Hyper/J™

Ein Werkzeug zur
Multidimensional Separation of Concerns
für Java™

Klaus L. Greulich
greulich@cs.bonn.edu

Was sind *Concerns* ?

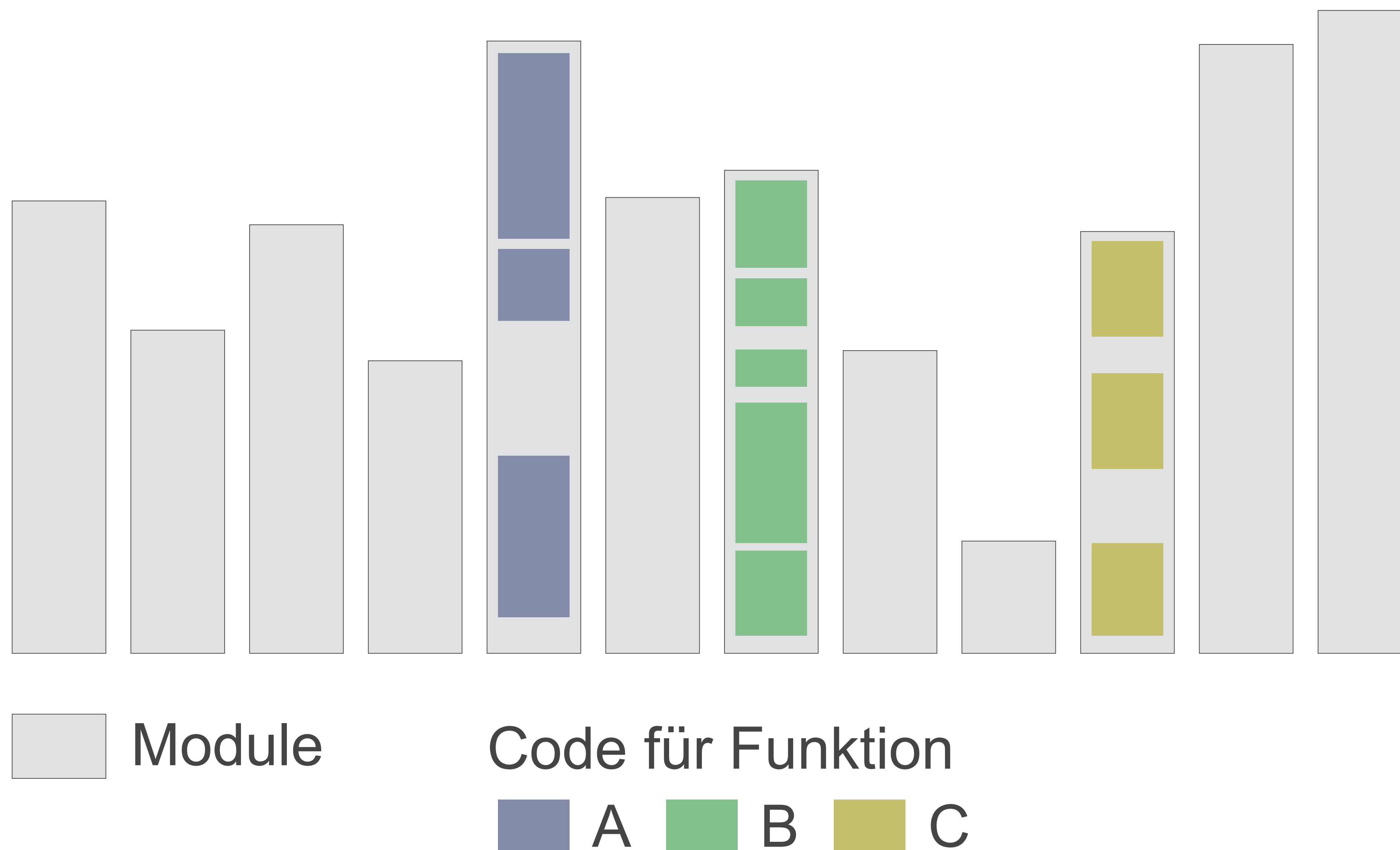
- Unter *Concern* versteht man
  - bestimmtes Ziel
  - ein Konzept
  - einen Interessenbereich
- Beispiel: ein Programm zum Verwalten von Kundendaten
  - Haupt-*Concern*: Verwalten der Kundendaten
  - Neben-*Concerns*:
    - Logging
    - Authentifizierung
    - Sicherheit
    - etc.

Wo liegt das Problem?

- Manche *Concerns* können gut mit den “traditionellen” Mitteln der Softwaretechnologie implementiert werden:
  - Kapselung in Objekten
  - Vererbung
  - Design Patterns
- Aber: viele dieser *Concerns* beeinflussen die Implementierung in verschiedenen Programm-Modulen.
- Die Folgen: das System bzw. Programm ist
  - schwieriger zu verstehen
  - schwieriger zu implementieren
  - schwieriger zu verbessern

