Eine Pattern-Sprache zur Strukturierung großer Software-Systeme mit Java

Arno.Haase@Haase-Consulting.com
Arno.Haase@acm.org

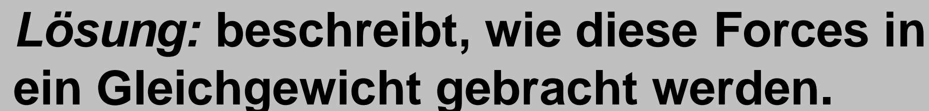
Übersicht

- **Patterns und Pattern-Sprachen**
- Packages als Kapselungs-Einheiten
- Patterns zur Gestaltung der Package-Schnittstelle
- Beziehungen zwischen Packages
- Diskussion

Patterns

- Ein Pattern ist eine wiederverwendbare Lösung für ein Problem in einem Kontext.
  - Nicht nur die Lösung sondern auch eine Beschreibung des Designraums.

Kontext: **definiert die Designsituation, in der das Problem auftritt.**



Pattern-Kataloge

- Ein Pattern-Katalog ist eine Sammlung von Patterns
  - Nicht notwendigerweise miteinander verbunden
- Gruppierung ist möglich nach der
  - Problemdomäne (Finanzwirtschaft, Telekommunikation, ...)
  - Lösungsdomäne (Java, C++, RDBMS, ...)
- Beispiel: „Design Patterns“ (Gamma et al.)

Pattern-Sprachen

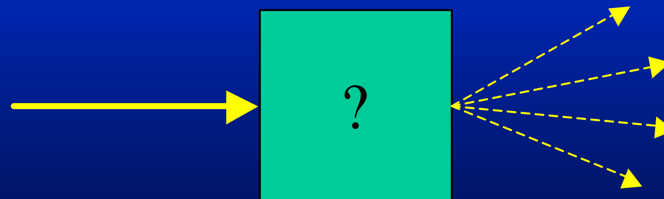
- Pattern-Sprachen beschreiben einen größeren Bereich im Lösungsraum
- Viele kleine Patterns mit ihren Beziehungen
  - Oft führt ein Pattern ganz natürlich zu einem anderen
  - Die Lösung des einen Patterns gehört zum Kontext des nächsten
- Das Ganze ist mehr als die Summe seiner Teile

Übersicht

- Patterns und Pattern-Sprachen
- ➢ **Packages als Kapselungs-Einheiten**
- Patterns zur Gestaltung der Package-Schnittstelle
- Beziehungen zwischen Packages
- Diskussion

