

## Stufe 4: Grüner Grad von CCD

Lutz Prechelt

Institut für Informatik, Freie Universität Berlin

## Prinzipien:

- Open Closed Principle (OCP)
- Tell, don't ask (TDA)
- Law of Demeter (LoD)

## Praktiken:

- Continuous Integration (CI)
- Statische Codeanalyse, Metriken
- Inversion of Control Container (ICC)
- Erfahrung weitergeben
- Messen von Fehlern

- Was?
  - Module sollten offen sein für Veränderungen ihres Verhaltens
    - vor allem für Erweiterungen
  - aber trotzdem geschlossen für Modifikationen ihres Codes
    - das geht mittels Vererbung oder Delegation/Injektion (Strategiemuster)
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
- Hinderungsgrund:
  - Unklar, wann man mit dem "geschlossen"-Zustand anfangen sollte
    - Modul sollte vorher "fertig" sein
      - doch was heißt das?
  - Unklar, wann man später die Regel überhaupt anwenden sollte
    - denn wenn etwas "wirklich" in ein Modul gehört, dann sollte man es auch reintun

# Open Closed Principle (OCP)

- Wirkungen von Verletzungen:
  - Für sehr zentrale Module mit vielen Benutzern:
    - Hoher Änderungsaufwand bei Modifikationen,
    - hohes Risiko bei Modifikationen
  - Für randständige Module:
    - Aufwand und Risiko können durch Nichtbeachten von OCP sogar sinken
- <http://c2.com/cgi/wiki?OpenClosedPrinciple>
- Hilfreiche Techniken:
  - Hellseherei
  - Unterscheide klar Bibliothekscode B von Projektcode P
    - B: OCP sehr wertvoll
    - P: OCP-Verletzungen eher unproblematisch
  - Verträge so zuschneiden, dass sie möglichst stabil werden
  - Single Responsibility Principle (SRP)

## Zu wenig OCP:

- Klassen mit analogen switch/case-Anweisungen in mehreren Methoden
  - falls ein neuer Fall hinzu kommen könnte
  - [Beispiel bei C-C-D](#)