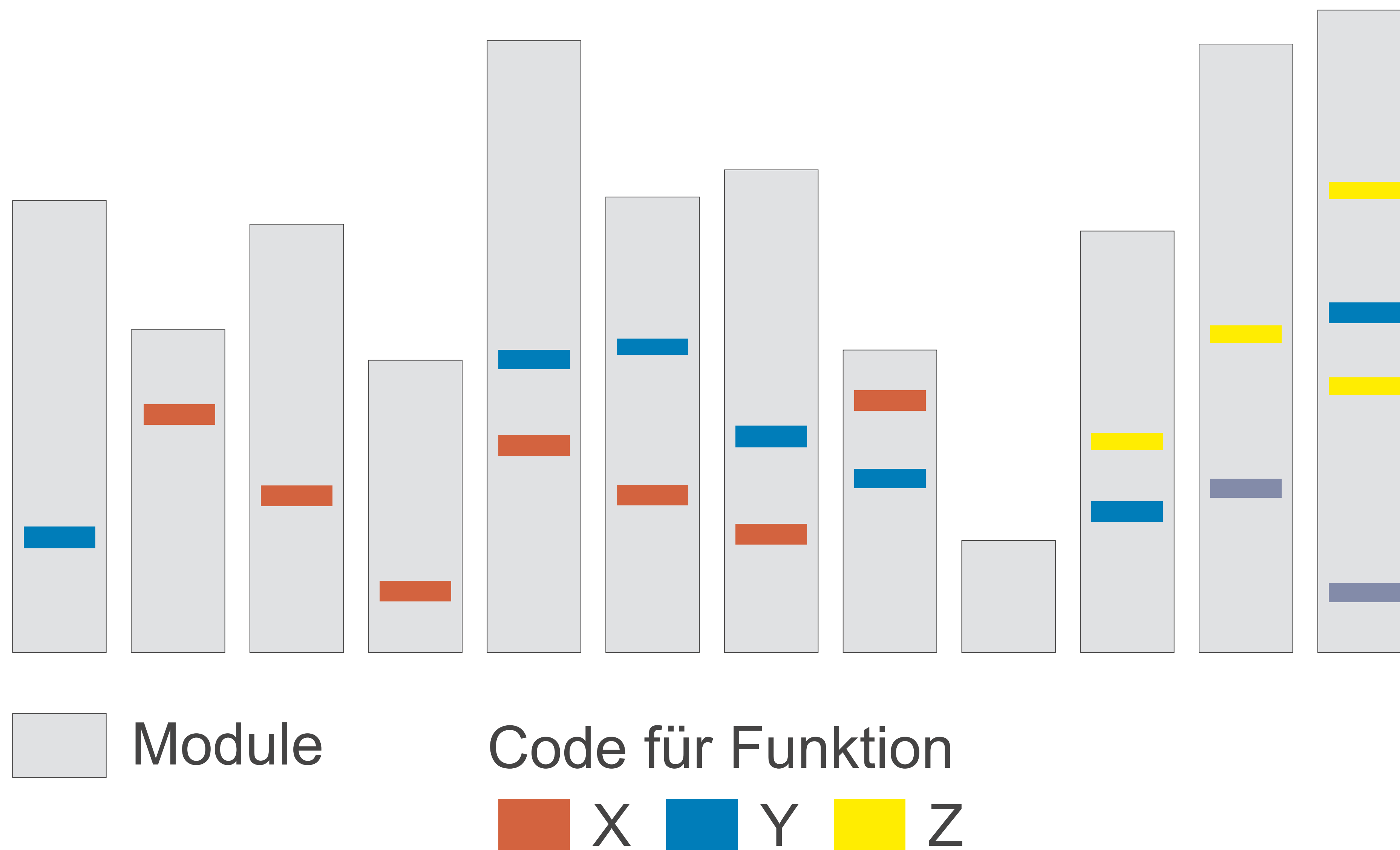
Concerns in Klassen implementiert

Die Funktionen A, B und C lassen sich hier gut in Module (z.B. Klassen) kapseln:



Concerns an verschiedenen Stellen

Die Funktionen X, Y und Z lassen sich nicht gut in Module kapseln:



Cross-Cutting-Concerns

- Concerns, die die Implementierung in verschiedenen Modulen beeinflussen, nennt man Cross-Cutting-Concerns
- Bei Nutzung bisheriger Methoden der SWT ergeben sich folgende Nachteile:
  - Module beschäftigen sich gleichzeitig mit unterschiedlichen Anforderungen, mehrere Concerns werden in einem Modul implementiert.
(**Code Tangling**)
  - Die Implementierung eines Concerns findet in unterschiedlichen Modulen statt.
(**Code Scattering**)

Folgen von CCC für den Code:

... bei Nutzung bisheriger Techniken

- Eventuell schlechter Code
  wenn zuviele Concerns in einem Modul behandelt werden.
- Vererbung vielleicht nicht möglich oder schwierig,
  da sich das Modul nicht nur um eine Anforderung kümmert.
- Mehr Aufwand,
  da Implementierungen mehrmals vorgenommen werden.
- Viel mehr Aufwand in der Zukunft,
  da die Implementierungen der CCC nicht modularisiert sind, muß Code an vielen Stellen geändert werden. Sollen neue Concerns berücksichtigt werden, wird die Implementierung vielleicht noch viel schwieriger.