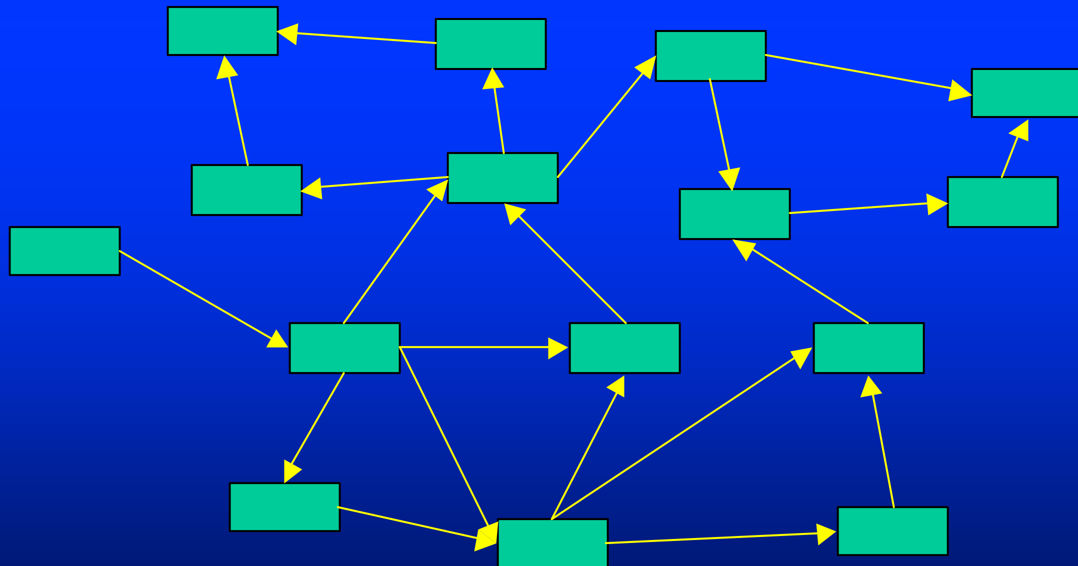
Kapselung

- Kapselung ist die Trennung von Schnittstelle (inkl. Kontrakt) und Implementierung
  - „Was ich nicht weiß, macht mich nicht heiß“: Kapselung vereinfacht Arbeitsteilung
  - Dokumentiert Systemstruktur
  - „Divide et impera“: Gesamtkomplexität ist reduziert
- Klassen: Kapselung im Kleinen



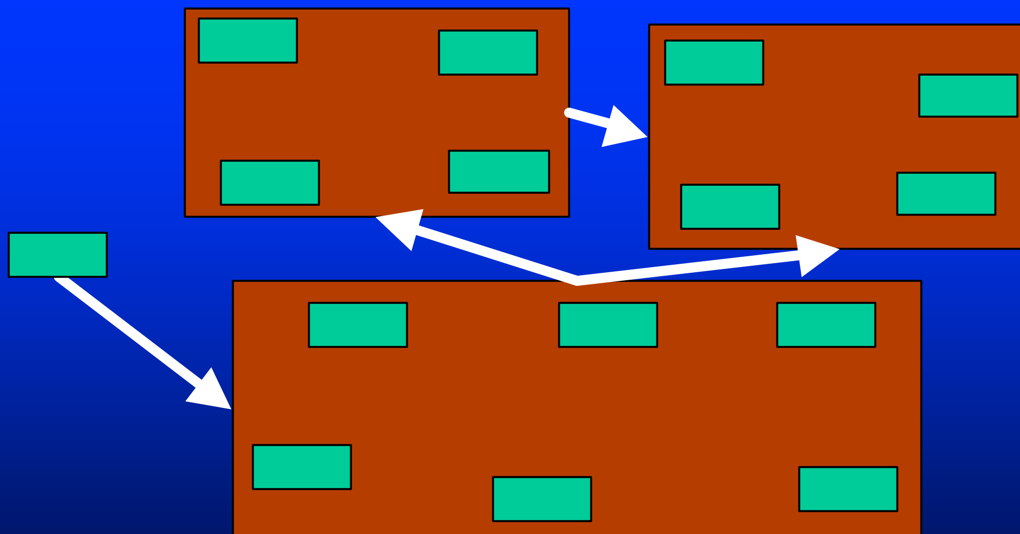
Encapsulating Package

Problem: In einem großen System werden die Beziehungen der Klassen unübersichtlich.



Encapsulating Package (2)

Lösung: Identifiziere zusammenhängende Gruppen von Klassen und kapsele sie als größere Abstraktionen in Packages.



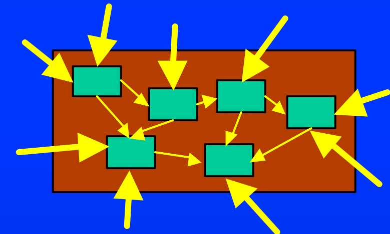
Encapsulating Package (3)

- ☺ Die Anzahl der Teile und ihrer Beziehungen wird kleiner
- ☺ Die Architektur des Systems spiegelt sich im Quelltext
- ☺ Packagenamen sind Dokumentation

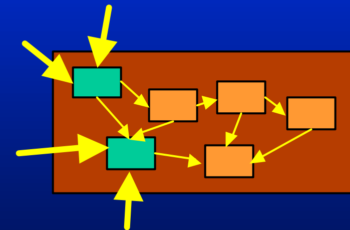
- ☹ Ungeschickt geschnittene Packages können später hinderlich sein
- ☹ Refactoring über Packagegrenzen hinweg ist aufwendiger

Package Local Class

Problem: Wenn potentiell jede Klasse eines Encapsulating Package von außen verwendet wird, erhöht das die Komplexität.



