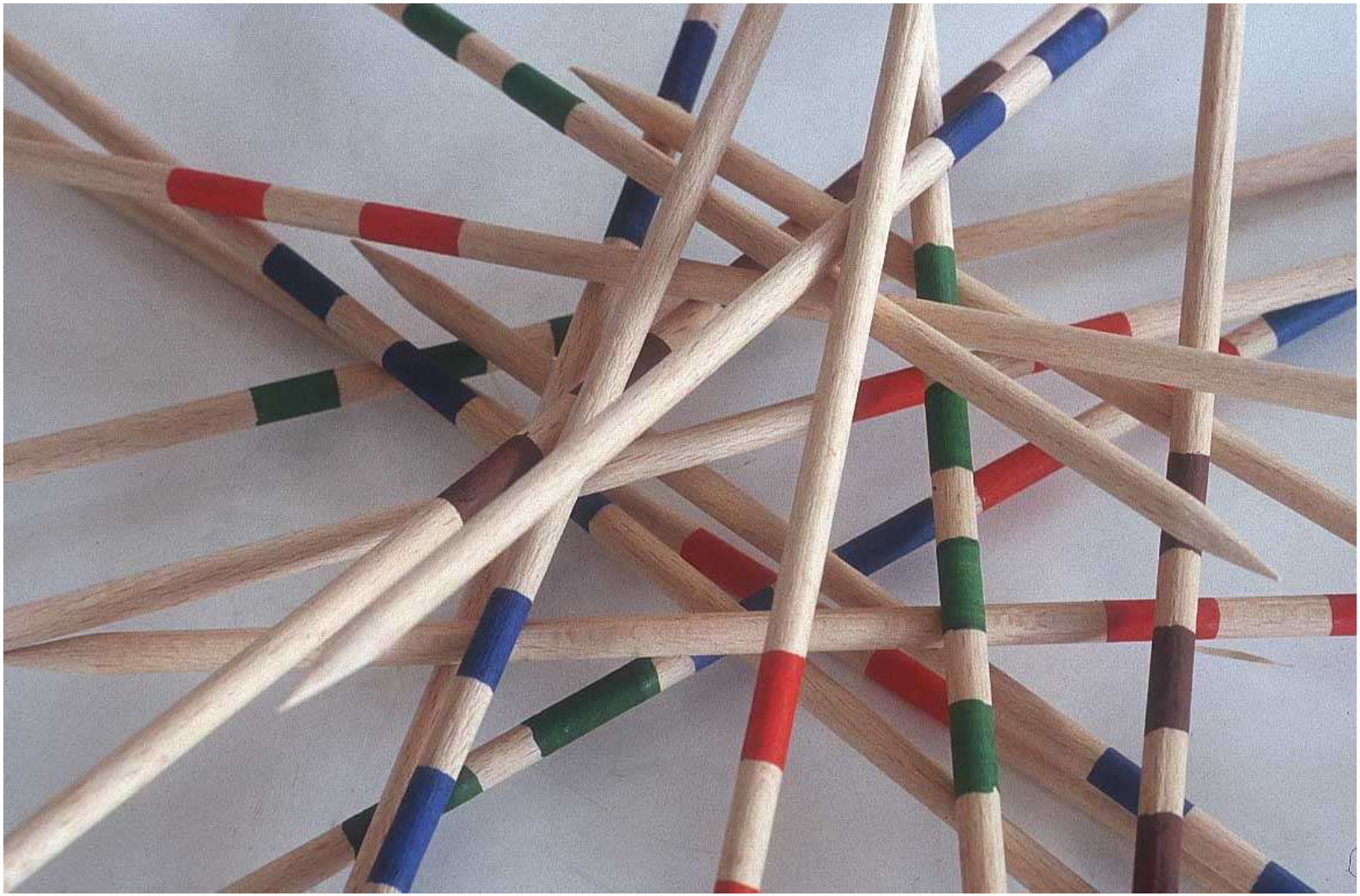
Aspekte in der Softwareentwicklung

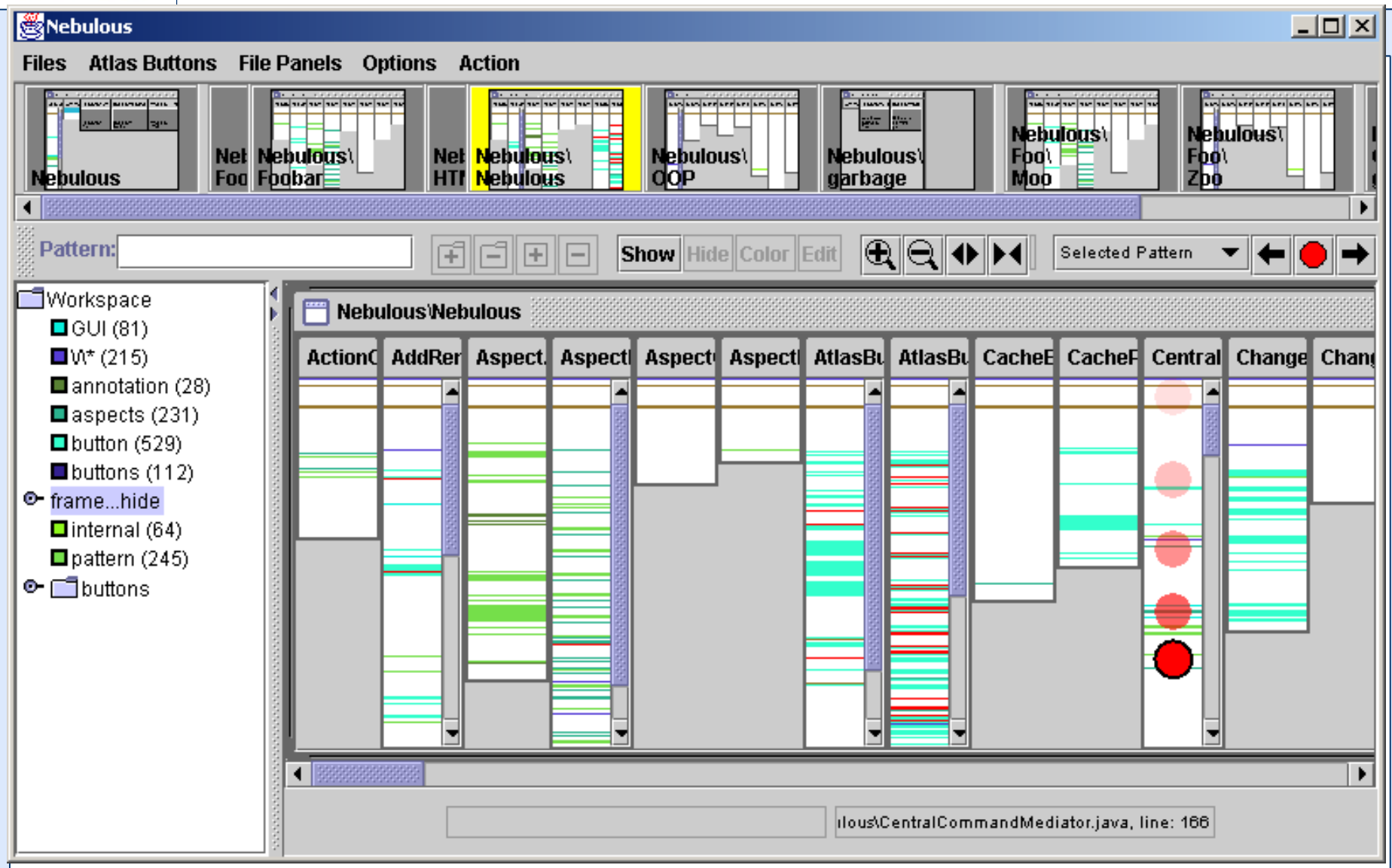
Stefan Jähnichen,
Stephan Herrmann,
Katharina Mehner

Ringvorlesung
„Modellbasierte Softwareentwicklung“
SoSe 2004, Humboldt Universität