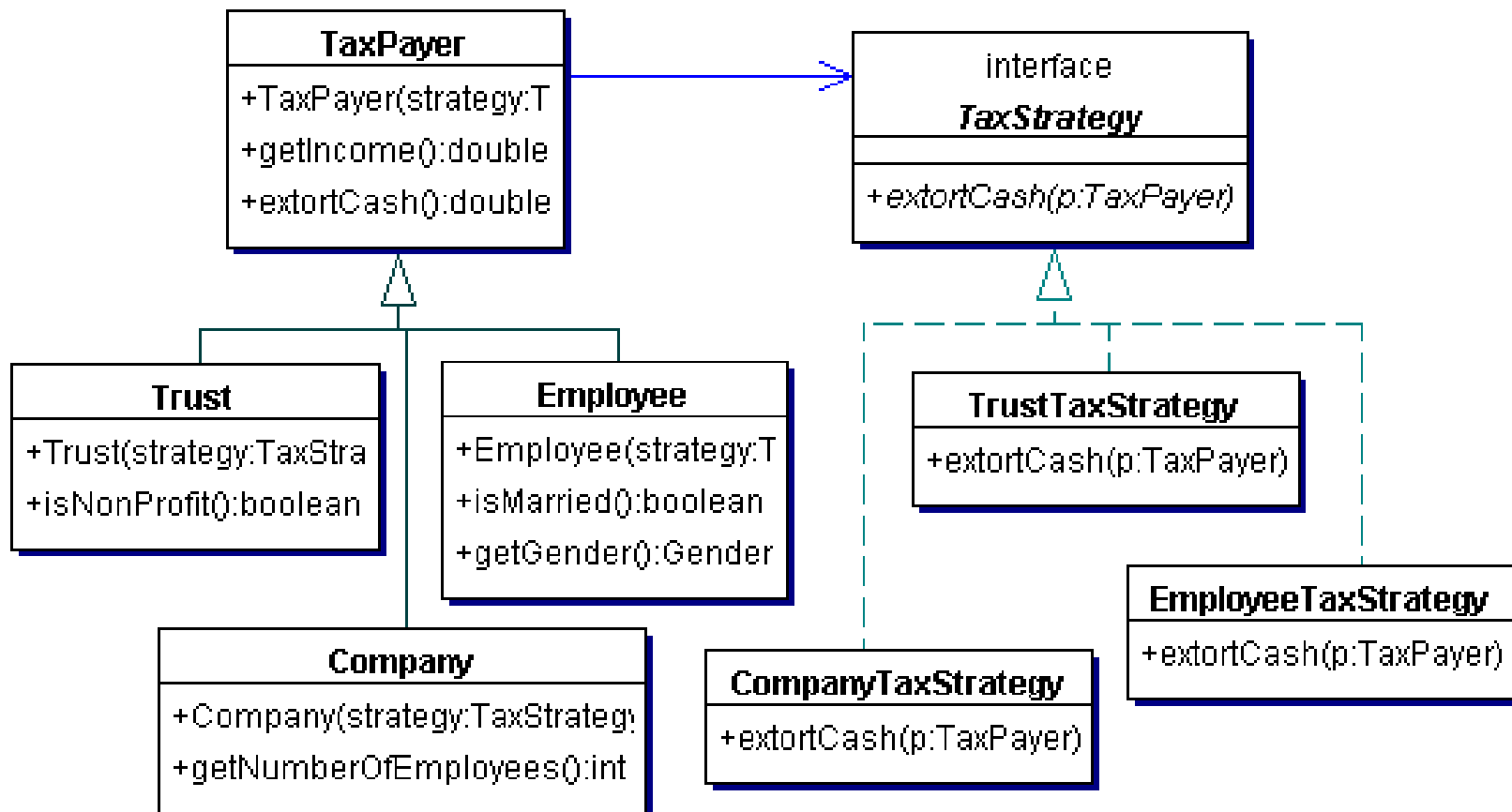
## Zu viel OCP:

- Einführung von Abstraktionen, die noch gar nicht benötigt werden
  - verletzt KISS und YAGNI
  - siehe Diskussion auf <http://c2.com/cgi/wiki?OpenClosedPrincipleAndXp>

# Open Closed Principle (OCP): Positivbeispiel (gelingen, jedoch naiv)

- TaxPayer ist gut geschlossen
  - Erweiterung durch Vererbung
- Auch die Unterklassen sind gut geschlossen:
  - neue TaxStrategies kann man unabhängig ergänzen

Spielzeugbeispiel



# Tell, don't ask (TDA)

- Was?
  - Klassen sollten vermeiden, viel Zustand offen zu legen
  - Sondern Methoden haben, die das Ziel des Klienten direkt unterstützen:
    - Alec Sharp: "*Procedural code gets information then makes decisions. Object-oriented code tells objects to do things.*"
- Wofür?
  - Evolvierbarkeit
- Hinderungsgrund:
  - Eventuell sind nicht alle Ziele von Klienten vorhersehbar
    - und wenn man dann OCP einhalten will und die Klasse nicht ändert...