Lösung: Definiere explizit ein öffentliches Interface und reduziere die Sichtbarkeit aller übrigen Klassen auf Default Visibility.



Package Local Method

Problem: Bei manchen Klassen gehört nur ein Teil der Methoden zum öffentlichen Interface eines Encapsulating Package.

Lösung: Setze die übrigen Methoden auf Default Visibility.

Package Local Constructor

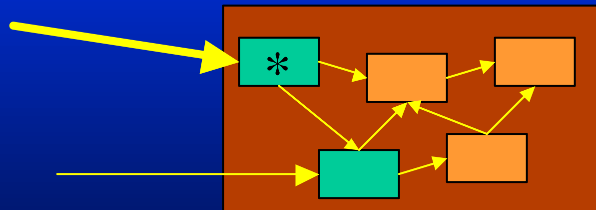
Problem: Wie kann ein Encapsulating Package die Erzeugung von Objekten eines Typs kontrollieren?

Lösung: Gib dem entsprechenden Konstruktor – bzw. allen – Default Visibility.

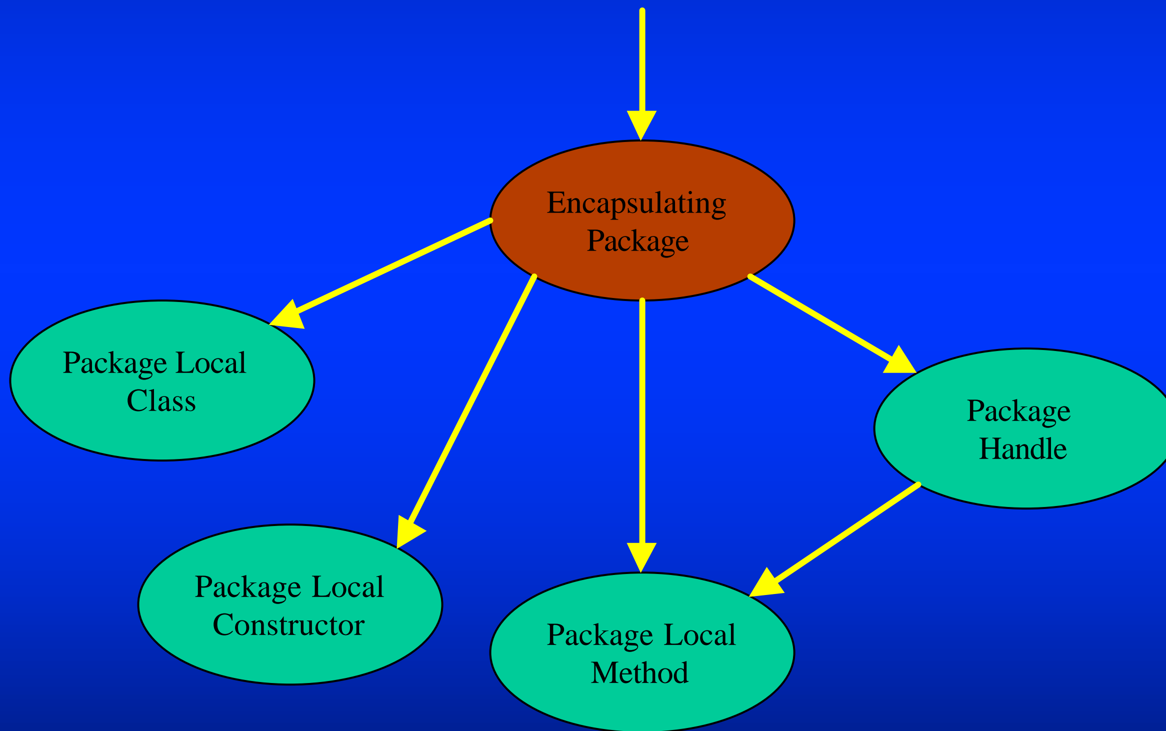
Package Handle

Problem: Ein Encapsulating Package benötigt als Ganzes (nichtstatischen) Zustand.

