

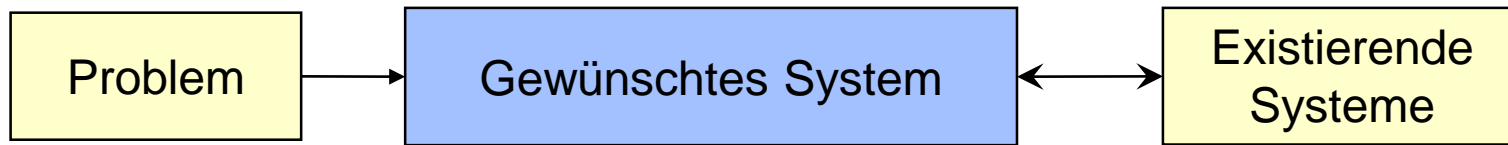
Kapitel 7a) Systementwurf -- Dienste, Subsysteme, Komponenten

Stand: 21.11.2017

**Keine inhaltlichen
Änderungen seither**

Systementwurf

- **Ziel:** Überbrücken der Lücke zwischen gewünschtem und existierendem System auf handhabbare Weise



- **Idee:** Anwendung des “Divide and Conquer”-Prinzips
 - ◆ Modellierung des neuen Systems als Menge von Subsystemen
- **Folgeproblem:** „Crosscutting concerns“ – Übergeordnete Belange die viele Subsysteme betreffen (z.B. Persistenz, Nebenläufigkeit, ...)
 - ◆ Erst wenn diese geklärt sind, kann man die Subsysteme unabhängig voneinander bearbeiten
- **Weg:** Zielbestimmung → Dekomposition → Klärung der übergeordneten Belangen
 - ◆ Danach erst Detailentwurf der Subsysteme

Systementwurf

Systementwurf

1. Entwurfsziele

Definition
Abwägungen

2. System

Dekomposition

Ebenen / Partitionen
Kohärenz / Kopplung

3. Hardware / Software

Zuordnung

Kaufen vs. Selbermachen
Netzwerktopologie
Allokation

4. Persistente Datenverwaltung

Dateien
Datenbanken
Datenstrukturen

5. Nebenläufigkeit

Identifizierung von
Thread

6. Globale Ressourcenverwaltung

Zugriffskontrolle
Sicherheit

8. Grenzfälle

Initialisierung
Fehler
Ende

7. Programm- steuerung

Monolithisch
Ereignisbasiert



Nutzung der Ergebnisse der Anforderungsanalyse für den Systementwurf

- **Nichtfunktionale Anforderungen** →
 - ◆ Aktivität 1: Definition der Entwurfsziele
- **Statisches Analyse-Modell (Objektmodell mit Stereotypen)** →
 - ◆ Aktivität 2: Systemdekomposition (Auswahl von Subsystemen nach funktionalen Anforderungen, Kohärenz und Kopplung)
 - ◆ Aktivität 3: Hardware/Software Zuordnung
 - ◆ Aktivität 4: Persistentes Datenmanagement
- **Dynamisches Analyse-Modell** →
 - ◆ Aktivität 5: Nebenläufigkeit
 - ◆ Aktivität 6: Globale Ressourcenverwaltung
 - ◆ Aktivität 7: Programmsteuerung
 - ◆ Aktivität 8: Grenzfälle

Kapitel-Überblick

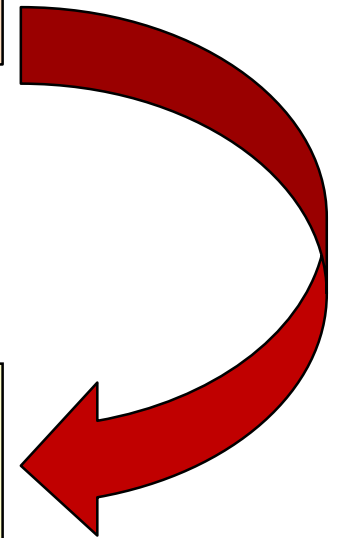
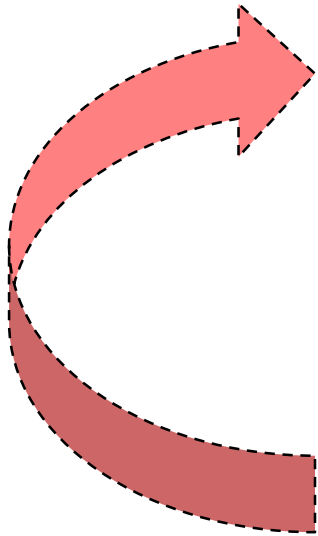
- Ziele und Dekomposition