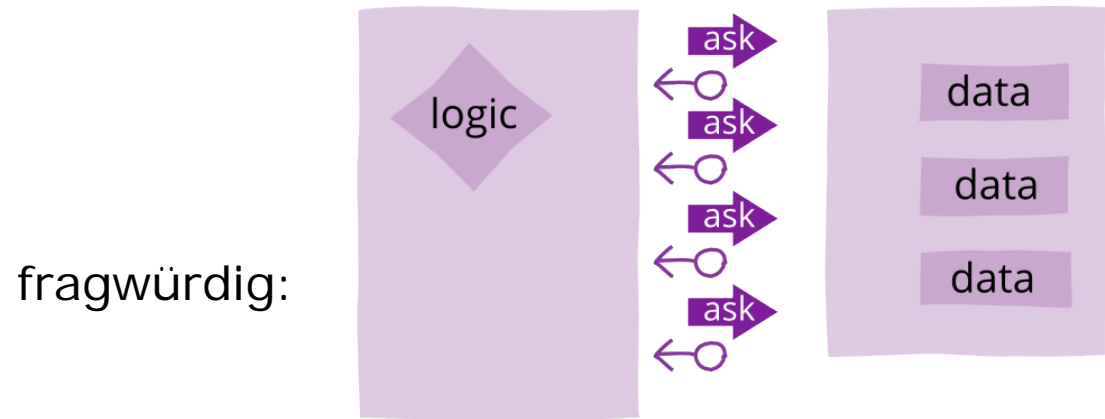
- Wirkung von Verletzungen:
  - Zuständigkeiten sind verschmiert
    - Klienten treffen Entscheidungen, die der Klasse zustehen sollten
    - Redundanz (über Klienten hinweg)
  - Tendenz zu reinen Daten"klassen"
- Hilfreiche Techniken:
  - Sich beim Offenlegen von Zustand (`public T getX()`) stets fragen: "WARUM brauche ich das?"
  - Auf [FeatureEnvy](#) achten
- Vorsicht vor "zu viel":
  - Nie ureigene Klientenzuständigkeiten in die Klasse verlagern!



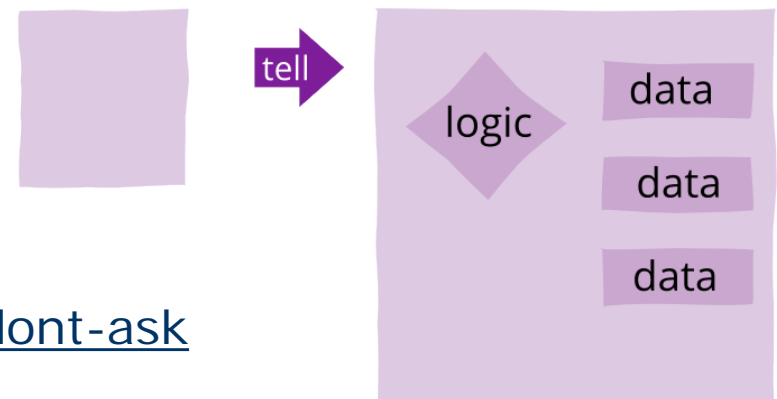
# Tell, don't ask: Negativbeispiel (zu wenig), Positivbsp.

- <http://martinfowler.com/bliki/TellDontAsk.html>

- AskMonitor
- TellMonitor



besser:



- Weitere Bsp:

- <http://robots.thoughtbot.com/tell-dont-ask>

# Law of Demeter (LoD)

- Was?
  - "Don't talk to strangers"
  - Um Kopplung zu verringern, rufe nur Deine Nachbarn, nicht deren Nachbarn
    - Nachbarn sind: assoziierte Klassen, Methodenparameter, selbst erzeugte Objekte
    - Ausnahmen gelten für Datenklassen
  - War eine Entwurfsvorschrift im Demeter-Projekt 1987
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
- Hinderungsgrund:
  - Verlangt häufig die Einführung zusätzlicher Methoden in Nachbarklassen
    - und wenn ich die nicht ändern kann/darf/soll/will...
  - Kann Einhaltung des SRP schwächen
    - Die von LoD verlangten Zusatzmethoden in Nachbarklassen passen dort evtl. schlecht hin

Ähneln im Effekt oft TDA.

# Law of Demeter (LoD): Negativbeispiele (zu viel)

1. Klassen, die den Kern einer Anwendung bilden, braucht man nicht zu entkoppeln
  - sie haben "Ewigkeitswert"
  - z.B. Tabelle, Zeile, Spalte, Zelle in Excel
2. Subsysteme können *lokale* solche Mengen sehr stabiler Klassen besitzen
  - LoD-Verletzungen innerhalb des Pakets sind dann unkritisch

3. Views (für Webseiten) sind dazu da, Daten zu präsentieren
  - sie sind faktisch ohnehin an alles gekoppelt, das sie präsentieren
  - LoD ist hier unnötig
- Diskussion:
  - [Phil Haack](#)
  - [c2.com](#), und
  - nett zusammenfassend: [Martin Fowler](#)