Lösung: Definiere eine Klasse, die als Handle auf den Zustand dient. Deren Instanzen dienen Clients als Referenz auf Package-Instanzen und typischen Ausgangspunkt der Navigation darin.



Verbindung der Patterns zu einer Sprache



Übersicht

- Patterns und Pattern-Sprachen
- Packages als Kapselungs-Einheiten
- **Patterns zur Gestaltung der Package-Schnittstelle**
- Beziehungen zwischen Packages
- Diskussion

Value Based Programming

- Wertsemantik: Zustand spielt zentrale Rolle, Identität ist unwichtig
- In Schnittstellen von Packages herrscht oft Bedarf an spezifischen Typen mit Wertsemantik, z.B. Geschlecht und Körpertemperatur in einem medizinischen System.
- Java unterstützt nicht direkt die Definition eigener Typen mit Wertsemantik; primitive Typen und „Konstanten-Interfaces“ sind schlechter Ersatz

Value Class

Problem: Wie lassen sich neue Typen mit Wertsemantik definieren?

Lösung: Definiere Klassen, in denen die wertbezogenen Methoden von Object überschrieben sind. Mache die Klasse entweder unveränderlich oder implementiere Mechanismen zum einfachen kopieren.

→ Einstieg in andere Patternsprache

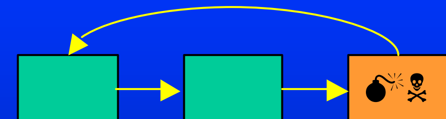
Value Class: Beispiel

```
public class Temperatur {
  private final float _gradCelsius;

  public Temperatur (final float gradCelsius) {
    _gradCelsius = gradCelsius;
  }

  public float getGradCelsius () {return _gradCelsius;}

  public boolean equals (final Object obj) {
    if (! (obj instanceof Temperatur)) return false;
    return (((Temperatur)obj)._gradCelsius == _gradCelsius);
  }

  public int hashCode () {return new Float (_gradCelsius).hashCode();}
}
```

Homogeneous Exception

Problem: Geworfene Exceptions sind Teil der Schnittstelle eines Encapsulating Package

- Intern auftretende Exceptions können oft nicht lokal behandelt werden
- Einfache Propagation offenbart Implementierungsdetails, koppelt Clients an Interna



- Viele verschiedene Exceptions im Interface erschweren das Fangen und die Behandlung

```
try {...}
catch (A a)  {...}
catch (B b)  {...}
catch (C c)  {...}
...
```