- ◆ 1. Entwurfsziele
- ◆ 2. Dekomposition in Subsysteme

**System-
Architektur**

- Zielgerichteter Entwurf

- ◆ 3. Hardware/Software Zuordnung
- ◆ 4. Management persistenter Daten
- ◆ 5. Nebenläufigkeit
- ◆ 6. Globale Ressourcenverwaltung
- ◆ 7. Programmsteuerung
- ◆ 8. Grenzfälle

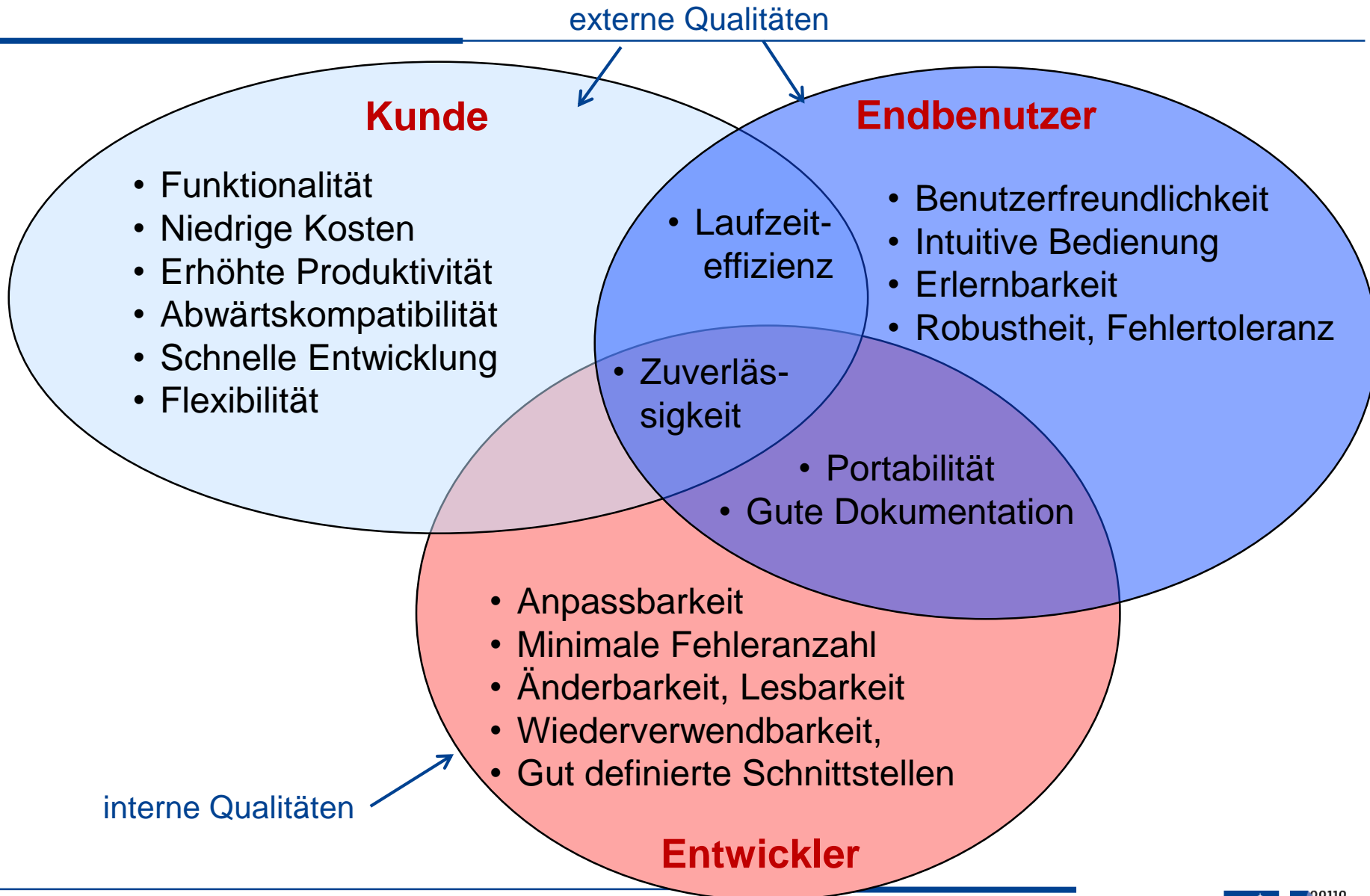


7.1 Ziele

(→ Brügge & Dutoit, Kap. 6)

Entwurfsziele
Dienstidentifikation
Subsystemaufteilung

1. Entwurfsziele



Interessenskonflikte → Abwägungen

- Funktionalität vs. Benutzbarkeit
 - ◆ Je überladener, um so schwerer zu erlernen
- Funktionalität vs. schnelle Entwicklung
 - ◆ Viel Funktionalität zu implementieren braucht Zeit
- Kosten vs. Robustheit
 - ◆ Sparen an Qualitätssicherung
- Kosten vs. Wiederverwendbarkeit
 - ◆ Quick and dirty
- Effizienz vs. Portabilität
 - ◆ Effizienz durch Speziallösung für bestimmtes Betriebssystem, DBMS, ...
- Abwärtskompatibilität vs. Lesbarkeit
 - ◆ Viele Sonderfälle für Altversionen erschweren die Lesbarkeit