Wartungsproblematik

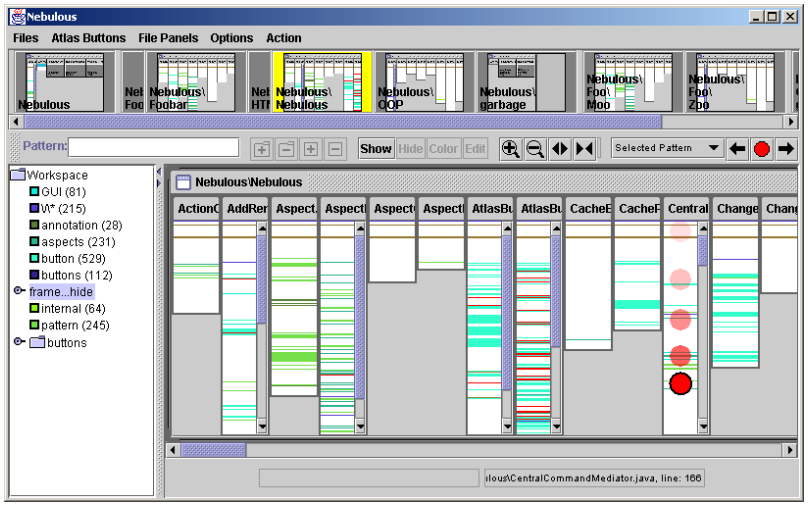


AspectBrowser



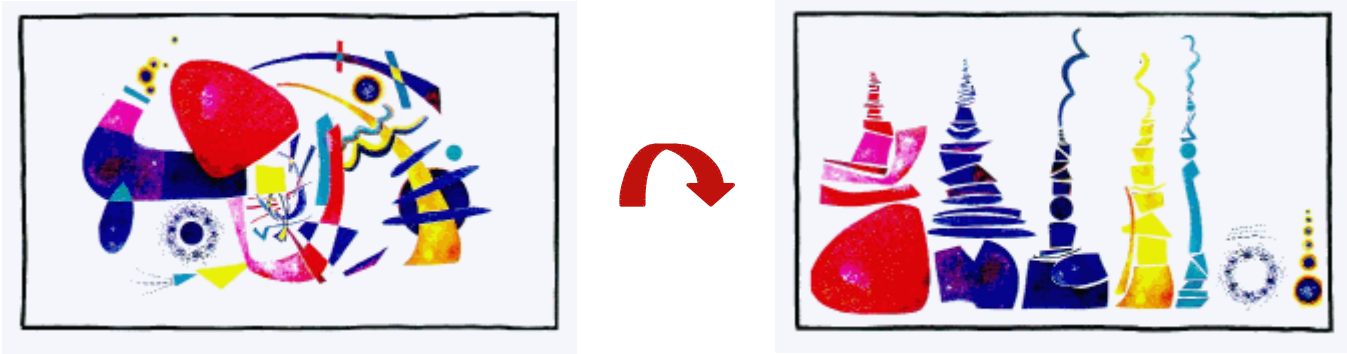
Crosscutting Concerns

- **Problemanalyse**
  - **Tangling**: Vermischung verschiedener Anforderung in einem Modul
    - Schwierig, Modul zu ändern
  - **Scattering**: Verstreuung einer Anforderung über ein Modulgeflecht
    - Schwierig, Aspekt zu ändern
- **Problemklasse Aspekte**
  - **Crosscutting Concerns**
    - Logging
    - Synchronisation
    - IT-Sicherheit
    - Persistenz
    - Caching
    - Profiling
    - Verteilung ...



Aspektorientierte Programmierung

- **Erweiterung objektorientierter Programmier-sprachen**
- **Trennung von Aspekt und Basisfunktionalität**
  - Aspektmodule für querschneidende Anforderung

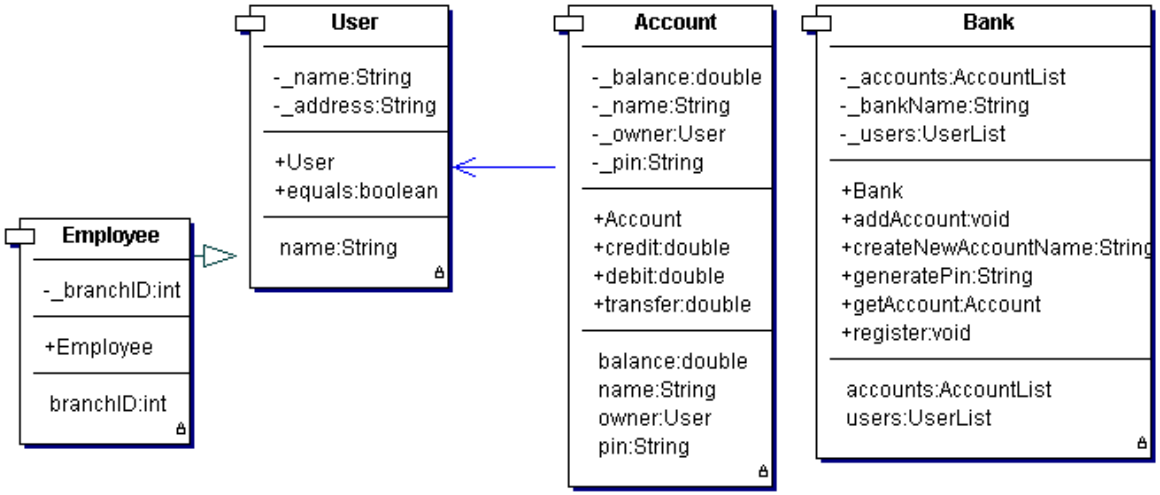


©Ursus Wehrli

- **Problem: Bezüge sind nicht mehr zu erkennen**
- **Integration**
  - Weben: Integration von Klassen und Aspekten

Beispiel Sicherheit

- **Basisfunktionalität:**



- **Authentisierung:**
  - Nur angemeldete Benutzer dürfen Operationen auf Konten ausführen
- **Authorisierung:**
  - Unterschiedliche Benutzerrollen haben verschiedene Zugriffsrechte für Konten

	getBalance	debit	credit	transfer
Owner	x	x	x	x
Employee	x	o	x	x
User	o	o	x	o

Authentisierung in AspectJ

```
public aspect Authentication
{
    pointcut requireAuthentication() : execution(public * Account.* (..));

    before() : requireAuthentication()
    {
        if (LoginContext.getInstance().getUser() == null)
            authenticateUser(); //Benutzer anmelden
    }
}

// Benutzer wird an anderer Stelle wieder abgemeldet
```