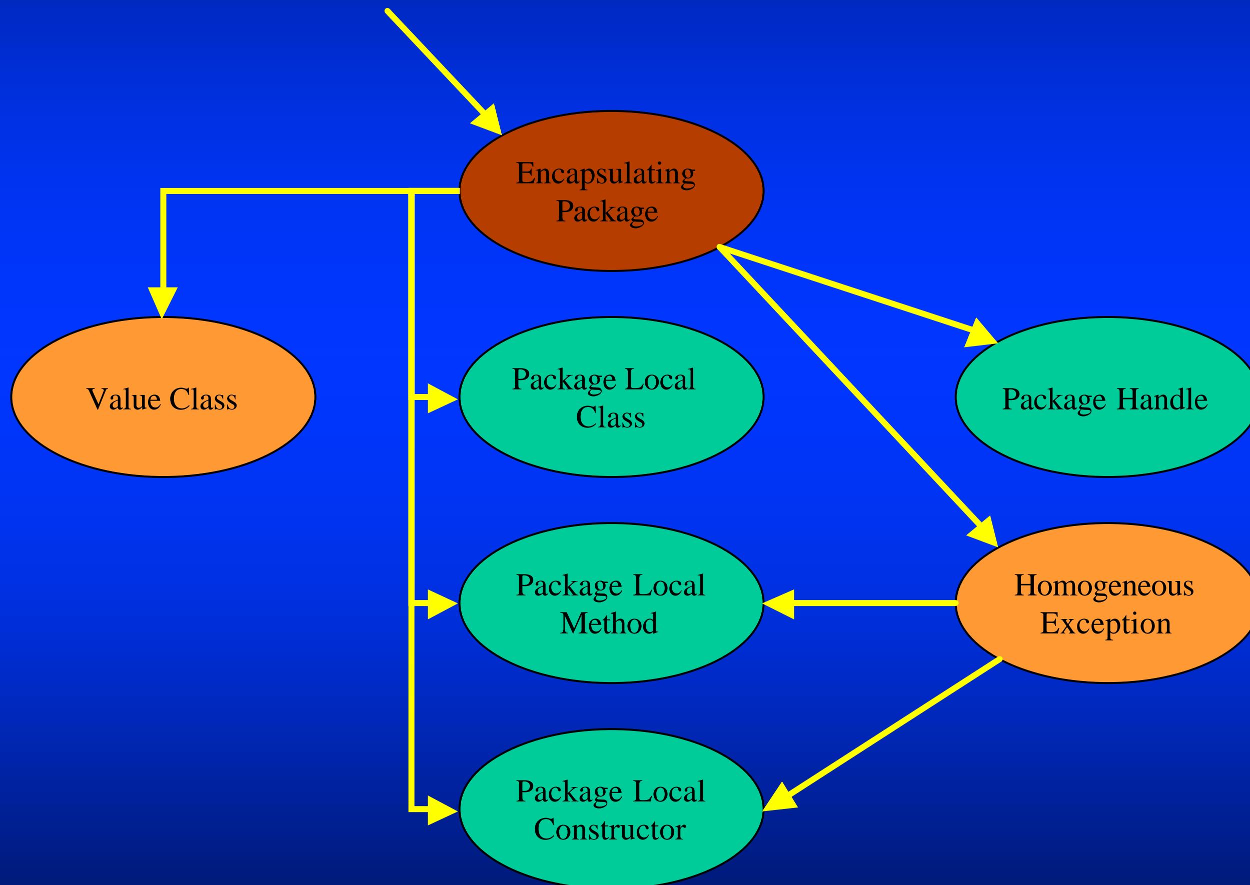
Homogeneous Exception (2)

Lösung: Definiere eine einzige Exception, in die alle auftretenden Exceptions konvertiert werden. Methoden des öffentlichen Interface deklarieren typischerweise nur diese Exception.

- Eine Differenzierung kann über Unterklassen oder über Zustand der Exception erfolgen, je nach dem erwarteten Muster der Behandlung
- Information über die ursprüngliche Exception (insbesondere der Stacktrace) wird durch Logging oder Wrapping bewahrt