## Prinzipien:

- Open Closed Principle (OCP)
- Tell, don't ask (TDA)
- Law of Demeter (LoD)

## Praktiken:

- Continuous Integration (CI)
- Statische Codeanalyse, Metriken
- Inversion of Control Container (ICC)
- Erfahrung weitergeben
- Messen von Fehlern

# Continuous Integration (CI)

- Was?
  - Die Änderungen aller Entwickler an der Codebasis werden häufig zusammengeführt und das Resultat getestet
    - allermindestens täglich
    - Tests sind automatisch
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
- Hinderungsgrund:
  - Verlangt, dass größere Änderungen in kleine Schnipsel unterteilt werden
    - und das verlangt Übersicht und Planung
  - Ergibt nur mit einem CI-Server richtig Sinn
    - und den gut aufzusetzen ist keine Kleinigkeit.
    - Regeln bezügl. Nutzung von Versionszweigen nötig

- Wirkungen von Verletzungen:
  - Zusammenführen wird mühsamer, oft frustrierend
    - Dadurch wird es noch seltener
    - Dadurch wird es noch mühsamer
  - Dadurch werden viele Änderungen gar nicht erst mehr gemacht
  - Insbesondere kein größeres Refactoring
  - Dadurch verfällt die Codebasis zügig
- **Wichtig!**
- Hilfreiche Techniken:
  - dezentrale Versionsverwaltung
    - [git](#), [Mercurial](#), [Bazaar](#), u.ä.
  - Versionsarchiv-Portale
    - [github](#), [bitbucket](#), u.ä.
  - [Continuous Integration Server](#)
    - Jenkins, Travis o.ä.



# Jenkins

# Continuous Integration (CI): Beispiel: Saros

- Bei Saros sollen Änderungen in "kleine Patches" zerlegt sein
  - Klein heißt: [in unter 15 Minuten zu begutachten](#)
- Nicht alle Entwickler halten sich dran
  - weil das Übersicht und Disziplin erfordert.
  - In [diesem Fall hier](#) war es selbst nach dem Zerkleinern noch zu groß.
    - Über 2000 Zeilen!
    - Der Code kam von einem Nicht-Saros-Entwickler.
  - Wurde dann weiter zerlegt: Patches [1970](#), [1971](#), [1972](#)
- Ein Jenkins-Server führt dann aus:
  - compilation
  - packaging
  - static analysis
  - unit tests
  - system tests
- <http://saros-con.imp.fu-berlin.de/jenkins/>
  - Siehe Saros: status, coverage report, test result

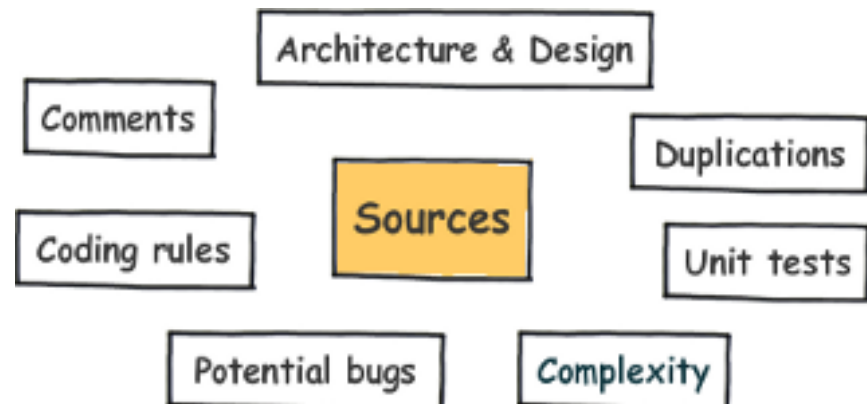


- Was?
  - Verwende automatische Messungen des Codes, um (a) problematische Trends oder (b) dubiose Stellen zu entdecken
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
  - Reflexion
- Hinderungsgrund:
  - (a), (b): Auswahl und Aufsetzen der Werkzeuge ist nicht einfach
  - (b): Werkzeuge erzeugen viele falsche Alarme
    - ohne genaue Konfiguration werden die Resultate deshalb sehr bald ignoriert
    - diese Konfiguration (und deren Pflege) macht Arbeit
  - (a), (b): Ergebnis kann beschämend sein