Prädikat über Ausführungspunkte

Ausführungen einer Methode

„bei welchen Ausführungen“

„vor“ diesen Ausführungen

Zusätzlich auszuführender Code

Authorisierung in AspectJ

```
public aspect Authorization
{
    declare precedence : Authentication, Authorization;
    //Au                rwendet Ergebnis der Authentisierun

    pointcut requireOwner(Account acnt) :

                execution(public * Account.debit(..))
                && !cflowbelow(execution(public * Account.*(..)))
                && target(acnt);

        before(Account acnt) : requireOwner(acnt)
        {

            System.out.println("RequireOwner");
            if (!LoginContext.getInstance().getUser().equals(acnt.getOwner()))
                {
                throw new AccessDeniedException("No valid user logged in!");
                }
        }
```

Reihenfolge

Parameter

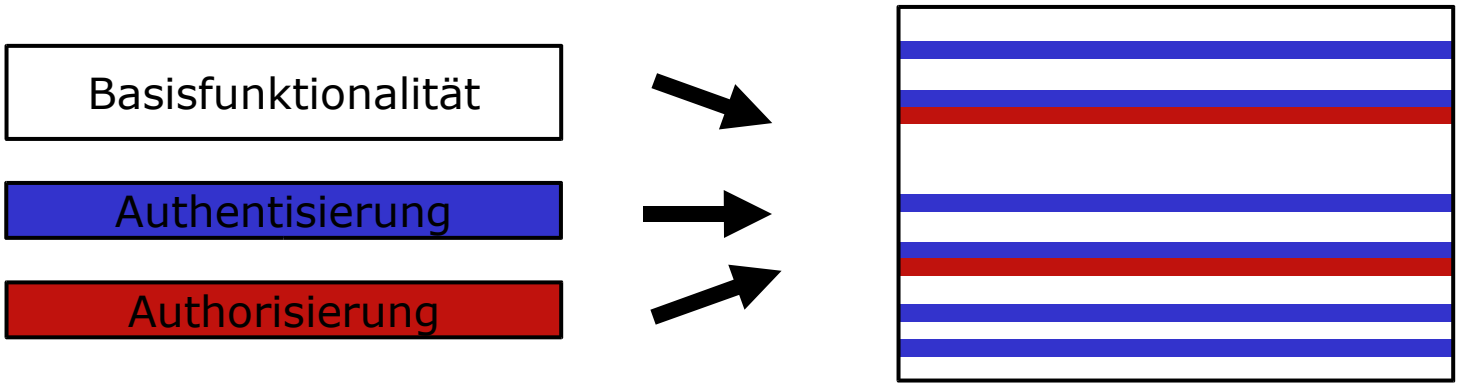
Nicht, wenn debit aus einer anderen Methode von Account aufgerufen wurde

Bindung

Owner Überprüfung

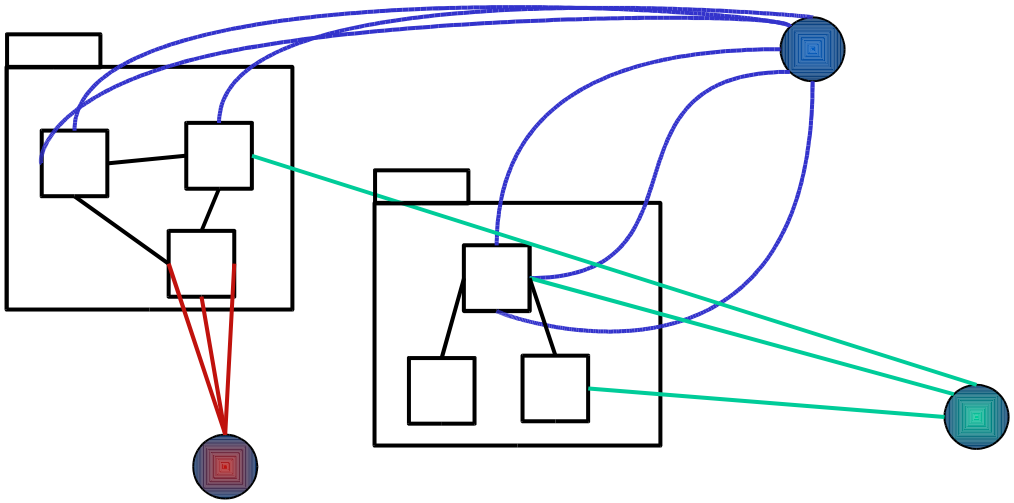
Integration

- **Compilezeit**
  - **Statisches Weben** (engl. Weaving)
    - Übersetzen der Aspekte nach Java
    - Transformation der Basisklassen



- **Laufzeit**
  - Ausführung auf einer Standard JVM
- **Varianten**
  - Dynamisches Weben zur Laufzeit