→ Einstieg in andere Pattern-Sprache

Zwischenstand Pattern-Sprache

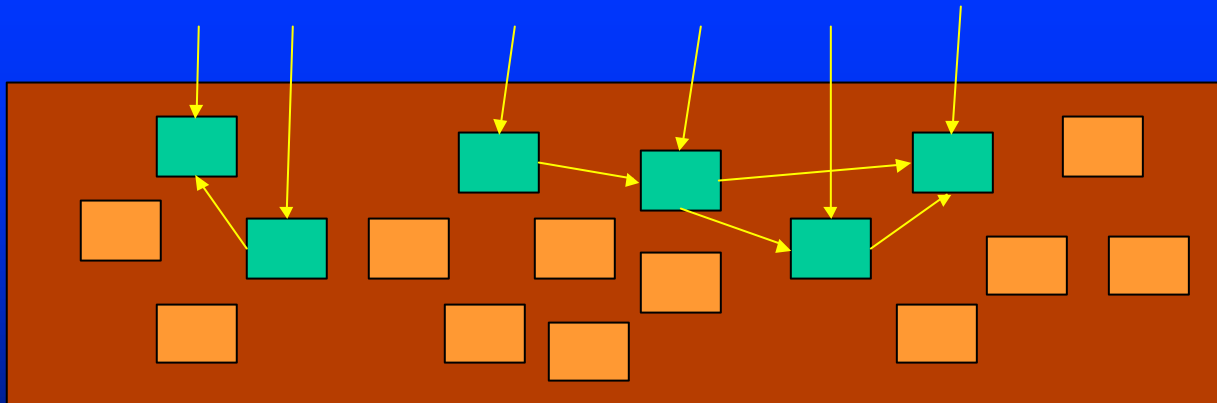


Übersicht

- Patterns und Pattern-Sprachen
- Packages als Kapselungs-Einheiten
- Patterns zur Gestaltung der Package-Schnittstelle
- **Beziehungen zwischen Packages**
- Diskussion

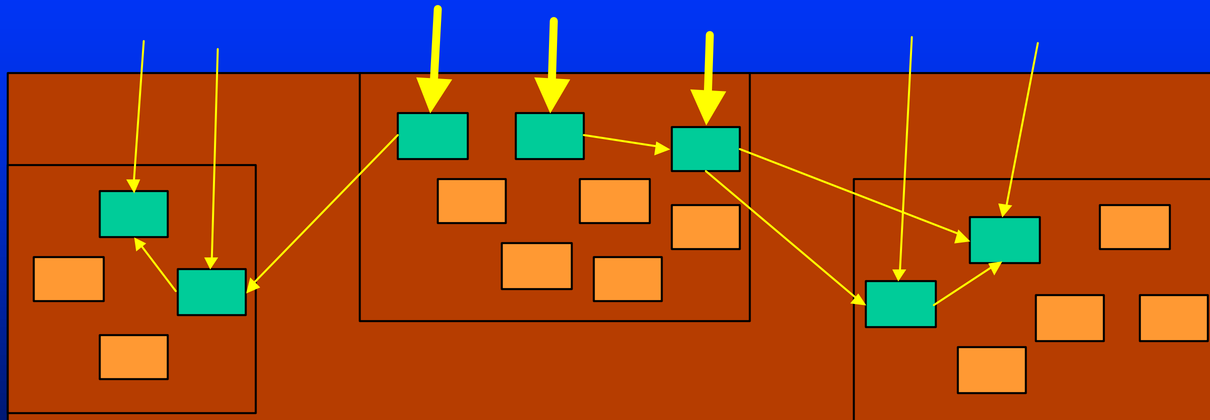
Facade Package

Problem: Ein Encapsulating Package kann so sehr wachsen, daß sein öffentliches Interface unübersichtlich wird.



Facade Package (2)

Lösung: Lagere seltener von Clients benötigte Funktionalität in andere Packages aus. Das ursprüngliche Package fungiert dann als Facade für die anderen.



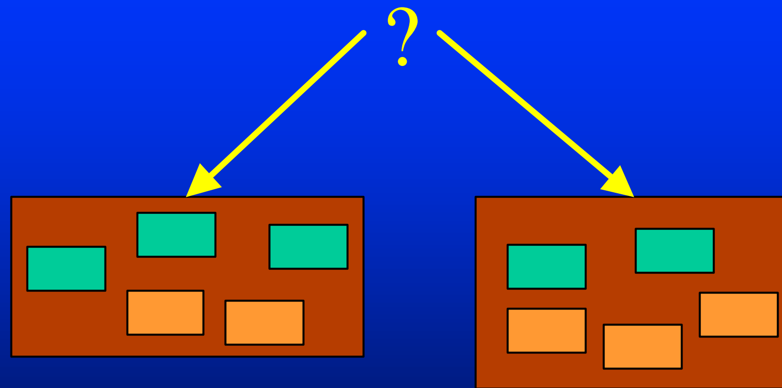
Facade Package (3)

- ☺ Es gibt ein schmaleres Interface, das meist genügt
- ☺ Wenn ein Client das urprüngliche, breitere Interface braucht, steht es ihm zur Verfügung
- ☺ Die internen Klassen sind besser organisiert, ihre Beziehungen werden dadurch dokumentiert

