sie anspricht



Dependency Inversion Principle (DIP): 2. Beispiel

- Für ein System wie dieses
  - (berechne Zeitpläne für Arbeitsposten, die Ressourcen brauchen und dadurch in Konflikt stehen können)
- muss Zeit eine frei manipulierbare Implementierung haben:
  - Beschleunigen, beliebige Zeitreisen.
- Denn wer hier fest Standardklassen benutzt, bekommt riesige Testprobleme.

http://martinfowler.com/articles/dipInTheWild.html



Liskov Substitution Principle (LSP)

- Was?
  - Untertypen müssen die Verträge ihrer Obertypen einhalten
    - egal ob Klassen oder Interfaces
    - Vererbung ist deshalb nicht "is a", sondern "behaves like"
      - Ein Quadrat ist ein Rechteck, aber ein Quadrat-Objekt ist kein Rechteck-Objekt!
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
- Hinderungsgrund:
  - Bequemlichkeit

Barbara Liskov
Wikimedia Commons

Liskov Substitution Principle (LSP)

- Wirkung von Verletzungen:
  - Nach Einführen eines neuen Untertyps treten plötzlich Versagen in ausgereiftem anderen Code auf.
  - Entwickler verzweifeln beim Versuch, die Benutzung einer größeren Klassenhierarchie zu verstehen
- Hilfreiche Techniken:
  - Design by Contract (DbC): Verträge in Obertypen explizit formulieren
    - Voraussetzungen
      - inkl. Aufrufreihenfolgen
    - Wirkungen
      - Rückgabewerte
      - Zustandsänderungen
      - Nebenwirkungen
    - ggf. nichtfunktionale Eigenschaften
    - Möglichst viel davon zur Laufzeit überprüfen

Liskov Substitution Principle (LSP): Positivbeispiel (gelungen)

- java.util.Iterator:
  - remove() sagt ausdrücklich, dass nicht jede Implementierung das erlaubt.

Principle of Least Astonishment (LA)

- Was?
  - Klassen, Methoden, Variablen sollten das tun und enthalten, was ihr Name suggeriert
    - nicht etwas anderes
    - nicht noch mehr
  - und außerdem die Konventionen der Sprache und Plattform einhalten
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
- Hinderungsgrund:
  - Führt zu langen Namen oder
  - verlangt schmale Klassen u. semantisch kleine Methoden

Principle of Least Astonishment (LA)

- Wirkung von Verletzungen:
  - Mehr Programmierfehler
  - Schnelleres Verfallen der Entwurfsstruktur
    - weil die Änderer sich weniger zum Ordnung-Halten veranlasst sehen
- Hilfreiche Techniken:
  - Funktionen *oder* Prozeduren
    - (nicht: Resultate und Nebenwirkungen mischen)
  - Namenskonventionen haben
  - Namenskonventionen streng einhalten
  - Domänenbegriffe benutzen
  - SRP
    - inbes. Software-Blutgruppen trennen
  - Design by Contract: Verträge angeben
    - und zwar sorgfältig

Principle of Least Astonishment (LA): Negativbeispiel

Freie Universität Berlin

- Wenn eine Methode so deklariert ist:
  `FileListDiff computeDiff(…)`
- Was erwarten Sie dann für eine Art von Funktionalität?
- Welche nicht?

- So sieht die Methode tatsächlich aus:
  - IncomingProjectNegotiation.java (ab Z511, Z542)

Information Hiding (IH, "Geheimnisprinzip")

Freie Universität Berlin

- Was?
  - Jede Schnittstelle sollte eine Entwurfsentscheidung verbergen
    - D. Parnas: Geheimnisprinzip
    - Ein Aspekt von SoC
  - Deren Thema sollte benannt sein
- Wofür?
  - Evolvierbarkeit
  - Produktionseffizienz
- Hinderungsgrund:
  - Faulheit
  - Mangel an Reflektion
  - ggf. der Fall: Verbergen ist aufwändig, aber Änderung unwahrscheinlich

Information Hiding (IH)

