Aspekte

Komponente: alles was sauber in Prozedur/Objekt gekapselt werden kann (Core-Level-Concerns)

Aspekt: anderes (System-Level-, Cross-Cutting Concerns) — Eigenschaften, die die Effizienz oder Semantik einer Komponente auf systematische Weise beeinflussen, wie z.B.

- Logging bzw. Profiling
- Ausgabe von Debug-Informationen
- Speicherung von Zwischenergebnissen
- konsistentes Verhalten
- Fehlerbehandlung, Persistenz, ...

AspectJ – Terminologie

Join-Point: wohldefinierter Punkt im Programmfluss

```
call(void C.x())
```

Pointcut fasst Join-Points zusammen, teilweise auch Werte

```
pointcut xy(): (call(void C.x()) || call(void C.y()));
```

Advice: Code, wird beim Erreichen eines Pointcuts ausgeführt

```
after(): xy() { System.out.println("xy") }
```

Aspect fasst Advices zu syntakitscher Einheit zusammen

```
aspect A { pointcut ...; after(): ... }
```

Aspect-Weaver macht aus Komponentencode (in Java) und Aspektcode den eigentlichen Java-Code

AspectJ – Beispiel

```
aspect CheckRange {
    pointcut set(MyClass o, int x): call(void o.set(x));

    before(): set(o, x) {
        if (x < MIN || x > MAX)
            throw new IllegalArgumentException(
                "x is out of bounds");
    }
}
```

Reflektive Programmierung

Reflektion: Sprache wird dynamisch analysiert oder verändert

- lässt dem Programmierer viele Freiheiten
- sehr fehleranfällig
- Basis für aspektorientierte Programmierung

Änderungen der Semantik eines Programms möglich durch

- Änderungen des Programms
- Änderungen der Sprachen bzw. Standardbibliotheken
- Parameterisierte Sprachen bzw. Bibliotheken

Alternativen zur Reflektion

Precompiler vor Programmübersetzung ausführen:

- effizient zur Laufzeit
- einfache Änderungen mit kleinem Aufwand
- größere Analysen können teuer sein
- Laufzeitinformation nicht verfügbar

Programmiersprache ändern oder erweitern

- nur für große Änderungen sinnvoll
- interne Informationen im Compiler wiederverwendbar
- nicht portabel (mit Ausnahmen)
- einfach, wenn Erweiterung nur mit Bibliotheken

Mixins

- Begriff unklar, etwas unterschiedliche Bedeutungen
- Mehrfach-Code+Interface-Vererbung (oder ähnlich dazu) mit konfigurierbarem geerbten Code (statisch/dynamisch)
- Anwendungsbeispiel: Template-Method geerbt und Operationen (der Template-Method) durch Mixins hinzugefügt
- Unterschiede zwischen Mehrfachvererbung und Delegation für Mixins in Praxis oft nicht entscheidend (asymmetrisch)
- Oberklassen-Typparameter (C++) u. Delegation geeignet
- Mixin-Oberklassen oft nur zwecks Delegation instanzierbar

Trait

- ein Trait stellt implementierte Methoden bereit
- ein Trait verlangt Methoden
- Traits haben und benutzen keine Instanzvariablen
- Reihenfolge des Zusammenfügens von Traits irrelevant
- Verwendung von Traits ist so als ob Methoden lokal in Klasse (oder Trait, der Traits verwendet) definiert wären
- Sprachen: Squeak, Scala, Perl 6 (Roles);
  Module-Mixins in Ruby ähnlich

Dependency Injection (DI)

- Objekte von außen übergeben, nicht lokal erzeugt
- Ziele: Abhängigkeit zwischen Client und Server verringern, *Boilerplate Code* vermeiden, einfache Konfigurierbarkeit
- Klassisch: *Inversion of Control* (z.B. Factory-Method)
- M. Fowler: DI = Service durch *Mini-Container* bestimmt
  - Interface Injection (von Client implementiert)
  - Setter-Injection (Mini-Container z.B. von Client befüllt)
  - Constructer-Injection (bei Erzeugung gesetzt)
- Problem: Flexibilität durch Unübersichtlichkeit erkauft