Aspekt-Strukturen

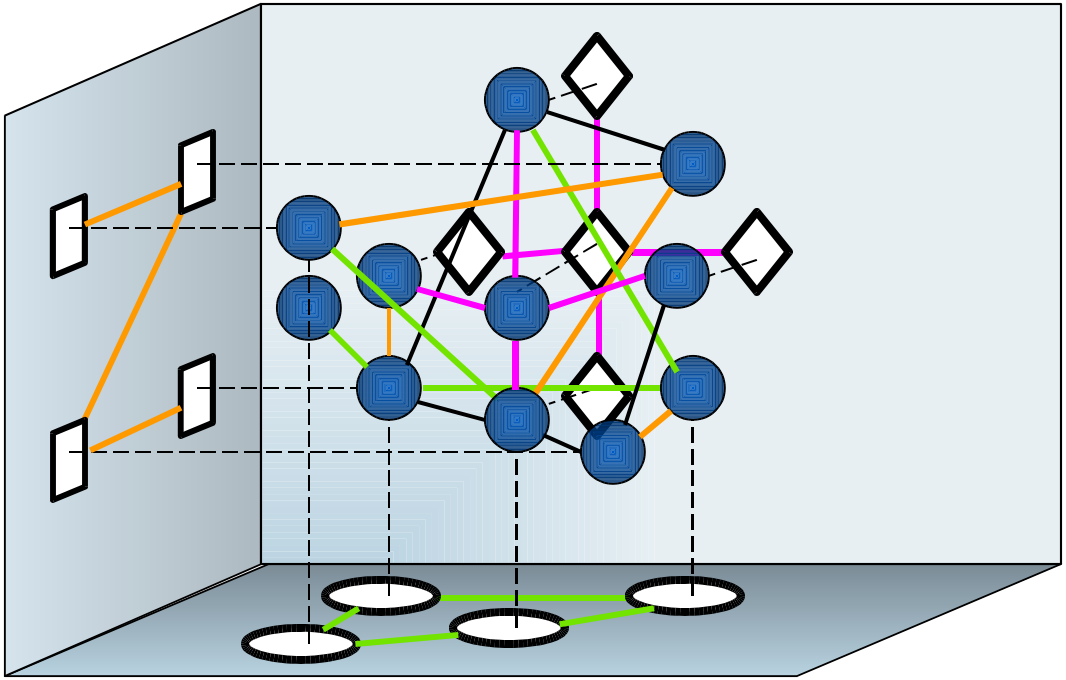


- **Aspekt**
  - Wenig Zusammenhang
  - Wirksam an vielen Stellen
- **Geeignet für eine bestimmte Klasse von Anforderungen**
- **Basisklassen müssen nicht geändert werden**

Defizite

- **Keine Ausdrucksmöglichkeit für Abhängigkeiten**
  - Aspekt setzt anderen Aspekt voraus
  - Aspekte schliessen sich gegenseitig aus
- **Keine Unterstützung für**
  - Stark strukturierte Anforderungen
    - Komplexe Workflows
    - Kollaborationen
  - Wiederverwendung von Aspekten
  - Aspektkomposition
- **Allgemeinere Lösung**
  - Symmetrie von Aspekten und Klassen
    - Komposition
    - Vererbung
    - Instanziierung

Separation of Concerns



- **Komplexe Relationen beherrschen**
- **Getrennte Definition von Sichten**

Sichten als Sprachkonzept

- **Modularisierung der Anforderungen nach Sichten**
  - Funktional (z.B. OOA Use Cases, Klassendiagramm, …)
    - Struktur und Verhalten
  - Nicht-funktional
    - Spezielle Sicht auf Struktur und Verhalten
- **Modularisierung im Design nach Sichten**
  - Nach Form
    - Struktur, Dynamik, Funktion **[OMT]**
  - Nach Inhalt
    - Kollaborationsbasiertes OO-Design mit Rollen **[Catalysis, OORAM]**
- **Objektorientierte Programmiersprachen**
  - Nach Form (Klassen = Einheit von Daten und Methoden)
  - Fokus auf Algorithmik

Nahtlose Entwicklung?

- **Paradigmen und Medienbrüche**
  - → Übergänge erfolgen nur als Gesamtheit
  - → schlecht erweiterbar
  - → schlecht umkehrbar
  - → Erweiterungen zerstören Strukturierung
- **Durch OO nur teilweise nahtloser Übergang**
  - ✓ OOD-> OOP „nahtlos“,
  - ✗ OOA->OOD problematisch
    - OOA: Use Cases und Aktivitätsdiagramme getrennt von Objekten
    - OOD: Objektorientierte Methoden
    - Im Allgemeinen keine Modularisierung von Kollaborationen

Herausforderung

- **Durchlässigkeit erreichen durch ein einheitliches Paradigma**
- **Übergänge zwischen Modellen müssen durchlässig sein**
  - Nahtloser Übergang zwischen Artefakten verschiedener Aktivitäten
    - Inkrementell
    - Bidirektional
    - Beliebig oft
  - Analoge Strukturierung/Modularisierung der Artefakte
- **Querbezüge und Sichten-Integration innerhalb eines Modells**
  - Möglichkeit, um Querbezüge zu beschreiben
  - Vollständigkeit
- **Paradigma der Sichten soll sich in allen Modellen wiederfinden**
  - „Traditionelle Aspekte“
  - Kollaborationen