Eine Lösung des Problems: AOP

- AOP ist eine Methode, um *Separation of Concerns* zu implementieren.
- OOP implementiert gemeinsame Aufgabenstellungen (**Common Concerns**) in **Klassen**
- AOP implementiert systemweite Aufgabenstellungen (**Crosscutting Concerns**) in **Aspekten**

Drei Entwicklungsschritte

- Aufteilen der Concerns in
  - Concerns auf Modul-Ebene
  - Concerns auf System-Ebene
- Implementierung der Concerns
  - auf Modul-Ebene in Klassen
  - auf System-Ebene in Aspekten
- Zusammenführen der Implementationen
  - durch ein geeignetes AOP-Werkzeug

Beispiel: Personal-Datenbank

- Aufteilen
  - Modul-Ebene: Verwaltung der Mitarbeiter
  - System-Ebene: Logging
- Implementieren
  - Verwaltung in Klassen
  - Logging in Aspekten
- Zusammenführen
  - Spezifizierung: vor und nach jeder Operation soll Logging stattfinden

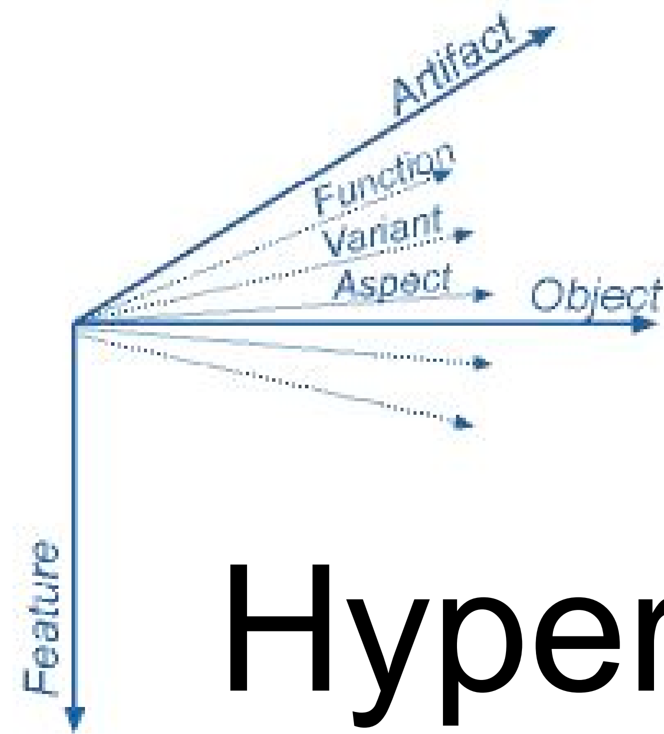
Wie wird AOP umgesetzt?

- Implementierung der Concerns
  - in Modulen in “traditionellen” Sprachen wie C, C++ oder Java
  - Objekt-Orientierung ist hier nicht Voraussetzung
- Festlegung der Regeln für das Zusammenführen der Aspekte
  - Benutzt eine Sprache, um Regeln zu bestimmen, die das System aus seinen einzelnen Komponenten (z.B. Klassen und Aspekten) zusammenfügt.
  - Diese Sprache kann eine Erweiterung der Basis-Sprache (z.B. Java) oder auch etwas völlig anderes sein.

Vorteile von AOP

- Modularisierte Implementierung von Crosscutting Concerns
  - kein Code Tangling
  - kein Code Scattering
- System ist leichter zu warten und zu erweitern
  - da jeder Concern separat und zentral implementiert ist
- Code läßt sich leichter wiederverwenden
  - da nicht mehrere Anforderungen gleichzeitig in einem Modul implementiert sind

So weit, so gut ...



Hyper/J™

Multi-Dimensional Separation of Concerns

Hyper/J
Multi-Dimensional Separation of Concerns

- Mehrere, eigenständige Dimensionen von *Concerns*
- Gleichzeitige Trennung entlang dieser Dimensionen
  - Keine dominante Dimension soll die Trennung entlang anderer Dimensionen verhindern
- Neue *Concerns* und neue Dimensionen von *Concerns* können dynamisch behandelt werden
  - z.B. wenn sie während der Entwicklung eines Produkts hinzukommen

MDSOC

- Die *Multi-Dimensional Separation of Concerns* ermöglicht:
- *On-Demand Remodularization*
- Entwickler können die beste Zusammensetzung wählen
- unter Berücksichtigung einiger oder aller *Concerns*
- zu jeder Zeit

Begriffe

- Hyperspace
  - mehrdimensionaler Concern-Space (Matrix)
- Units
  - syntaktisches Konstrukt einer Sprache
- Hyperslices
  - deklarativ komplette Mengen von Concerns
- Hypermodules
  - Menge von Hyperslices

Hyperspaces

- *Hyperspaces* ermöglichen die explizite Identifikation jeder Dimension und jedes *Concerns*, zu jeder Zeit
- *Hyperspaces* kapseln die *Concerns*
- *Hyperspaces* identifizieren und managen die Beziehungen (*relationships*) unter diesen *Concerns*
- *Hyperspaces* identifizieren und managen die Integration der *Concerns*

Units

- Eine *Unit* ist ein syntaktisches Konstrukt in einer Sprache, z.B.
  - Deklaration
  - Statement
  - Klasse
  - Interface
- *Primitive Units*
  - z.B.: Methode, Instanz Variable
- *Compound Units*
  - z.B.: Klasse, Package

