Aspekte

Komponente: alles was sauber in Prozedur/Objekt gekapselt werden kann (core-level concerns)

Aspekt: alles andere (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

OO Arten der Reflektion

- interne Klassenstruktur sichtbar machen (Metadaten)
- Klassen zur Laufzeit hinzufügen, Typinfo. verwenden
- Klassen dynamisch erzeugen bzw. verändern
- „message passing mechanism“ oder andere Sprachkonzepte der untersten Stufe sichtbar machen und verändern

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

Attribute in C#

- Attribut = Objekt, das mit Programmelementen verknüpfte Daten darstellt
- verknüpfbar mit Klassen, Methoden, Parametern, etc.:

```
[Serializable]
class MySerializableClass { ... }
```

- intrinsic (in .NET integriert) versus benutzerdefiniert
- Beispiel für Verwendung eines benutzerdef. Attributs:

```
[BugFixAttribute("Stefan", "1.4.2011", Comment="OK")]
class MyTestClass { ... }
```

- über Reflection zur Laufzeit zugänglich

Beispiel für Attributdefinition

```
[AttributeUsage(AttributeTargets.Class,AllowMultiple=true)]
public class BugFixAttribute : System.Attribute {
  public BugFixAttribute (string programmer, string date) {
    this.programmer = programmer;  this.date = date);
  }
  public string Comment {
    get { return comment; }
    set { comment = value; }
  }
  public string Programmer { get { return programmer; } }
  public string Date { get { return date; } }
  private string programmer, date, comment;
}
```

Annotationen in Java

- entsprechen ungefähr Attributen von C# – Beispiel:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
  String prog(); String date(); String comment();
}
...
@BugFix(prog="Stefan", date="1.4.2011", comment="OK")
class MyTestClass { ... }
```

- SOURCE Annotationen ersetzen javadoc Annotationen

Dynamische Erweiterung von Klassen

- dynamische Sprachen unterstützen Klassenerweiterung zur Laufzeit, statische Sprachen nicht direkt – Auswirkungen?
- Decorator erlaubt dynamische Erweiterung
- durch „final“ nur ohne Ersetzbarkeit verhinderbar

  $\rightarrow$ Erweiterbarkeit auch in statischen Sprachen Normalfall
- direkte Klassenänderung in dyn. Sprachen etwas effizienter als Decorator (Laufzeit + Programmieraufwand)
- statisches Konzept effizienter wenn keine Klassenänderung nötig (auch in dynamischen Sprachen)

Statische versus dynamische Typen

- einiges geht nicht statisch (z.B. Array-Index)
- z.B. dynamisch: Generizität mit dynamischen Parametern und alles was Ersetzbarkeit verletzt (CAT calls in Eiffel)
- tatsächlicher Unterschied:

  — Programmierer dynamischer Sprachen überlegen kaum, welche Information statisch ist → alles bleibt dynamisch

  — Anhänger stark typisierter Sprachen betreiben viel Aufwand, um so viel wie möglich statisch zu machen
- starke Typsysteme fast immer halbentscheidbar

  → Typkompatibilität erfordert bestimmte Typstruktur

  → Zusatzaufwand um bestimmte Struktur herzustellen

Meta-Klassen in stat./dyn. Sprachen

- dynamische Sprachen haben oft ein ausgeprägtes Meta-Klassen-Konzept, stark typisierte Sprachen eher nicht
- Hauptgrund:
  - — Reflektion in dynamischen Sprachen einfacher
    $\rightarrow$ z.B. für aspektorientierte Programmierung verwendet
  - — statische Sprachen verwenden dafür eher Precompiler
- Meta-Klassen auch in Java, C#, C++, . . .
  - — oft nur als statische Methoden/Variablen betrachtet

C++ Templates

- auch Konstanten als Typparameter:

```
template<typename T, int n> class buffer { ... };
```

- partielle/volle Spezialisierung mit „pattern matching“

```
template<int n> class buffer<bool, n> { ... };

template<> class buffer<bool, 0> { ... };
```

- implizite Instanzierung wenn keine passende Spezialisierung
- Methoden nur erzeugt wenn im Code verwendet

Ausdrucksstärke von Templates

- statische Auswertung von Ausdrücken verwendbar:

```
template<int x, int n> struct power {
  static const int r = x * power<x, n-1>::r;
};
template<int x> struct power<x, 0> {
  static const int r = 1;
};
```

- auch algebraische Datentypen ausdrückbar:

```
template<class Head, class Tail> struct Cons { };
struct Nil { };
typedef Cons<int, Cons<float, Nil> > list_of_types;
```

Policy-Based Programming

```
#include <iostream>
template<typename lang> class HelloWorld : public lang {
   public: void Run() { cout << Message() << endl; }
};
#include <string>
class HelloWorld_Msg_German {
   protected: string Message() { return "Hallo Welt!"; }
};
typedef HelloWorld<HelloWorld_Msg_German> MyHelloWorld;
MyHelloWorld hello;
hello.Run();
```

Beispiel für Strategy (Design Pattern)

```
interface IMessage { String message(); }
class DMessage implements IMessage {
    public String message() {return "Hallo Welt!";}
}
class HelloWorld {
    private IMessage msg;
    public HelloWorld(IMessage msg) {this.msg = msg;}
    public void run() {System.out.println(msg.message());}
}
class Main {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld(new DMessage());
        hello.run();
    }
}
```