Prozess und Techniken

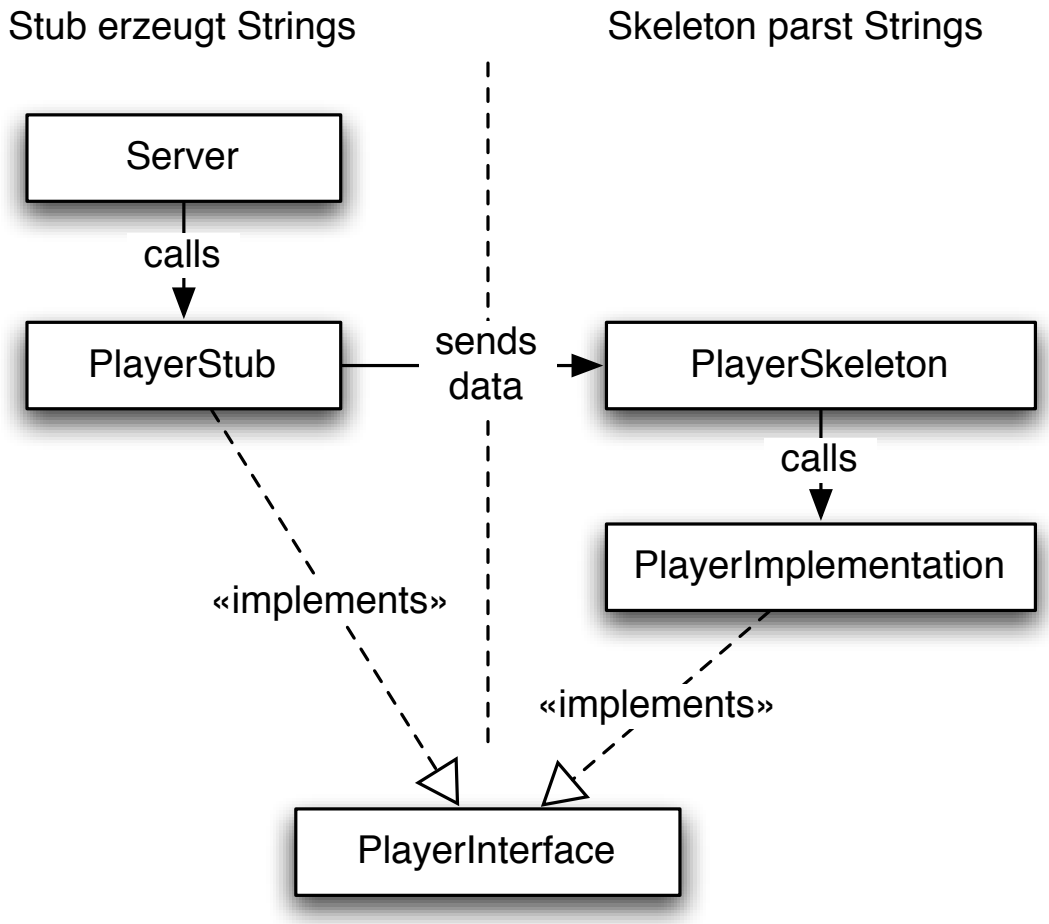
Plenum Programmierpraktikum

2006-11-16

Techniken

Implementierungs-Technik: Stub und Skeleton

- Ziel: Verberge vom Nutzer eine Dienstes, ob der Dienst lokal oder entfernt erbracht wird.
- Definiere hierzu: Interface des Dienstes.
- Der eigentliche Diensterbringer implementiert dieses Interface.
- Er kann aber auf einem entfernten Rechner durch einen $Stub$ vertreten werden. Dieser leitet die Aufrufe übers Netz an den $Skeleton$ auf dem entfernten Rechner weiter, der wiederum den Diensterbringer aufruft.
- Weiterer Vorteil: Der Diensterbringer muss sich nicht mit Protokollen herumschlagen.


Stub erzeugt Strings
Skeleton parst Strings
Server
calls
PlayerStub
sends
data
PlayerSkeleton
calls
PlayerImplementation
«implements»
«implements»
PlayerInterface

Regular Expressions

- Tipp fürs Zusammenbauen von Zeichenketten: Methode `String.format()` und Klasse `java.util.StringBuilder` (= neuere Variante von `StringBuffer`).

- Problem: Auseinandernehmen von Zeichenketten

- Prinzipiell: Parser, spezielle Grammatik

- Einfacher: Reguläre Ausdrücke

Regular Expressions

Anfänglich: einzelne Zeichen. Erster Operator: Konkatenation.

Regex	**Beschreibung**	**Positiv**	**Negativ**
abc	Literal	“abc”, “-abc”, “abc-”	“ab”
123	Literal	“123”, “-123”, “123-”	“12”
...	Beliebiges Zeichen	“abc”, “!?!”, “abc123”	“ab”, “12”
`^ab`	Anfang	“ab123”	“-ab”
`ab$`	Ende	“-ab”	“ab123”
`x+`	Min. einmal	“x”, “xxxxx”	“”, “y”
`x*`	Bel. oft	“”, “xxx”	*nichts*
`(abc)+`	Gruppe	“abc”, “abcabc”	“aa”, “”
`^(ab)|(12)$`	Oder	“ab”, “12”	“ab12”
`^[a-zA-Z]+$`	Zeichenmenge	“StringBuffer”	“Hello-World”
`[^x]+`	Kein “x”	“a”, “abc”	“”, “xxx”

Reguläre Ausdrücke in Java

```
Pattern pattern = Pattern.compile("compute ([0-9]+)");
String input = "compute 123";
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) { // implizit mit ^ und $ geklammert!
    System.out.println("Argument: "+matcher.group(1));
}
```

Selbst ausprobieren

Klasse `de.lmu.ifi.regex.RegexTester`

```
--> !^--([^-]*)--$
--> ?--abc--
MATCH: --abc--
Group 1: abc
--> ?----
MATCH: ----
Group 1:
--> ?--abc--+
NO MATCH
```

Unit Tests: Techniken

- Wichtig: Kleine Inkremente planen (hier helfen Spikes). Damit möglich: Test-first Entwicklung.