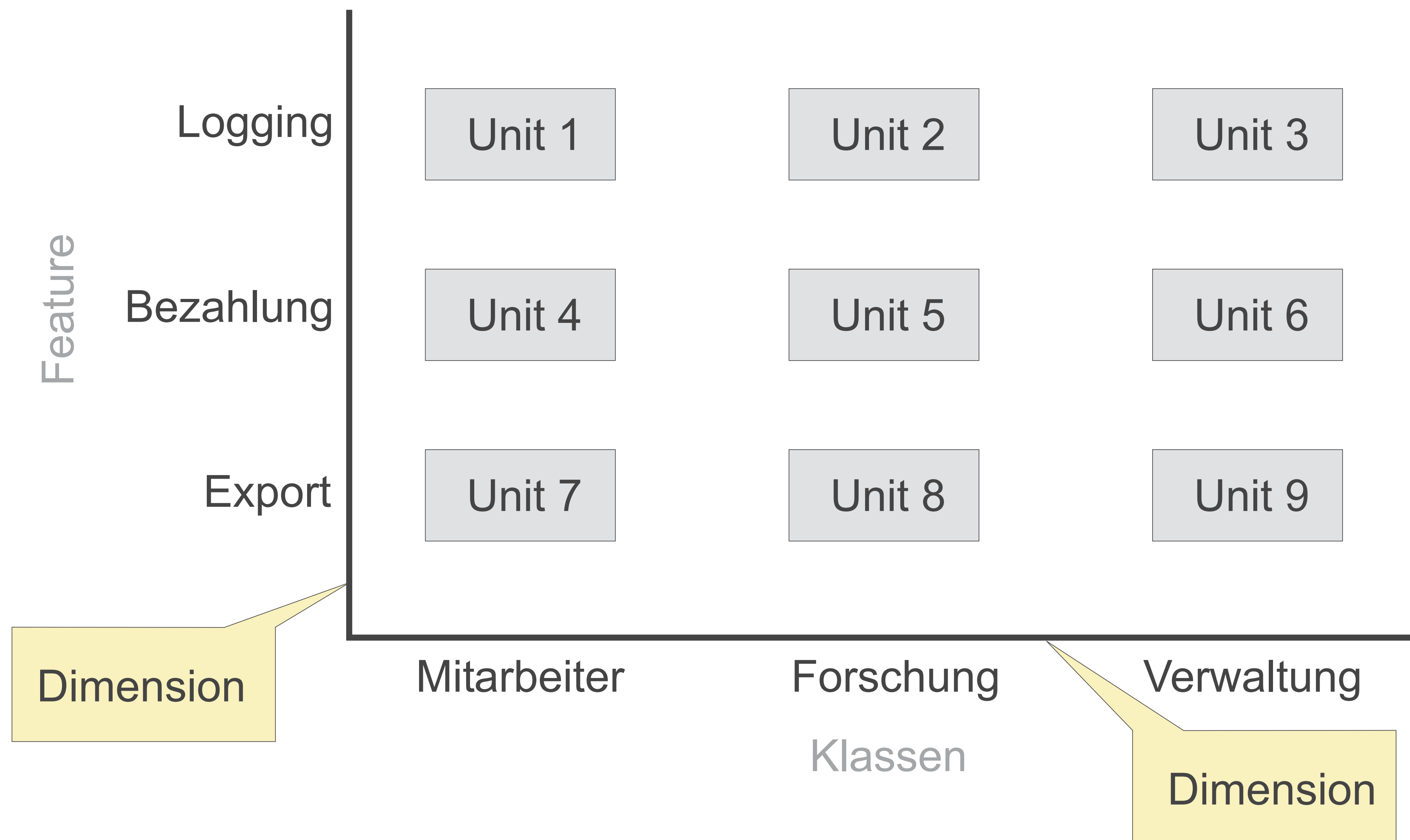
Dimensions of Concerns

- Es existieren viele Arten von *Concerns,* z.B.:
  - Feature (z.B. Export, Bezahlung, Logging)
  - Klasse (z.B. Mitarbeiter, Forschung)
  - Aspekte (z.B. Verteilung, Übereinstimmung)
- Diese werden *Dimensions of Concerns* genannt.