Komplexe Aspekte

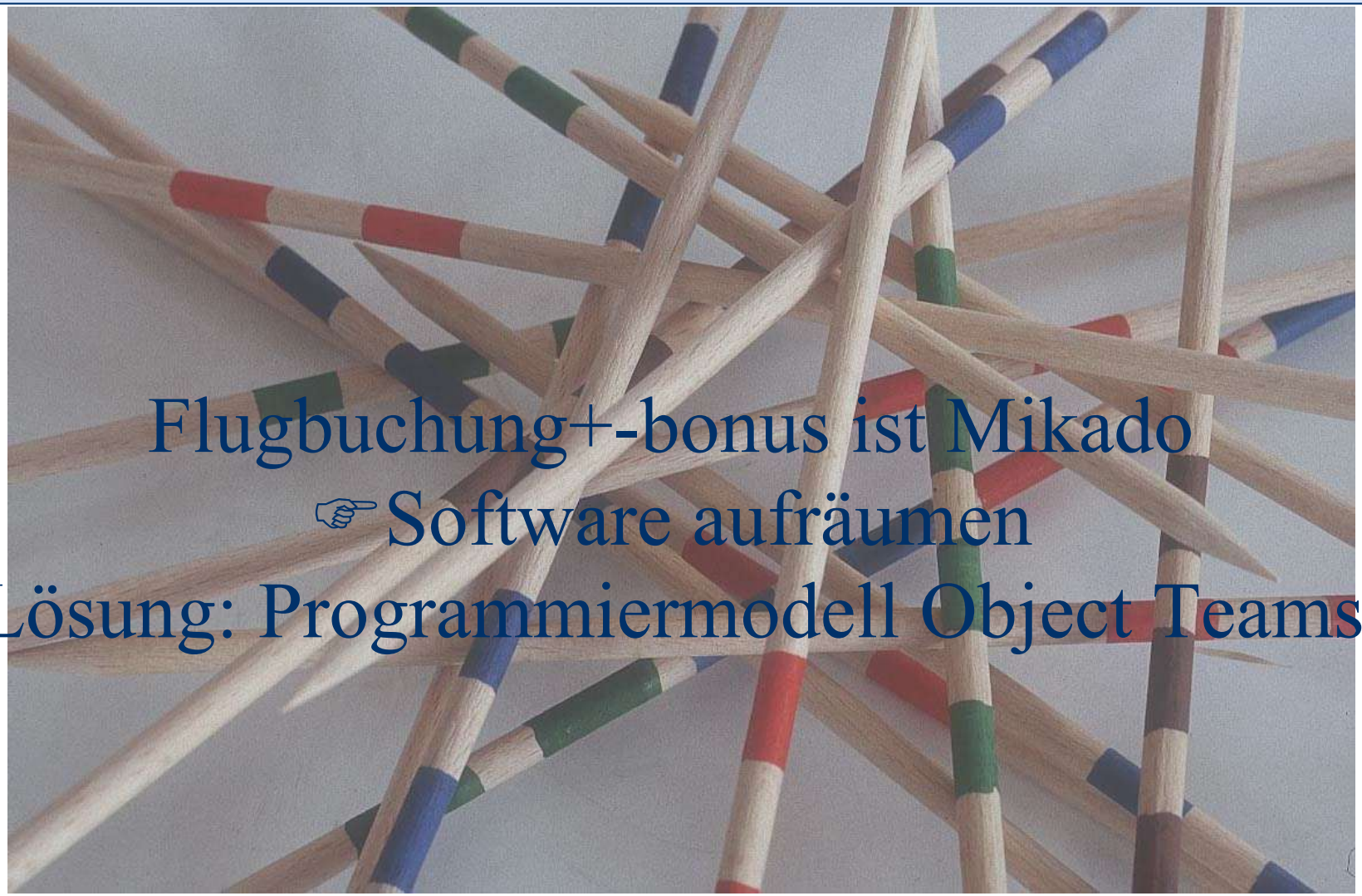
- **Beispiel: Geschäftsfälle**
  - **Flugbuchung**
  - **Bonusprogramm**
- **Konventionelle Lösung**

  Jede Buchungsoperation muß
  - Buchung durchführen
  - Bonusmeilen gutschreiben
- **Problem**
  - Geschäftsfälle sind **nicht unabhängig**
  - Kaskaden von Fallunterscheidungen
    - Welche Fluggesellschaft?
    - Passagier registriert?
    - Welcher Status?

Wiederverwendung??

Wartung??

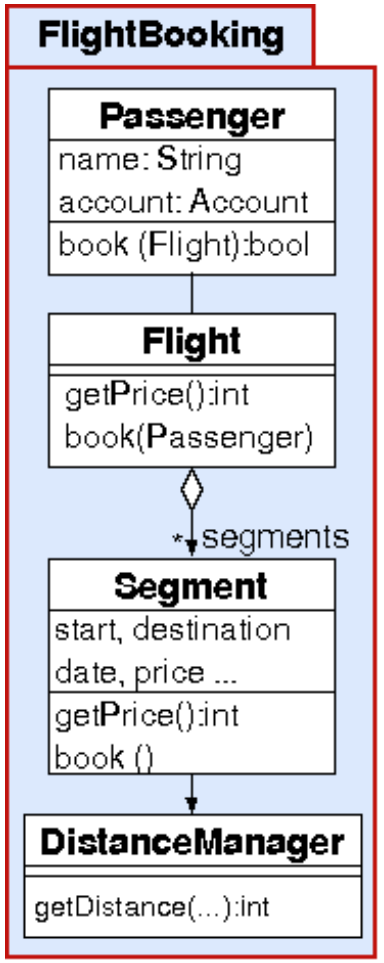
Verständnis??