- Arten von Verletzungen:
  - Nicht tun
  - Ergebnisse ignorieren
  - Werkzeuge fast überall stumm schalten
- Wirkung von Verletzungen:
  - (a) allmählicher Verfall der Codestruktur kann nicht nachgewiesen werden
  - (b) viel vermeidbarer Aufwand für Debugging

- Hilfreiche Techniken:
  - [SonarQube](#) u.a.
  - (b) beim ersten Einsatz genug Zeit nehmen, allen Meldungen nachzugehen
    - pro Meldungstyp: entweder komplett abschalten
      - oder auf halbjährlichen Turnus ändern
    - oder Code (fast) komplett säubern



## 1. SonarQube bei Saros:

- <http://saros-build.imp.fu-berlin.de/sonarqube/>

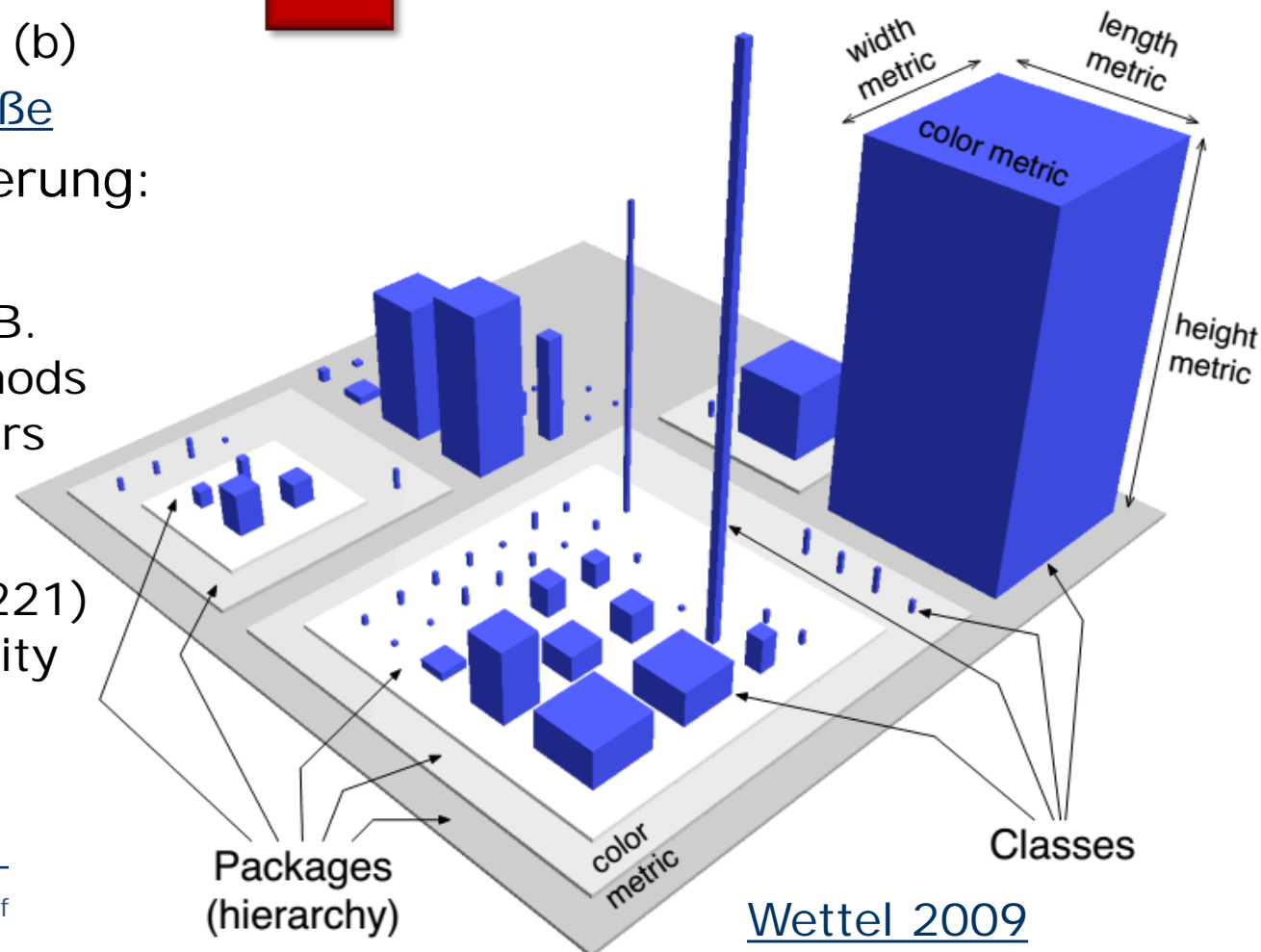


- bislang fast nur (b)
- [SonarQubes Maße](#)

## 2. Entwurfsvisualisierung:

- Stadt-Metapher
  - bei [[EvoVis](#)]: z.B.  
height = #methods  
basesize = #attrs

color ~ age (p.221)  
location = identity  
(p.222)



- Was?
  - Bei DI: Injiziere benötigte Objekte nicht manuell, sondern lasse einen IoC-Behälter sie liefern
    - oder einen Service Locator.
    - Nach Bedarf neue Objekte oder Singletons,
    - per Interface-Name oder Deklarator, nicht per Klassenname
- Wofür?