- ☹ Manchmal ist nicht offensichtlich, in welchem Package eine spezielle Klasse ist: Man muß suchen
- ☹ Es gibt mehr Packages, die Gesamtkomplexität steigt leicht

Interface Package

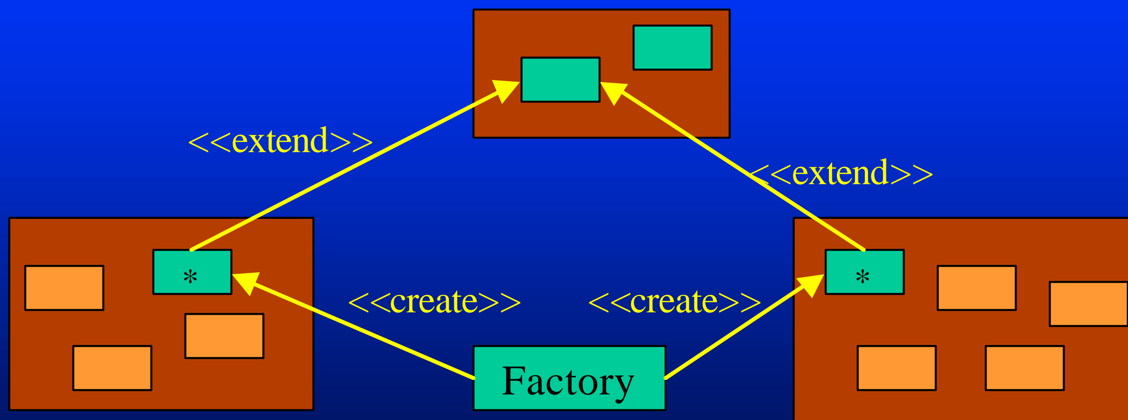
Problem: Wie definiert man austauschbare Implementierungen von Encapsulating Packages, die Clients polymorph verwenden können?



Interface Package (2)

Lösung: Definiere ein Interface-Package, das die gemeinsame Schnittstelle enthält. Diese Schnittstelle enthält alle spezifischen Parametertypen und Exceptions.

Eine Factory instanziiert die gewünschte Implementierung und liefert einen Package Handle auf sie.