Flugbuchung+-bonus ist Mikado
☞Software aufräumen
Lösung: Programmiermodell Object Teams

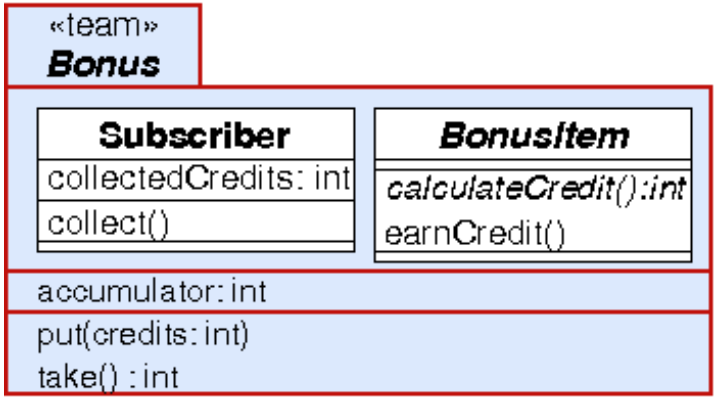
Kollaborationsmodule (1)

- **FlightBooking**
  - Gewöhnliches Paket
  - Abgeschlossen
  - Lauffähig
  - Soll nicht verändert werden!



Kollaborationsmodule (2)

- **Bonus**
  - Unvollständiges Paket
  - Enthält verschiedene **Rollen**
  - **Interaktion** zwischen Rollen
  - Kollaboration kann instanziiert werden



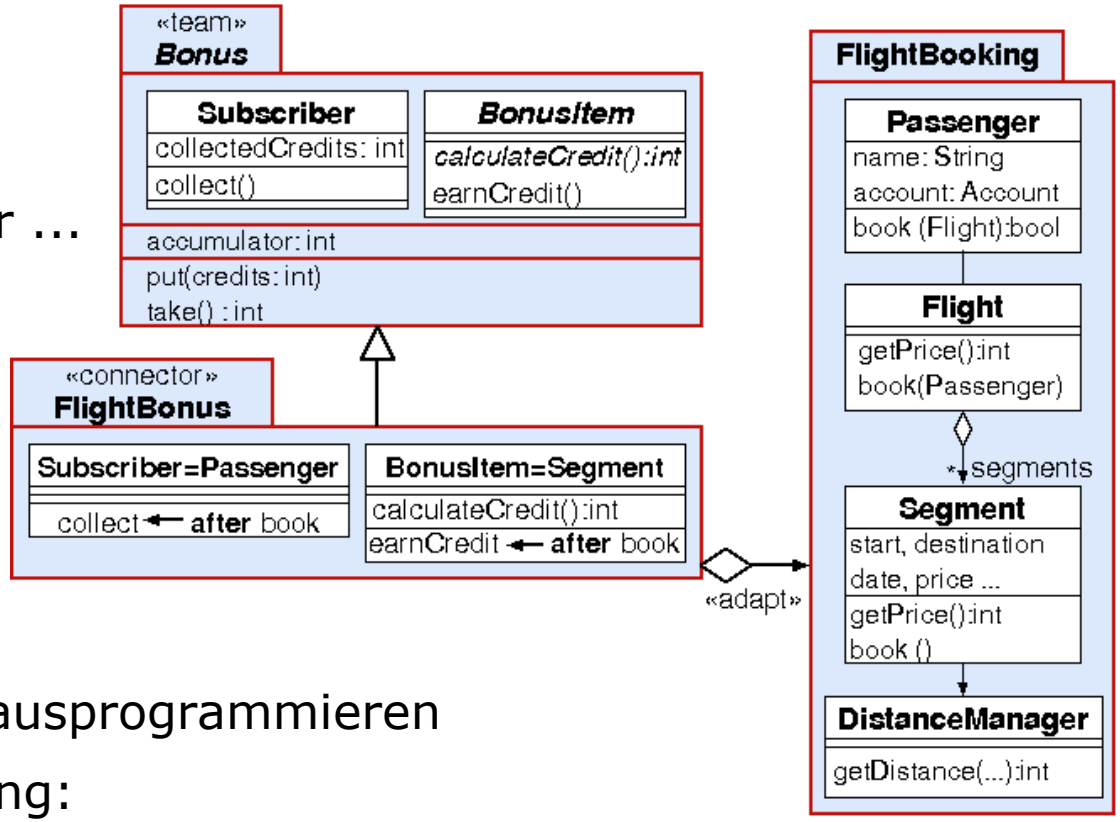
Team!

```
public team class Bonus {
    class Subscriber { ... }
    class BonusItem { ... }
    private int accumulator;
}
```

A-Posteriori-Integration

- **Integration**

  Zerlegen ist einfach, aber ...