  - Evolvierbarkeit
  - Produktionseffizienz
- Hinderungsgrund:
  - Verlangt anfangs Know-How-Aufbau
    - (dann ist es aber recht bequem)

- Wirkung von Verletzungen:

- Es stehen im Code (auch mit DI) an vielen Stellen konkrete Klassennamen
  - und manchmal wäre es schön, die los zu sein
  - z.B. zum Ausprobieren eines neuen Ersatzobjekts in *vielen* Tests
- Komplexe Objekte zu erzeugen ist umständlich
  - weil die Konstruktoren viele Parameter haben
  - und das verschachtelt
- Systeminitialisierung und –stoppen sind schwierig
  - viele Reihenfolgezwänge

- Hilfreiche Techniken:

- Java: [Spring](#) (das ist noch *viel* mächtiger), [PicoContainer](#)



- Python: [eine Mini-Lösung reicht oft](#)

# Inversion of Control Container (ICC): Beispiel: PicoContainer in Saros

- SarosContext

- enthält den MutablePicoContainer (Z85)
- jede Klasse braucht also Zugriff hierauf

- SarosCoreContextFactory

- hier wird der Behälter konfiguriert
- Dutzende von fixen Klassen oder Impls-für-Interfaces eingetragen, z.B.
  - XMPPTransmitter als Impl für ITransmitter
  - IsInconsistentObservable
  - AwarenessInformationCollector