- Der Code muss auf Tests vorbereitet werden (weniger Abhängigkeiten, Back-doors).

- Debuggen mit Unit-Tests: Schliesse Fehlerquellen aus, isoliere Problem in kontrollierter Umgebung.

Mock Objects

- Problem: Zu testendes Objekt ist eng an einen Verband von anderen Objekten (seine *Kollaborateure*) gekoppelt.
- Idee: Schiebe dem zu testenden Objekt spezielle Testvarianten („Mock Objects“) unter.
- Beispiel: `de.lmu.ifi.mock.ServerTest`

Unit Tests für Tic-Tac-Toe

Ideen:

- Macht der Server das richtige? Mock-Player.
- Macht der Player das richtige? Mock-Server.
- Netzwerkbetrieb (auf Localhost) testen.

Prozess

Prozess

Software-Entwicklungsprozess: Steuere die Entwicklung von Software von der anfänglichen Idee bis zum Einsatz des fertigen Produkts und danach. Beispiele:

- ISO 12207: Definiert Phasen und Prozesse, damit Anbieter und Kunde besser verhandeln können.

- Capability Maturity Model Integration (CMMI): Prozessmodell des US-Verteitigungsministeriums zur Beurteilung der Qualität („Reife“) eines Softwareprozesses. [Nachfolger von CMM]

- Rational Unified Process (Rational Software, IBM): Eng mit UML verknüpft, relativ schwergewichtig.

- Extreme Programming: Leichtgewichtig („agil“), weniger Dokumentation, viel mündliche Kommunikation.

Extreme Programming (XP)

- Pragmatische Praktiken, die sich bewährt haben.
- Fokus auf Ausführbarkeit ("The code is the model (and vice versa)").
- Teamarbeit: Möglichst gleichmäßige Wissens- und Kompetenzverteilung im Team. Einzelgänger vermeiden.
- Offene, kontinuierliche Kommunikation mit dem Kunden.
- Inkrementelles Entwickeln, Vetrauen auf die Formbarkeit von Software: "You are not going to need it (YAGNI)" als Antithese zu Vorsorge und Erweiterbarkeit.

XP: Techniken

- User Stories (Requirements Engineering)
- Zeitplanung
- Pair Programming (Code Review)
- Spikes (Prototypen)
- Unit-Tests
- Iteratives Entwickeln

Teamarbeit

- Es wird viel Wert auf ein durchweg kompetentes Team gelegt. (Traditionell: Leichte Abwertung des „Programmierers“ und Aufwertung des „Designers“).
- Gruppe sollte immer komplett anwesend sein (=im CIP-Pool treffen).
- Vorteil: Unklarheiten können jederzeit beredet werden.

Pair Programming

- Original: Code Review. Code wird auf Verständlichkeit hin optimiert, wichtige Art einer echt „semantischen“ Überprüfung.
- Extemere Variante: Dauerndes Code Review
- Je zwei Leute programmieren zusammen: Fahrer + Beifahrer.
- Ab und an Fahrer wechseln, öfters Pause machen.
- Paare ändern sich oft, finden sich teilweise Aufgaben-spezifisch zusammen.

Pair Programming

Vorteile (bei $anfangs$ verminderter Entwicklungsgeschwindigkeit):

- Wissen wird gleichmäßig verteilt.
- Schnellere und bessere Entscheidungen.
- Weniger Fehler.
- Verständlicherer Code.
- Weniger Defensivität/Besitzanspruch bei dem Quellcode.

Spikes

- Deutsch: „Durchstich“
- Zur Verbesserung der Zeitschätzung.
- Kritische Dinge spielerisch ausprobieren, bevor es ans eigentliche Implementieren geht: Technik, Designprobleme etc.

Unit Tests: Vorteile

- Dokumentation
- Vermeidung manueller Arbeit
- Sicherheit, dass niemand den eigenen Code „torpediert“ bzw. dass man selbst nichts kaputt macht.

Iteratives Entwickeln

- Es gibt (fast) jederzeit ein ausführbares Produkt.
- Der Kunde gibt bereits früh Feedback.
- Planen in kleinen Funktionalitäts-Inkrementen.
- Weiterentwicklung ist stark vom direkten Bedarf getrieben.
- Minimiert das Risiko, dass falsche Funktionalität implementiert wird.

Referenzen

- Extreme Programming
  - `http://www.extremeprogramming.org/`
  - „Extreme Programming Installed“, Jeffries, Anderson, Hendrickson.

- Regular Expressions: Das JavaDoc zu `java.util.regex.Pattern` ist gut. Ansonsten: selbst eine Suchmaschine bemühen, es gibt unzählige Webseiten zu dem Thema.