Bedeutung nichtfunktionaler Anforderungen für den Systementwurf

- **Dilemma** ▶ Zu viele Alternativen
 - ◆ Die gleiche Funktionalität ist auf verschiedenste Arten realisierbar
- **Nutzen von NFA** ▶ Auswahlkriterien
 - ◆ Nichtfunktionale Anforderungen dienen als Auswahlkriterien
 - ◆ Sie fokussieren die Entwurfsaktivitäten auf die relevanten Alternativen
- **Beispiele** (NF Anforderung → Lösungsmöglichkeiten)
 - ◆ „Hoher Durchsatz“ → Parallelität, optimistische Vorgehensweise, ...
 - ◆ „Zuverlässigkeit“ → Einfache GUIs, Redundanz, ...

7.2. Subsystem-Dekomposition

(→ Brügge & Dutoit, Kap. 6)

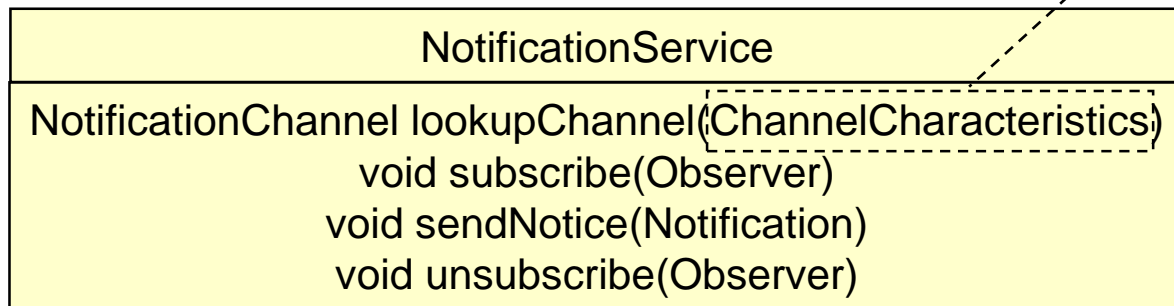
Dienstidentifikation
Subsystemaufteilung
Kopplung und Kohärenz

Subsystem-Dekomposition

- Erster Schritt: **Subsystem-Dekomposition**
 - ◆ Welche Dienste werden von dem Subsystemen zur Verfügung gestellt (Subsystem-Interface)?
 - ◆ →1. Gruppiere Operationen zu Diensten
 - ◆ →2. Gruppiere Typen die einen Dienst realisieren zu Subsystemen
- Zweiter Schritt: **Architektur (Subsystem-Anordnung)**
 - ◆ Wie kann die Menge von Subsystemen strukturiert werden?
 - ◆ Wie interagieren sie?
 - ⇒ Nutzt ein Subsystem einseitig den Dienst eines anderen?
 - ⇒ Welche der Subsysteme nutzen gegenseitig die Dienste der anderen?
 - ◆ → 3. Software Architekturen

Dienst

- **Dienst:** Menge von Operationen mit gemeinsamem Zweck
 - ◆ Beispiel: Benachrichtigungsdienst
 - ⇒ lookupChannel(), subscribe(), sendNotice(), unsubscribe()
 - ◆ Dienste werden während des Systementwurfs identifiziert und spezifiziert
- **Dienstspezifikation:** Vollständig typisierte Menge von Operationen
 - ◆ In UML und Java würde das einem 'Interface' entsprechen
 - ◆ Beispiel: Spezifikation des obigem Dienstes



Entscheidung: Suche nach Channel der bestimmte Fähigkeiten hat soll möglich sein.

Alternative: Suche anhand von Namen

- ◆ Verwendete Schnittstellen (Observer, ...) müssen natürlich auch spezifiziert werden

Subsystem-Schnittstelle

- Besteht aus einem oder mehreren zusammenhängenden Diensten
 - ◆ **Vollständig typisiert** → Parameter und Ergebnistypen
 - ◆ **Zusammengehörig** → Dienen gemeinsam einem bestimmten Zweck bzw. sinngemäß verwandten Funktionen
 - ⇒ Beispiel: Druckdienst (Druckerinstallation, -suche, -anmeldung, ..., Drucken, Drucken anhalten, ...)
- Spezifiziert Interaktion und Informationsfluss von/zu den Grenzen des Subsystems, aber nicht innerhalb des Subsystems
- Sollte wohldefiniert und schlank sein