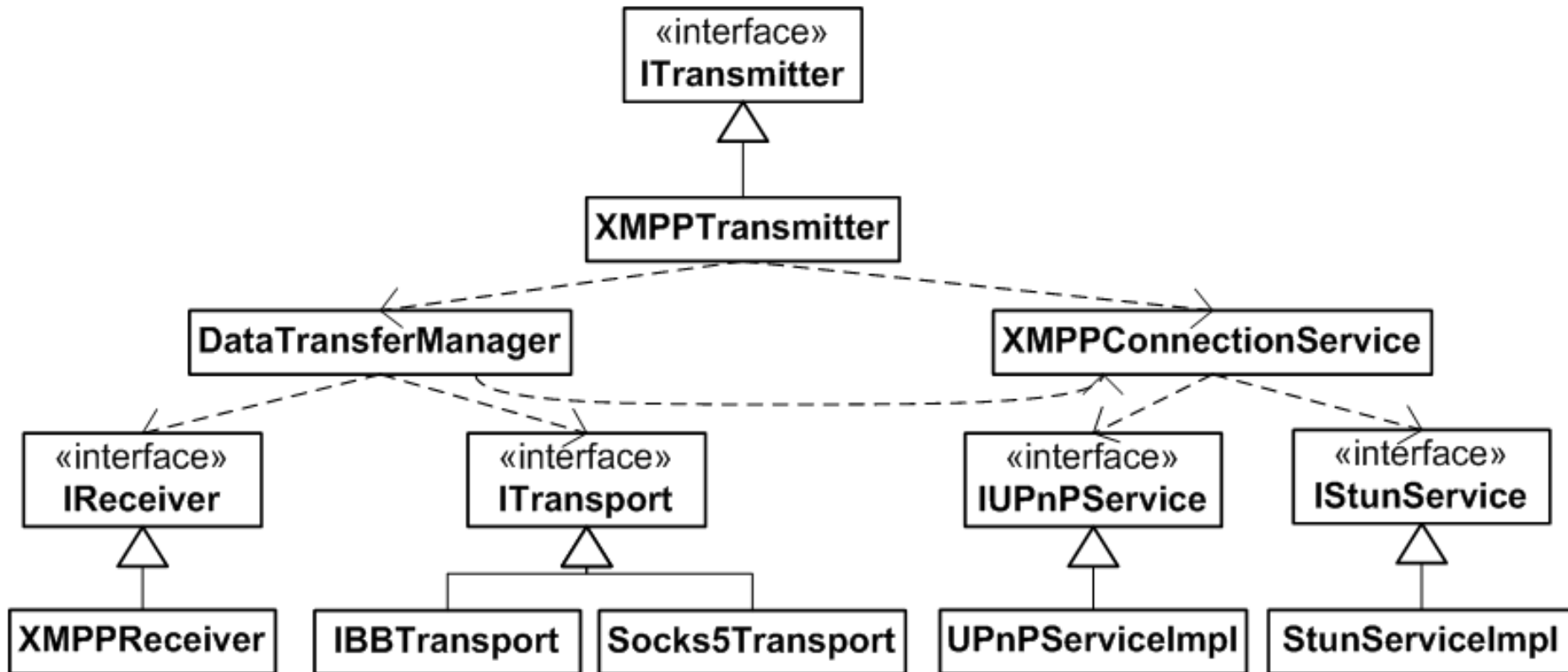
- XMPPTransmitter

- Konstruktor (Z56) braucht: DataManager, XMPPConnectionService
  - DataManager-Konstruktor (170) braucht: XMPPConnectionService, IReceiver, ITransport, ITransport 
  - XMPPConnectionService-Konstruktor (125) braucht: [...] usw.
- u.s.w.
- Tatsächlicher "Aufruf" (in SessionNegotiation, Z63): @Inject ITransmitter transmitter;

# Inversion of Control Container (ICC): Beispiel: PicoContainer in Saros (2)

XMPP-Transmitter:

- Injektion nicht nur zur Bequemlichkeit,
- sondern auch für Austausch gegen FakeTransmitter für Tests