- Arten und Wirkung von Verletzungen:
  - Durch Modulbenutzer: Wissen, das nicht explizit Teil der Schnittstelle ist, ist dennoch im System verteilt
    - Abhilfe: Dependency Injection
  - Durch Modulprogrammierer: Änderungen betreffen häufig viele verstreute Codestellen anstatt nur ein Modul
- Hilfreiche Techniken:
  - Module durch die Identifikation von Geheimnissen bilden
    - anstatt durch die Identifikation von Zuständigkeiten
      - Oft fällt beides ohnehin zusammen
  - Dokumentation angewöhnen: "Diese Klasse/Methode/… verbirgt […]" / "…knows how to…"

Information Hiding (IH): Negativbeispiel (zu viel)

Freie Universität Berlin

- Dieses Python-Programm hier:
  - print(chr(7))
- könnte man in folgende Module zerlegen (mit Blutgruppe):
  - AsciiControlCodes (T)
  - CharacterCreation (R)
  - ChannelOutput (A)
  - ChannelDirectory (T)
  - ChannelCapabilityChecker (T)
- Das genügt stark dem Geheimnisprinzip, ist aber unsinnig.

Information Hiding (IH): Positivbeispiel (gelungen)

- Python-Standardmodul email
  - verbirgt unzählige Einzelheiten von acht verschiedenen MIME- und Email-Standards
- Java-Standardmodul JapaneseChronology
  - in java.time.chrono
  - verbirgt die Regeln des kaiserlichen japanischen Kalendersystems
    - jedenfalls die jüngeren

Information Hiding (IH): Negativbeispiel (zu wenig)

Freie Universität Berlin

- Wenn die Portierung einer SW auf eine andere Plattform schwierig ist, wurde meist Wissen zu weit verstreut.

- Ein Paradebeispiel: Portieren von 32 nach 64 bit
  - Hauptproblem ist oft verstreutes Wissen (das jetzt ungültig wird) über die Größe der diversen int-Typen
  - http://stackoverflow.com/questions/3557877/porting-linux-32-bit-app-to-64-bit

Übersicht

Prinzipien:

- Interface Segregation Prin. (ISP)
- Dependency Inversion Prin. (DIP)
- Liskov Substitution Prin. (LSP)
- Principle of Least Astonishment (LA)
- Information Hiding Prin. (IH)

Praktiken:

- Automatisierte Unit Tests
- Testattrappen (Mock-ups)
- Code Coverage Analyse
- Teilnahme an Fachveranstaltungen
- Komplexe Refaktorisierungen

Automatisierte Unit-Tests

- Was?
  - Automatisiere Tests, die jedes Modul separat gründlich prüfen
    - Tests sollen schnell ablaufen
    - Tests sollen keine evtl. unverfügbaren Dienste benötigen
  - → Debugging geht sehr schnell und Zutrauen zum Code ist hoch
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
- Hinderungsgrund:
  - Gründliche Tests machen viel Arbeit
  - Isolation von anderen Modulen macht ggf. zusätzlich Arbeit
    - und verlangt konsequente Einhaltung von DIP

Automatisierte Unit-Tests

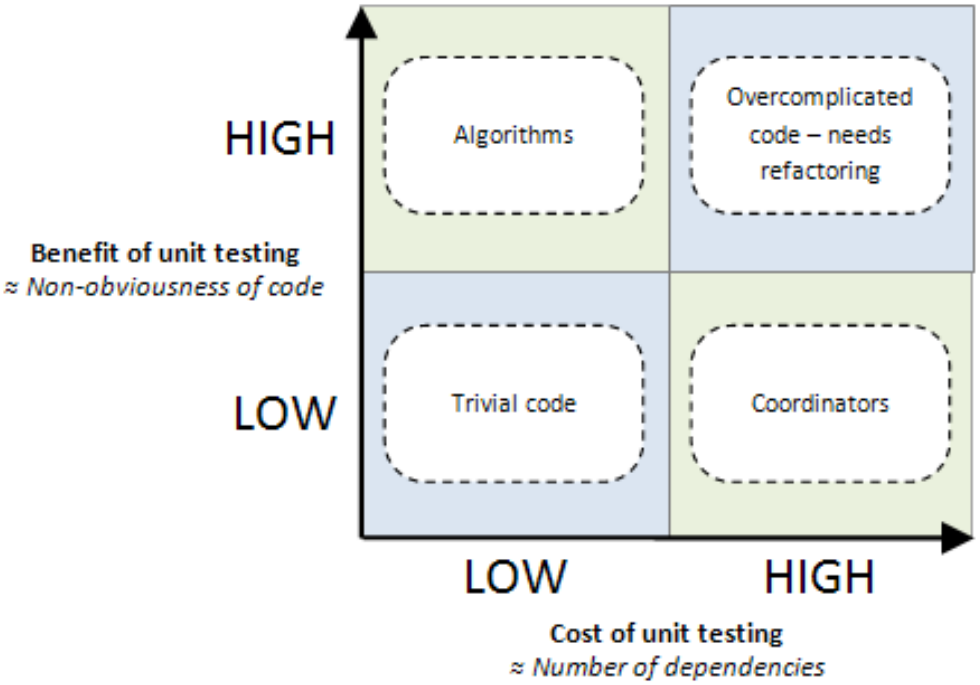
- Wirkungen von Verletzungen:
  - Angst vor Codeänderungen
    - deshalb insbes. zu wenig Aufräumen per Refactoring
  - Bei Versagen ist das Debugging oft aufwändig