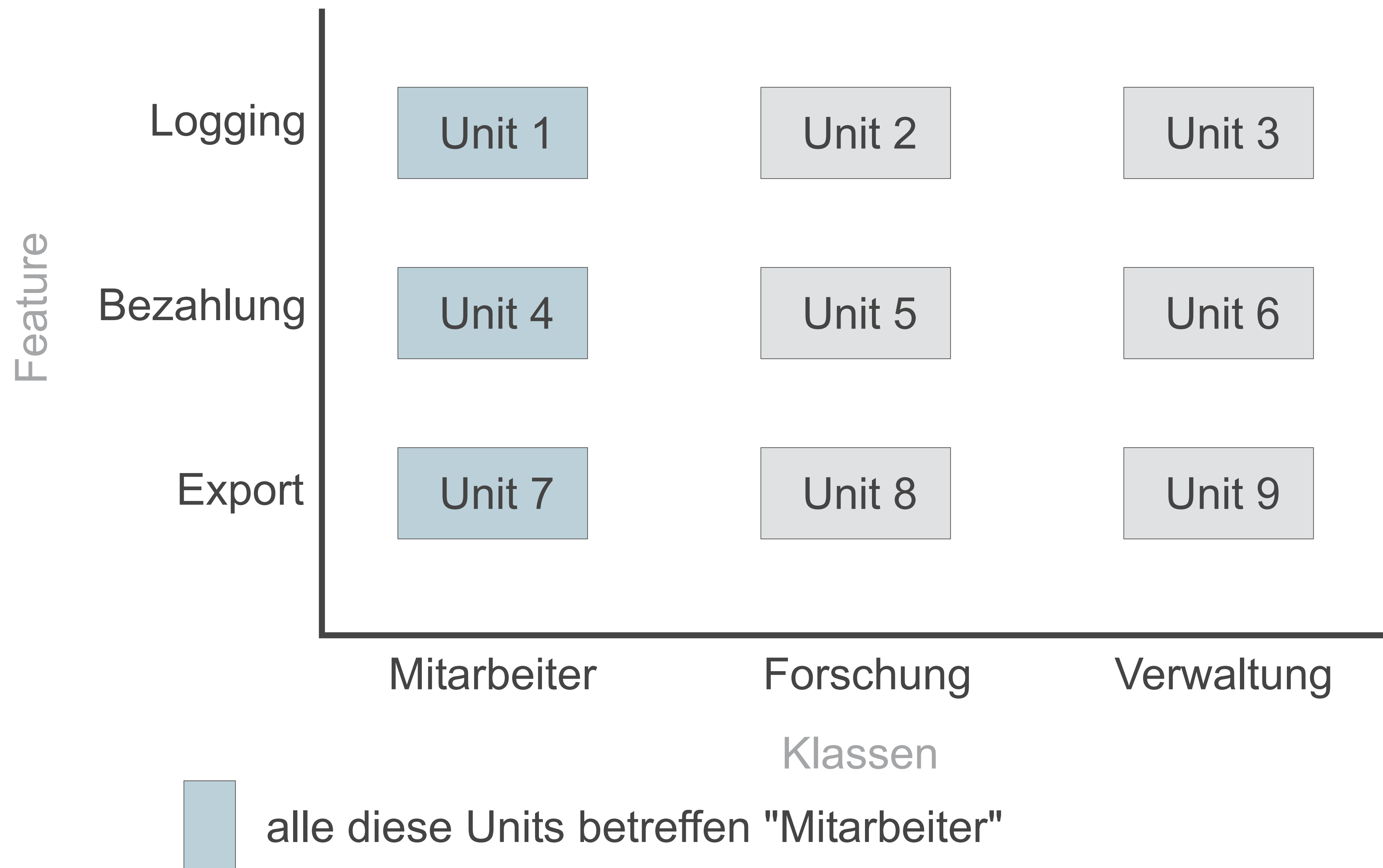
Dimensions

- Die *Units* eines *Hyperspace* sind in einer mehrdimensionalen Matrix organisiert.
- Jede Achse repräsentiert eine *Dimension of Concerns.*
- Jeder Punkt auf der Achse repräsentiert ein Concern dieser Dimension.
- So werden alle Dimensionen von Interesse definiert, die *Concerns*, die zu jeder Dimension gehören und welche *Concerns* von den einzelnen *Units* betroffen werden.
- Die Koordinaten einer *Unit* zeigen alle Concerns, die von ihr betroffen sind.
- Jede *Unit* betrifft genau ein Concern in jeder Dimension