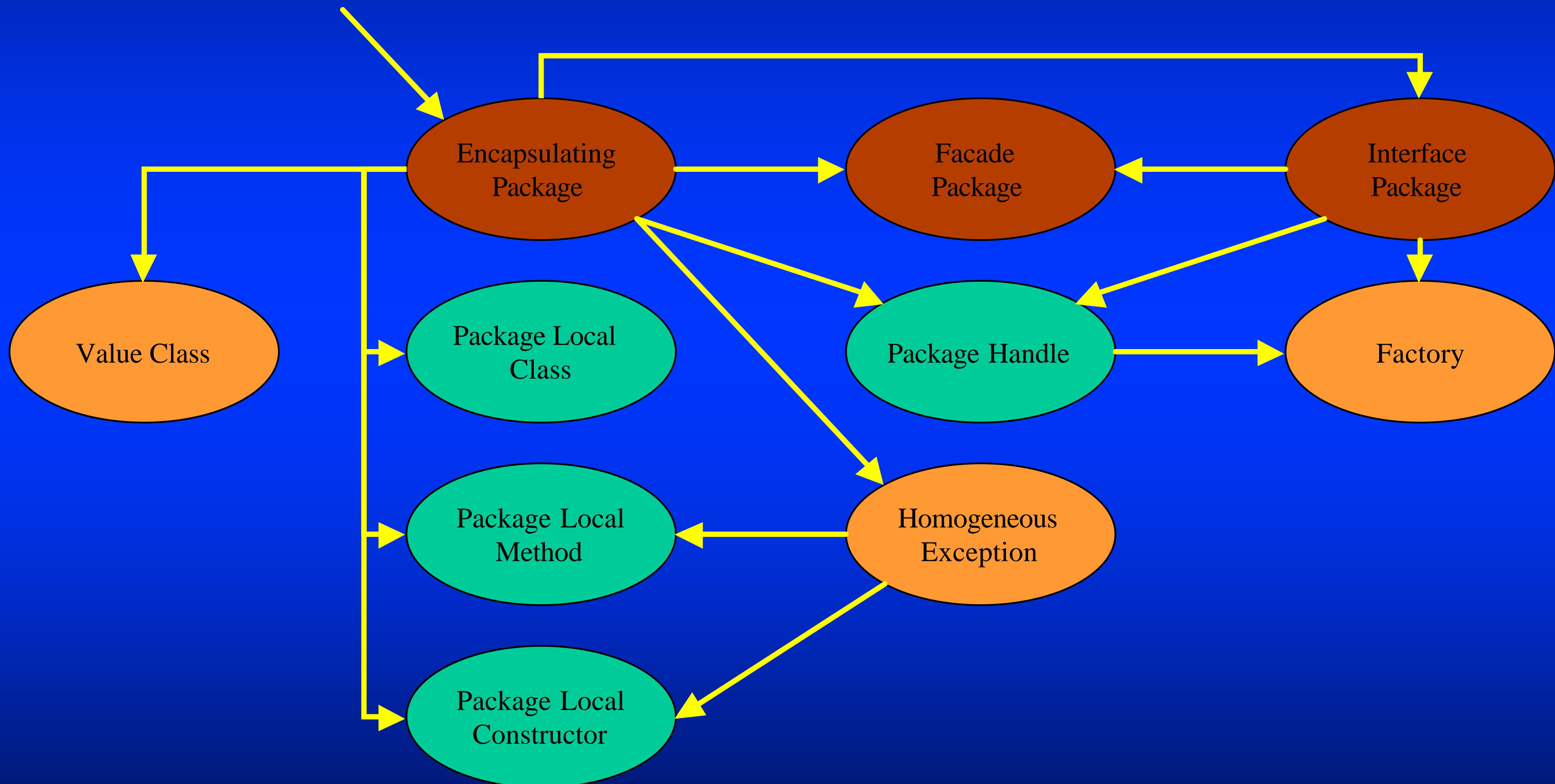


Interface Package (3)

- ☺ Es gibt ein sehr wohldefiniertes Interface
- ☺ sehr große Flexibilität
- ☺ Austauschbarkeit der Implementierungen (sogar zur Laufzeit)
- ☺ Einzelne Implementierungen können selektiv die Kapselung durchbrechen (z.B. Oracle-JDBC)

- ☹ Deutlich höhere Komplexität, sowohl für Implementierung als auch für Client
- ☹ Interface ist sehr aufwendig zu ändern

Die Pattern-Sprache



Fazit Kapselung mit Packages

- Java-Packages erlauben die Kapselung von Systemteilen mit grober Granularität
  - explizite Trennung von Schnittstelle und Implementierung
  - Instanzen und nichtstatischer Zustand
  - Polymorphie
  - Keine Implementierungsvererbung...

Fazit Pattern-Sprache

- ☺ Viele kleine Patterns mit Beziehungen
  - Das Diagramm gibt einen schnellen Überblick
  - Die Sprache liefert verschiedene Blickwinkel auf das Gesamtproblem
  - Alternativen und Varianten werden diskutiert
  - Bewährte Design-Lösungen werden dokumentiert
  - Navigation vom Einstiegspunkt her ist natürlich
- ☺ Verweise auf andere Pattern-Sprachen
  - Die einzelne Sprache ist nicht überfrachtet
  - Man kann jede Sprache unabhängig von anderen nutzen

Übersicht

- Patterns und Pattern-Sprachen
- Packages als Kapselungs-Einheiten
- Patterns zur Gestaltung der Package-Schnittstelle
- Beziehungen zwischen Packages
- ➢ **Diskussion**

Wert-basierte Programmierung

nach [Henney2000]