- Hilfreiche Techniken:
  - Test-First-Entwicklung
    - insbes. Test-Driven-Design (TDD)
  - Refactoring
    - mit Ziel Testbarkeit
    - [Legacy]
  - Einsatz von Testframeworks

Automatisierte Unit-Tests: Negativbeispiel (zu viel)

- Unit-Testing übertreiben heißt:
  - Es gibt viele redundante Tests
  - Niemand hat einen guten Überblick
  - Es entsteht Scheu vor semantischem Umbau
    - weil man dafür zu viele Tests mit ändern müsste
  - Zweite Art: Triviale Tests
    - z.B. getter/setter in Java

- Andere Art:
  - Man testet mühsamst Code, der schwierig zu testen ist, anstatt ihn zu vereinfachen
  - Überlegungen hierzu: Sanderson: "Selective Unit Testing – Costs and Benefits"



Automatisierte Unit-Tests: Positivbeispiel (gelungen)

Freie Universität Berlin

Python-Standardbibliothek calendar-Modul:

- Lib/calendar.py (700 LOC)
- Lib/test/test_calendar.py
  - 700 LOC
  - Hartkodiertes Testresultat *result_2004_days* für *test_yeardayscalendar*
    - Einmal manuell prüfen, dann einfrieren.
  - Heuristische Prüfung für test_days
    - 7 Stück? Keine doppelt?
    - Namen kommen vom Betriebssystem
    - diverse Python-Logik, die das kaputt machen könnte

Python-Standardbibliothek datetime-Modul:

- Lib/datetime.py (2100 LOC)
- Lib/test/test_datetime.py
  - Rahmen wg. 2 Varianten
    - technisch kompliziert
    - dann alle Testfälle gleich
- …/datetimetester.py
  - 3800 LOC
  - *test_computations* prüft Zeitintervallarithmetik
    - 48 asserts
      - tapfer bleiben!
    - Und nicht auf Leute hören, die verlangen: "Nur 1 assert pro Test!"

Automatisierte Unit-Tests: Negativbeispiel (zu wenig)

Freie Universität Berlin

- Java Standardbibliothek java.util.GregorianCalendar implements Calendar
  - GregorianCalendar.java (3200 LOC)
  - Calendar.java (2800 LOC)
    - (beide inkl. viel Doku)
    - 480 LOC netto allein für add(field, amount)

- Tests zu beidem sind nur:
  - WeekDateTest.java (190 LOC)
    - 21+3+7 tabellengesteuerte Testfälle, 303 Plausib.-Prfg., zu 4 Themen, nur über Wochenlogik
  - Bug6645263.java (47 LOC)
    - 1 Testfall (Exception)
  - Übrigens alles ohne Testframework

- Einziger Trost:
  - Eine Standardbibliothek wird selten geändert

Testattrappen (test doubles): Isolation

- Was?
  - Benutze im Unittest von U eine Attrappe K' anstelle einer abhängigen Klasse K, wenn
    1. K schwierig zu konfigurieren oder nicht immer verfügbar ist,
    2. K zu langsam abliefe,
    3. Du Aufrufe prüfen möchtest anstatt deren Resultate
       - *Verhaltensprüfung* (behavior verification) statt *Ergebnisprüfung* (state-based verif.),
    4. K noch nicht existiert,
    5. oder K selbst Defekte haben könnte