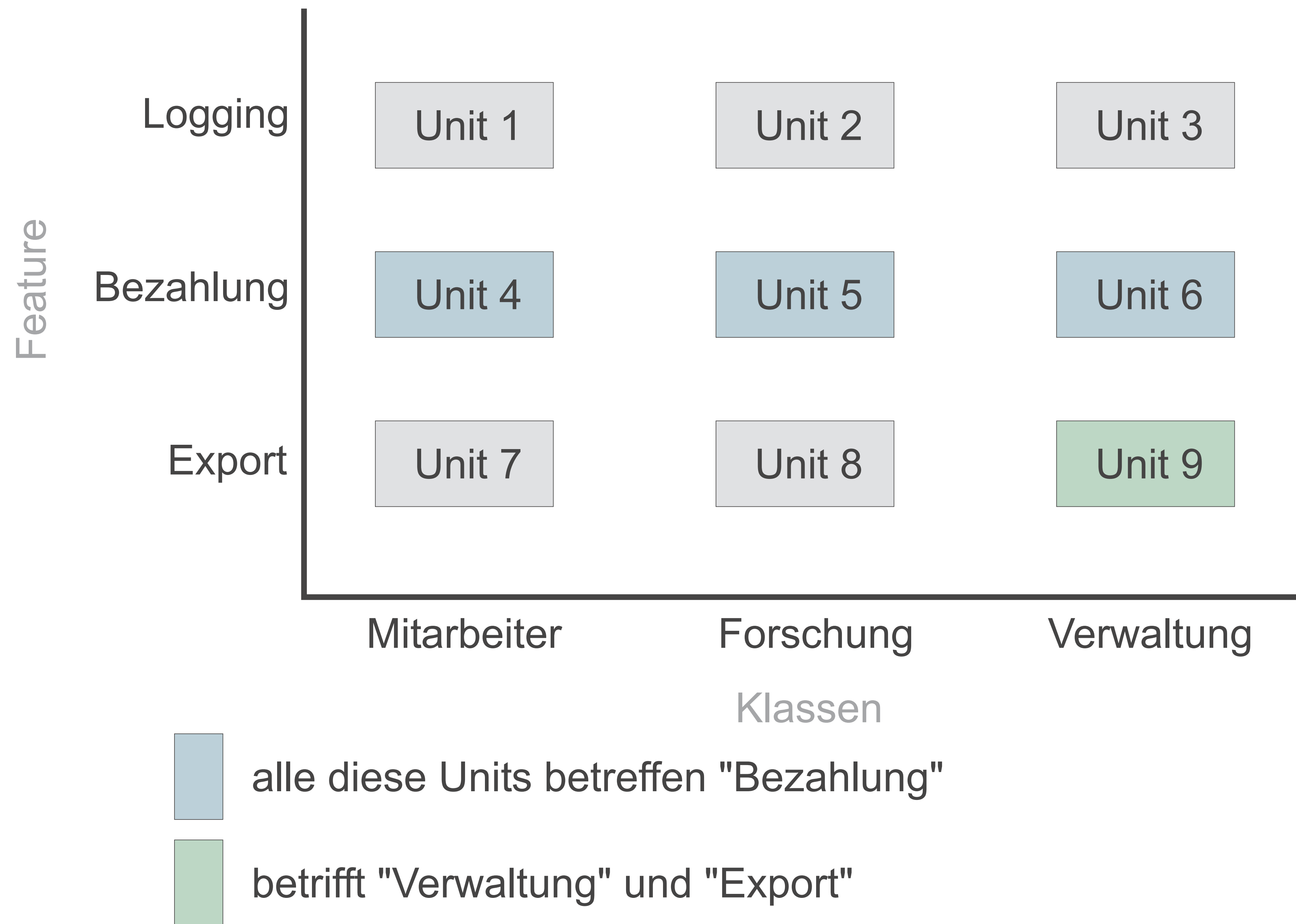
Beispiel: 2 Dimensionen



Beispiel: 2 Dimensionen



Beispiel: 2 Dimensionen



Hyperslices

- Die *Concern Matrix* identifiziert *Concerns* und organisiert *Units* entsprechend der Dimensionen und *Concerns.*
- Allerdings unterstützt sie nicht die Kapselung der *Concerns*.
- Dies geschieht mit sog. *Hyperslices.*
- Hyperslices sind Mengen von *Concerns*, die deklarativ komplett (*declaratively complete*) sind,
- d.h. sie deklarieren alles, was sie benutzen, z.B.
  - Methoden
  - Funktionen
- Die Funktionen müssen allerdings nicht implementiert sein.

Hyperslices

- Somit sind Hyperslices *self-contained*, d.h. sie sind nicht auf andere *Hyperslices* angewiesen.
- Allerdings benötigen sie eventuell irgendwelche andere Hyperslices, um eine bestimmte Funktion auszuführen.
- Beispiel:
  - Hyperslice A, B und C.
  - A ruft Methode print() auf, muß also print() deklarieren, implementiert diese aber nicht.
  - B implementiert Methode print(), C ebenso
  - nun kann man B gegen C austauschen
  - man kann B und C auch ganz weglassen (nur dann keine Funktionalität)

Hypermodules

- Hyperslices sind *Building Blocks*.
- Sie können in größere Blöcke integriert und schließlich zu einem kompletten System zusammgefügt werden.
- Dieses Zusammenfügen geschieht durch Hypermodules
- Hypermodules bestehen aus einer Menge von Hyperslices und aus einer Menge von *Integration Relationships*, die spezifizieren, wie genau die Hyperslices zusammengefügt werden sollen.
- Hypermodules sind selbst wieder Hyperslices (deklarativ komplett)
- Hypermodules können somit ineinandergeschachtelt werden.