- **Konnektor**
  - Weitgehend deklarativ
  - Sonderfälle imperativ ausprogrammieren
  - Drei Ebenen der Bindung:
    - Klassen
    - Methoden
    - Parameter

Flexible Kopplung

- **Rolle-Basis-Beziehung „playedBy“**
  - Unabhängigkeit
    - Basis „ignorant“
    - Beliebige Anzahl Rollen pro Basis
  - Integration
    - Rolle + Basis = Konzeptionelle Einheit
    - Objektbasierte Vererbung
    - Realisiert u.a. durch unsichtbaren Link Role→Basis
- **Rollen sind Aspekte der Basis**
  - Vergleiche auch: Entwurfsmuster Decorator!

```
class Subscriber playedBy Passenger { ... }
class BonusItem playedBy Segment { ... }
```

Bidirektionale Schnittstelle

- **Analogie CORBA Components:**
  - expected/provided interface
- **Zwei Richtungen von Methodenbindungen**
  - Aus Sicht der Rolle
    (Basis bleibt ignorant)
- **Vollständige Interaktion zwischen Paketen**
  - Wo ruft das Team Basisfunktionalität auf?
  - Wo läßt sich das Team aus der Basis heraus aufrufen?

Forwarding: Callout

- **Rolle leitet Aufrufe an Basis weiter**

```
class BonusItem playedBy Segment {
   getPoints -> getPrice;
}
```

- **Empfänger der Nachricht bleibt implizit**
  - Rolle-Basis-Link nicht direkt zugreifbar
  - Vermeidung von Inkonsistenzen
- **Parametermappings**

  Falls Signaturen nicht passen
  - Deklarative Abbildung von Parametern und Resultat
  - Kann einfache Berechnungen enthalten

Aspektbindung: Callin

- **Einflechten von Triggern in die Basis**

```
class BonusItem playedBy Segment {
   collectCredit() <- after book;
}
```

Effekt:

- Nach jeder Ausführung von `Segment.book()` wird `BonusItem.collectCredit()` aufgerufen

- **An welchem Objekt?**
  - Implizites Aufsuchen „des richtigen“ Rollenobjektes.
  - Mechanismus „lifting“
  - Vollständig automatisiert

Dynamische Aktivierung

- **Teaminstanzen**
  - Repräsentieren Bonusprogramme versch. Fluggesellschaften
  - Einzeln aktivieren/deaktivieren
  - Passagiere einzeln bei Teaminstanzen registrieren
- **Aktivierung bedeutet**
  - Einschalten aller callin-Bindungen des Teams
  - Deaktiviertes Team ist als Aspekt wirkungslos

Teamvererbung

- **Konsistente Verfeinerung einer Kollaboration**
  - Virtuelle Klassen, Class Overriding, Family Polymorphism (typsichere Kovarianz!)
  - Kann als **Framework**-Vererbung eingesetzt werden
  - Verfeinerung kann als **Konnektor** fungieren
  - Beliebige Mischformen:
    - Hinzufügen von Implementierung
    - Hinzufügen von Bindungen
- **Neue Form von Wiederverwendung**

  Beispiel:
  - **FlightBonusVIP** als Spezialisierung von FlightBonus
  - Komplette Kollaboration wird geerbt (Rollen und ihre Interaktionen)
  - Gezielte Anpassungen im Team und/oder seinen Rollen möglich

Was Teams noch leisten

- **Vereinigung Paket & Klasse**
  - **Dateistruktur** + Objektstruktur
  - Eigene Attribute und Methoden
  - Vererbung
- **Kapselung**
  - Rollen können effektiv vor Zugriff geschützt werden
  - Team ist **Fassade**
  - Techniken der *Alias Control*:
    Typsystem berücksichtigt Instanzen

Object Teams im Kontext

- **Implementierung**
  - **Compiler**

    1. Version: erprobt in Diplomarbeiten und einer Lehrveranstaltung

    2. Version: Erweiterung des Java Compilers von Eclipse
  - **Laufzeitumgebung**

    Load-time weaving: Späte Aspekt-Bindung
  - **Entwicklungsumgebung**

    Eclipse-Erweiterung:
    - Editieren
    - Compilieren
    - Navigieren
    - ...

Object Teams im Kontext (2)

- **Praxiseinführung**

  Verbundprojekt **TOPPrax** (TUB, TUD, FIRST, GEBIT, Daedalos)

  - Evaluierung in vergleichenden Fallstudien
  - Konsolidierung von Konzepten und Werkzeugen
  - Umfassende Entwicklungsmethode
  - Bewertung: Hilft AOP mit Object Teams für
    - Qualität
    - Verständlichkeit
    - Wartung und Evolution

?