- Wofür?
  - Produktionseffizienz

- Hinderungsgrund:
  - umständlich
  - Verhaltensprüfung macht den Test von U abhängig von U's Implementierung
    - der Test teilt also Geheimnisse von U: **White-Box-Testen,**
      - das muss einem unbedingt bewusst sein

Testattrappen (test doubles): Arten von Attrappen

- Dummy
  - Funktionsloses Objekt, das nur durchgereicht, aber gar nicht aufgerufen wird
- Stub
  - Fixes Verhalten
  - meist hartkodiert
- Fake
  - Kann nicht alle Fälle behandeln
  - stark vereinfachte "echte" Lösung
- Spy
  - Ein Testobjekt, das Abläufe geeignet protokolliert
    - Achtung: Das führt zu White-Box-Testen
- Mock
  - Ein Spy oder Stub+Spy, der die Korrektheitsprüfung gleich mit eingebaut hat
  - Im Jargon heißen Stubs, Fakes, Spies oft alle Mock

- http://martinfowler.com/bliki/TestDouble.html
  - Relevanz f. Prozess: [MockStub]
- http://blog.8thlight.com/uncle-bob/2014/05/14/TheLittleMocker.html

Testattrappen (test doubles): Arten von Attrappen: Namen??

Freie Universität Berlin

- Wie üblich sind diese Namen aber nicht einheitlich:
  - Meszaros: "Mocks, Fakes, Stubs, and Dummies"
  - Website zum Buch "xUnit Test Patterns: Refactoring Test Code" von Gerard Meszaros [xUTP]

Testattrappen (test doubles): Isolation

Freie Universität Berlin

- Wirkungen von Verletzungen:
  - Debugging nach Versagen ist aufwändig
    - wg. Defekten in anderem Code
    - wg. Crash in anderem Code, den der aber gar nicht verschuldet hat
  - Tests laufen langsam
  - Tests sind nicht robust
    - z.B. wg. Abhängigkeiten von externen Servern
  - Manche Dinge können nicht oder kaum getestet werden
    - z.B. Zeitabhängiges!
    - z.B. Ausnahmebehandlung
- Hilfreiche Techniken:
  - Für Stubs:
    - Duck typing (in dyn. Sprachen)
    - Monkeypatching (dito) (siehe z.B. DHH blog)
  - Für Spies/Mocks: Mocking-Frameworks
    - Java: JMock, EasyMock, Mockito und andere
    - Python: unittest.mock und andere

Testattrappen/Isolation: Negativbeispiel (zu viel)

Freie Universität Berlin

- Wenn ein Mockobjekt K' ein anderes Mockobjekt L' benutzt, werden oft sogar Geheimnisse fremder Klassen L aufgebrochen!
  - Dann ist der Entwurf oder der Testentwurf ungünstig
  - Wahrscheinlich sollte L' ein Stub sein und in K' eingebaut: komplexes Datenobjekt wird benötigt

- Siehe auch: Blogartikel "Why I don't mock"
  - und Diskussion dazu (gut z.B. emsy, mpdehaan, InclinedPlane, Pling(!))
  - Aber Achtung: Immer aufpassen, ob über echtes Mocking gesprochen wird (insbesondere Verhaltensprüfung) oder über Testattrappen generell.

Testattrappen/Isolation: Positivbeispiel (gelungen)

Freie Universität Berlin

Python-Standardbibliothek:

- Lib/test/test_asynchat.py (asynchrones Socket-I/O)
  - Testziel: "handle_read must ignore BlockingIOError"
  - Attrappe *sock* induziert Fehler (Zeile 282)
  - Attrappe *handle_error* (Zeile 288) stellt seine Behandlung fest
    - (nämlich: darf nicht aufgerufen werden, Zeile 290)

```
278 class TestAsynchatMocked(unittest.TestCase):
279     def test_blockingioerror(self):
280         # Issue #16133: handle_read() must ignore BlockingIOError
281         sock = unittest.mock.Mock()
282         sock.recv.side_effect = BlockingIOError(errno.EAGAIN)
283 
284         dispatcher = asynchat.async_chat()
285         dispatcher.set_socket(sock)
286         self.addCleanup(dispatcher.del_channel)
287 
288         with unittest.mock.patch.object(dispatcher, 'handle_error') as error:
289             dispatcher.handle_read()
290         self.assertFalse(error.called)
```