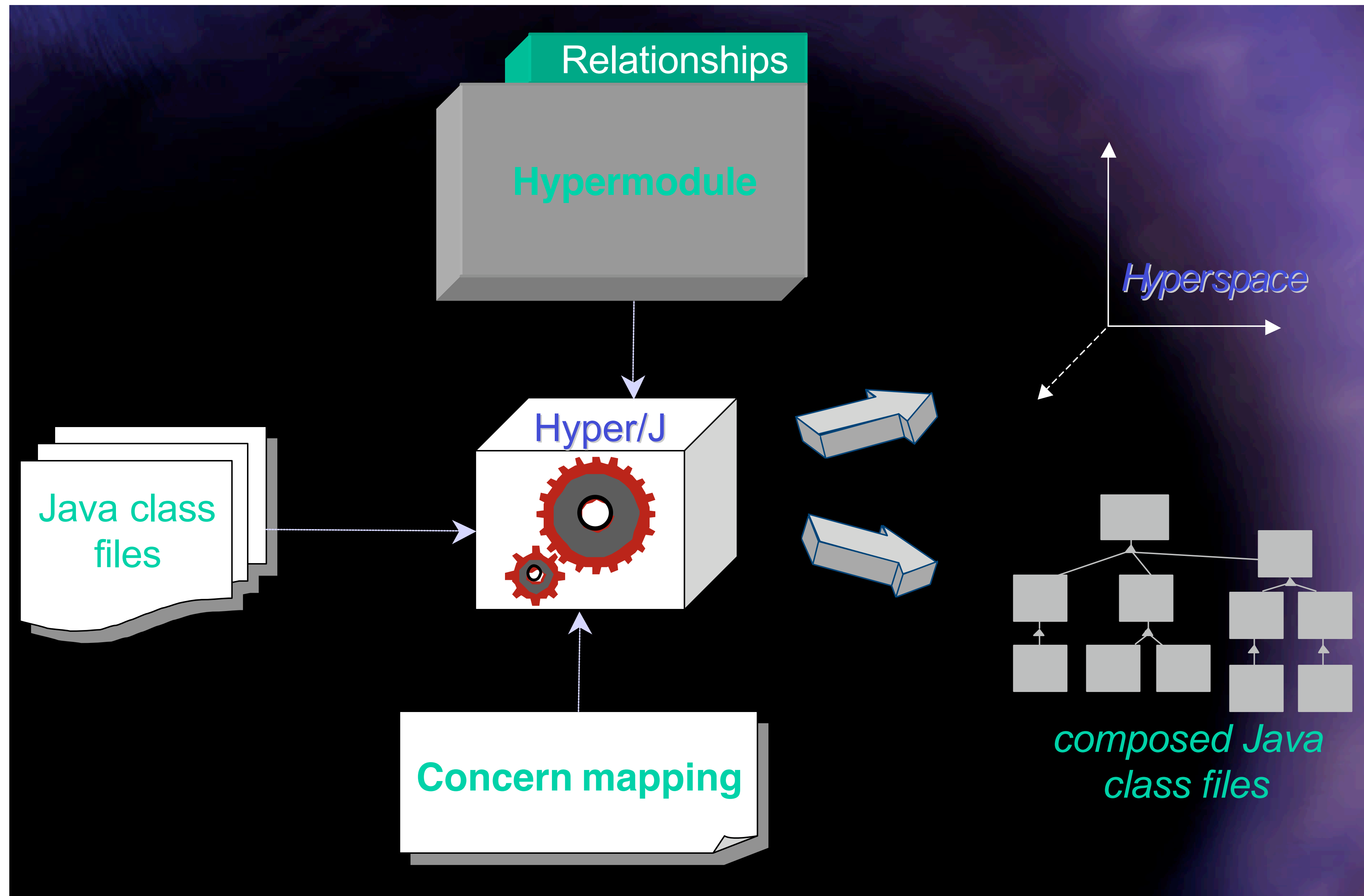
Composition

- Das Zusammenfügen kann auf drei verschiedene Arten geschehen:
- MergeByName
  - Units in verschiedenen Hyperslices haben den gleichen Namen und korrespondieren.
  - Sie werden gemeinsam in eine neue Unit integriert
- NonCorrespondingMerge
  - Units haben zufällig den gleichen Namen, stehen sonst aber in keiner Beziehung zueinander
  - Sie werden nicht verbunden.

Composition

- overrideByName
  - die letzte Unit überschreibt die vorherigen
  - bezieht sich nur auf Methoden, die überschrieben werden

Arbeitsweise von Hyper/J



Quelle: P.Tarr, H.Ossher, S. M. Sutton Jr., Hyper/J:Multidimensional Separation of Concerns, IBM T.J. Watson Research Center

Soweit die Theorie ...

... doch nun zur Praxis

Installation von Hyper/J

- Hyper/J kommt als JAR-Datei
- lauffähig auf allen Java Systemen
  - getestet mit Sun´s JDK von 1.1.5 bis 1.2.1
- Installation:
  - Entpacken des Archivs
  - Setzen des Classpath
    - z.B. export CLASSPATH=$CLASSPATH:/usr/local/hyperj/bin/hyperj.jar
    - Bei einigen Versionen müssen noch andere Pfade gesetzt werden. (siehe Anleitung)
- fertig!
- Hyper/J verändert nichts an der Java-Installation

Hyper/J Projekt Dateien

- Hyperspace Specification File
  - ähnlich einer Projektbeschreibung
  - Auflistung aller Java Klassendateien, auf die Hyper/J angewendet werden soll
- Concern Mapping Files
  - beschreiben, wie verschiedene Teile der Java Klassendateien verschiedene Concerns betreffen
- Hypermodules oder Relationships Specification File
  - beschreibt ein Hypermodule oder mehrere Hypermodules, die erstellt werden sollen (Integration of Concerns)
- All diese Spezifikationen können auch in einer Datei angegeben werden.

Hyperspace Specification File

- Dateiendung .hs
- Beispiel:

```
hyperspace myHJProject
    class myHJPackage.someClass, myHJPackage.otherClass;
    composable class klgPackage.* except klgPackage.kio;
    composable file /home/klaus/myHJProject/sf/*;
    uncomposable class java.lang.*, java.io.*;
```

Concern Mapping File

- Dateiendung .cm
- Beispiel:

```
package java.lang : Feature.Library;
package myProject.util : Feature.Utilities;

class myProject.util.myClass : Optimizations.UtilityOpts;

operation myPackage.aClass.aMethod : Feature.myConcern;
operation log : Feature.Logging;

field somePackage.aClass.instanceVar : Feature.someConcern;
field fooVar : Feature.foo;
```