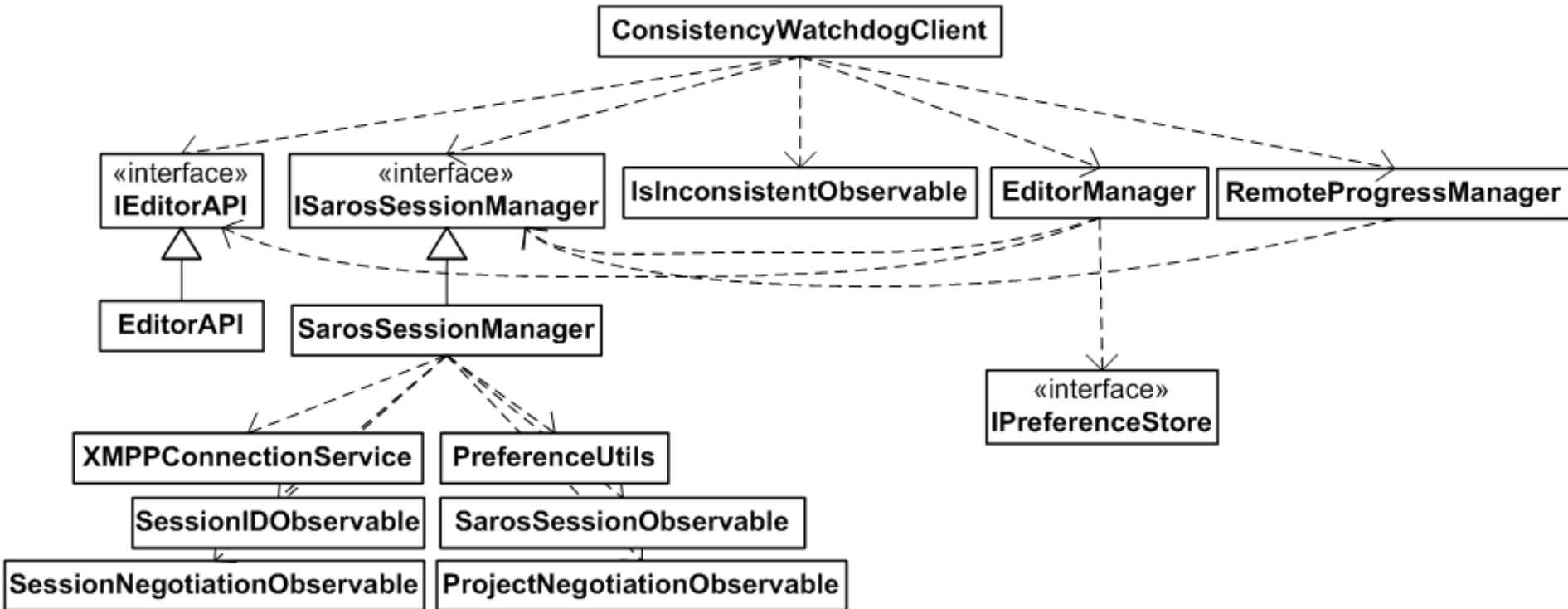


# Inversion of Control Container (ICC): Beispiel: PicoContainer in Saros (3)

Die Zahl von indirekten Abhängigkeiten kann recht groß sein



- z.B. [ConsistencyWatchdogClient](#)
  - Bild geht nur bis Stufe 2, es gibt aber noch mehr!
  - [Erläuterung](#)



- Was?
  - Eigenes Wissen und Verständnis auch anderen vermitteln
- Wofür?
  - Reflexion
- Hinderungsgrund:
  - Verlangt sorgfältig konsolidiertes Wissen
    - nicht nur Halbwissen
  - und klare Begrifflichkeiten, um es zu transportieren
    - Seltsamerweise können sich viele SW-Entwickler nur dann klar ausdrücken, wenn es Programmcode ist.



- Wirkung von Verletzungen:
  - Viel Halbwissen
  - Langsameres Lernen
    - Rabbi Hanina:  
"I have learnt  
much from my teachers,  
more from my colleagues,  
and most from my  
students."

- Hilfreiche Techniken:
  - Vorträge auf Fachveranstaltungen halten
  - Interne Vorträge halten
  - Paarprogrammierung
  - evtl. Blogging

so lange muss  
man nicht warten



- Was?
  - Protokolliere Defektbehebungen, um zu lernen
    - wann Du welche Fehler machst und
    - was sie kosten
- Wofür?
  - Korrektheit
  - Reflexion
- Hinderungsgrund:
  - Verlangt viel Disziplin, weil man eigentlich auf was anderes konzentriert ist

- Wirkung von Verletzungen:
  - George Santayana: *"Those who cannot remember the past are condemned to repeat it."*



- Hilfreiche Techniken:
  - Passende Tastaturmakros
    - global
    - mit automatischer Zeiterfassung
  - passender Defekt-Klassifikations-Standard

## Prinzipien:

- Open Closed Principle (OCP)
- Tell, don't ask (TDA)
- Law of Demeter (LoD)

## Praktiken:

- Continuous Integration (CI)
- Statische Codeanalyse, Metriken
- Inversion of Control Container (ICC)
- Erfahrung weitergeben
- Messen von Fehlern

**Thank you!**