Code-Coverage-Analyse

- Was?
  - Messe, wie viel Anteil des Codes durch Tests exerziert wird
    - Statements (C_0)
    - Verzweigungsziele (C_1)
- Wofür?
  - Korrektheit
  - Produktionseffizienz
- Hinderungsgrund:
  - Messung ist evtl. kompliziert bei Code, der verteilt oder in Containern abläuft

Code-Coverage-Analyse

- Wirkungen von Verletzungen:
  - Viele redundante Tests

  und/oder
  - Viele Lücken in der Testabdeckung

- Nicht übertreiben!
  - 90% C_0 -Abdeckung sollte man meist hinbekommen können
    - und sind auch sinnvoll
  - aber 100% sind manchmal unmöglich (bei C_1 sogar oft) und oft übertrieben teuer

- Hilfreiche Werkzeuge:
  - Java: Cobertura, Emma u.a.
  - Python: coverage.py
    - auch als Plugin für nose und pytest verfügbar

Code-Coverage-Analyse: Beispiel: Cobertura-Bericht

Freie Universität Berlin

- http://mojo.codehaus.org/cobertura-maven-plugin/sample-maven-shared-io-report/index.html
  - siehe: Übersicht,
  - org.apache.maven.shared.io.logging,
  - DefaultMessageHolder.

Code-Coverage-Analyse: Negativbeispiel (zu viel)

Freie Universität Berlin

- Coverage-Analyse darf sich nicht verselbständigen
  - weder lohnt meistens der Aufwand für die letzten paar Prozent Abdeckung
  - noch besagt eine hohe Abdeckung, dass die Tests auch **gut** sind
    - nur Assertions ergeben Tests!

!

Teilnahme an Fachveranstaltungen

- Was?
  - Austausch mit immer wieder anderen Entwicklern vermeidet, nur im eigenen Saft zu schmoren
    - Firma: zweiwöchentlich
    - Regional: monatlich
    - Überregional: jährlich
- Wofür?
  - Produktionseffizienz
    - ein Tipp fällt immer ab
  - Reflexion
- Hinderungsgrund:
  - Scham, Schüchternheit
  - Übergroße Introversion
  - Kostet Zeit und überregional auch Geld

Teilnahme an Fachveranstaltungen

- Wirkungen von Verletzungen:
  - Lesen ist gut, aber manches wird erst durch Diskussion klar.
  - Und auch zum Lesen braucht man erst mal Anregungen und Motivation
- Hilfreiche Anlaufpunkte:
  - meetup.com
  - c-base
  - Berlin Web Week
  - GOTO conference
  - und diverse andere

Komplexe Refaktorisierungen

- Was?
  - Umstrukturierungen wie
    - Parameter zufügen/entfernen
    - Konstanten/Variablen extrahieren
    - Klassen verschmelzen/auftrennen
    - Verschieben zwischen Ober-/Unterklassen
    - und viele mehr
- Wofür?
  - Evolvierbarkeit
- Hinderungsgrund:
  - Unaufmerksamkeit für den Verfall der Strukturen
  - Faulheit
    - gegen beides hilft die Pfadfinderregel
  - Braucht gute Werkzeugunterstützung
    - Hier sind dynamische Sprachen schwächer

Komplexe Refaktorisierungen

- Wirkung von Verletzungen:
  - Trotz emsiger Benutzung von Methodenextraktion und Umbenennen wird der Entwurf allmählich immer schlechter
  - Architekturänderungen (selbst kleinere) sind schmerzhaft
    - selbst wenn man automatisierte Tests hat
    - weil man auch die mit umstrukturieren muss
- Hilfreiche Techniken:
  - Martin Fowlers Katalog von Refactorings
  - Werkzeuge wie Eclipse, die JetBrains-IDEs, MS Visual Studio

Übersicht

Prinzipien:

- Interface Segregation Prin. (ISP)
- Dependency Inversion Prin. (DIP)
- Liskov Substitution Prin. (LSP)
- Principle of Least Astonishment (LA)
- Information Hiding Prin. (IH)

Praktiken:

- Automatisierte Unit Tests
- Testattrappen (Mock-ups)
- Code Coverage Analyse
- Teilnahme an Fachveranstaltungen
- Komplexe Refaktorisierungen

Danke!