Hypermodules

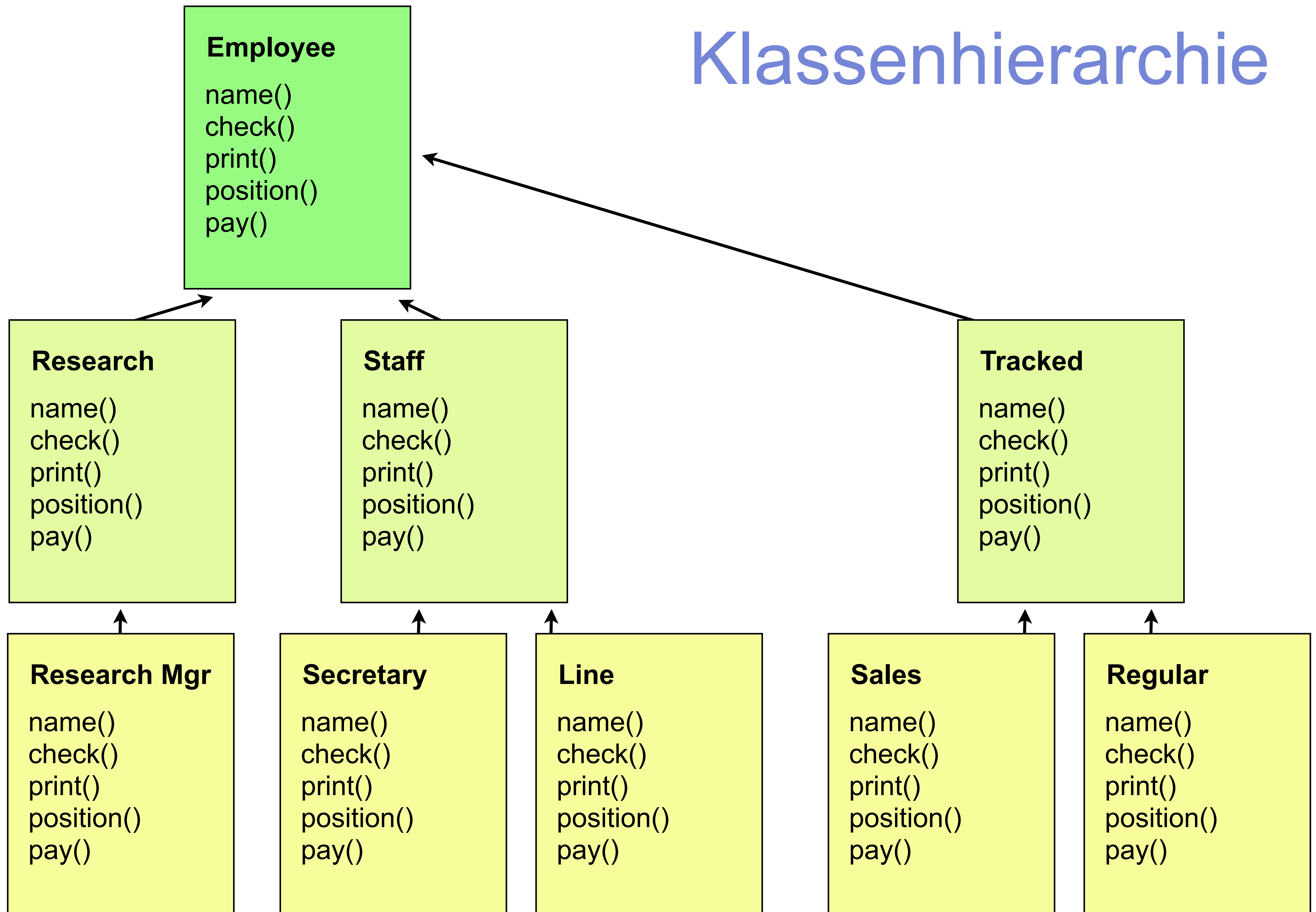
- Dateiendung .hm
- Beispiel:

```
hypermodule myHyperModule
     hyperslices:
     Feature.Display;
     Feature.Logging;
     relationships:
     mergeByName;
     equate operation Feature.Display.show,
     Feature.Logging.log_show;
end hypermodule;
```

Beispiel: Personaldatenbank

- Als Beispiel dient hier eine Personal-Datenbank
- Relativ große Klassenhierarchie
- Es soll implementiert werden:
  - Lohn-Kalkulation: ein neuer Aspekt
  - Export: ein neues Feature

Klassenhierarchie



Beispielcode 1

```
public class Research extends Employee {
     ...
     public boolean check() {
        return ( super.check() &&
        (theSalary >= theFloor) &&
        (theSalary <= theCeiling) );
     }

     public float pay() { return theSalary; }
}
```

Beispielcode 2

```
public abstract class Tracked extends Employee {
    ...
    public boolean check() {
        return ( super.check() &&
        pay() >= minPay() );
    }
}
```

```
public class Regular extends Tracked {
    ...
    public boolean check() {...}
    public float pay() {
        return theWage * theHours;
    }
}
```

Beispielcode 3

```
public class Research extends Employee {
     ...
     public boolean check() {
        return ( super.check() &&
        (theSalary >= theFloor) &&
        (theSalary <= theCeiling) );
     }

     public float pay() {
        System.out.println(”Paying Research “ + name());
        return theSalary; }
}
```

Beispielcode 4

```
public abstract class Tracked extends Employee {
    ...
    public boolean check() {
        return ( super.check() &&
        pay() >= minPay() );
    }
}
```

```
public class Regular extends Tracked {
    ...
    public boolean check() {...}
    public float pay() {
        System.out.println(”Paying Reg. “ + name());
        return theWage * theHours;
    }
}
```

Beispielcode 5

```
public class Research extends Employee {
    ...
    public boolean check() {
        return ( super.check() &&
        (theSalary >= theFloor) &&
        (theSalary <= theCeiling) );
    }

    public float pay() {
        System.out.println("Paying Research " + name());
        return theSalary; }

    public void export() {
        // Stream out XML<Research>
    }
}
```

Beispielcode 6

```
public class Regular extends Tracked {
     ...
     public boolean check() {...}
     public float pay() {
        System.out.println(”Paying Reg. “ + name());
        return theWage * theHours;
     }

     public void export() {

        // Stream out XML:<Regular>

     }
}
```

Beispielcode 7

package Personnel

```
public class Research extends Employee {
     ...
     public float pay() {
        return theSalary;
     }
}
```

package Personnel.Export

```
public abstract class Employee {
     public abstract void export(PrintStream s);
}
```

```
public abstract class Research extends Employee {
     public void export(PrintStream s) {
        s.println(”Research Name = “ + name() );
     }
     public abstract String name();
}
```

Und nun die Live-Demo !