Subsystem

- Subsystem (UML: Package)
 - ◆ Stark **kohärente** Menge von Klassen, Assoziationen, Operationen, Events und Nebenbedingungen die einen **Dienst** realisieren
 - ◆ Wenn es gute Gründe gibt kann ein Subsystem mehr als einen Dienst anbieten

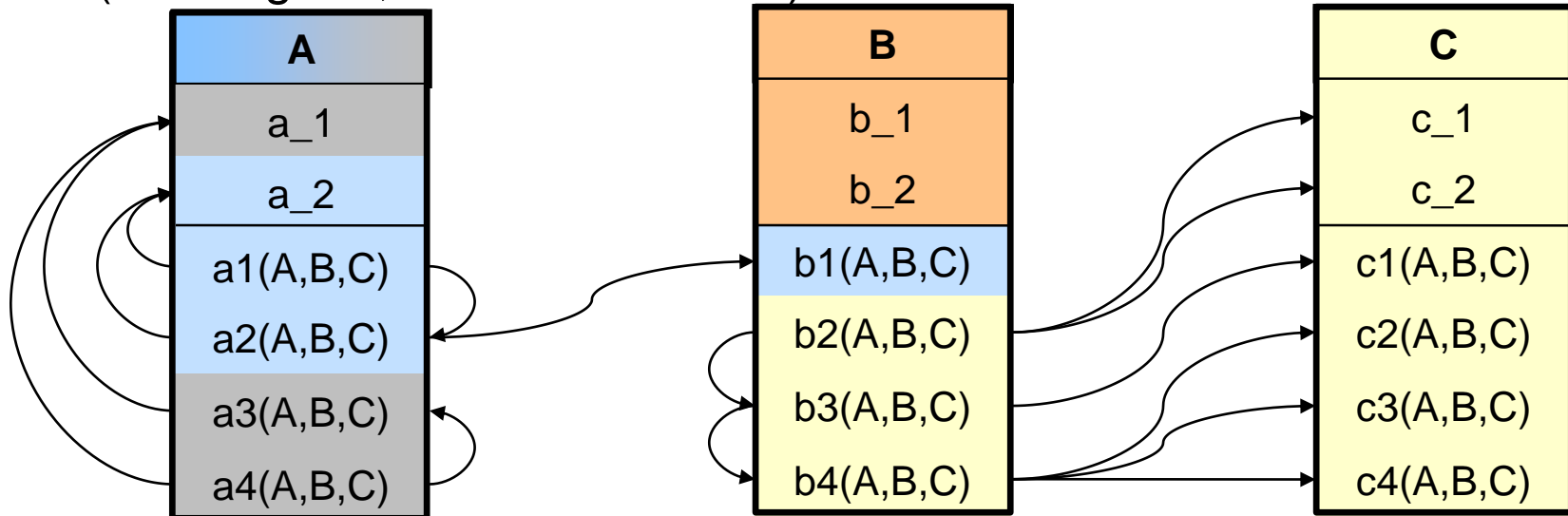
- Frage: Was ist Kohärenz?

Kopplung und Kohärenz

- **Kohärenz** = Maß der Abhängigkeiten innerhalb der Kapselungsgrenzen (hier: innerhalb eines Subsystems)
 - ◆ **Starke Kohärenz**: Die Klassen im Subsystem haben ähnliche Aufgaben und sind untereinander verknüpft (durch Assoziationen)
- **Kopplung** = Maß der Abhängigkeiten zwischen den Kapselungsgrenzen (hier: zwischen den Subsystemen)
 - ◆ **Starker Kopplung**: Modifikation eines Subsystems hat gravierende Auswirkungen auf die anderen (Wechsel des Modells, breite Neukompilierung, usw.)
- Ziel: Wartbarkeit
 - ◆ Die meisten Abhängigkeiten sollten innerhalb einzelner Subsysteme bestehen, nicht über die Subsystemgrenzen hinweg.
- Kriterien
 - ◆ Subsysteme sollten **maximale Kohärenz** und **minimale Kopplung** haben

Beispiel: Kopplung und Kohärenz

- Gegeben folgende drei Klassen. Die Pfeile zeigen Abhängigkeiten (Feldzugriffe, Methodenaufrufe):



Klasse mit 2 unabhängigen Kohäsionseinheiten
→ aufsplitten in 2 Klassen!

Kaum Kohäsion.
→ b_1, b_2 ungenutzt
→ b1 gehört nach A!
→ b2 - b4 gehören nach C!

Völlig unkohäsive Klasse.
B kümmert sich mehr um C-Elemente als C selbst!

7.3 Implementierung von Subsystemen

1. Einheitlicher Einstiegspunkt → „Facade“ Pattern
2. Einziger Einstiegspunkt → „Singleton“ Pattern

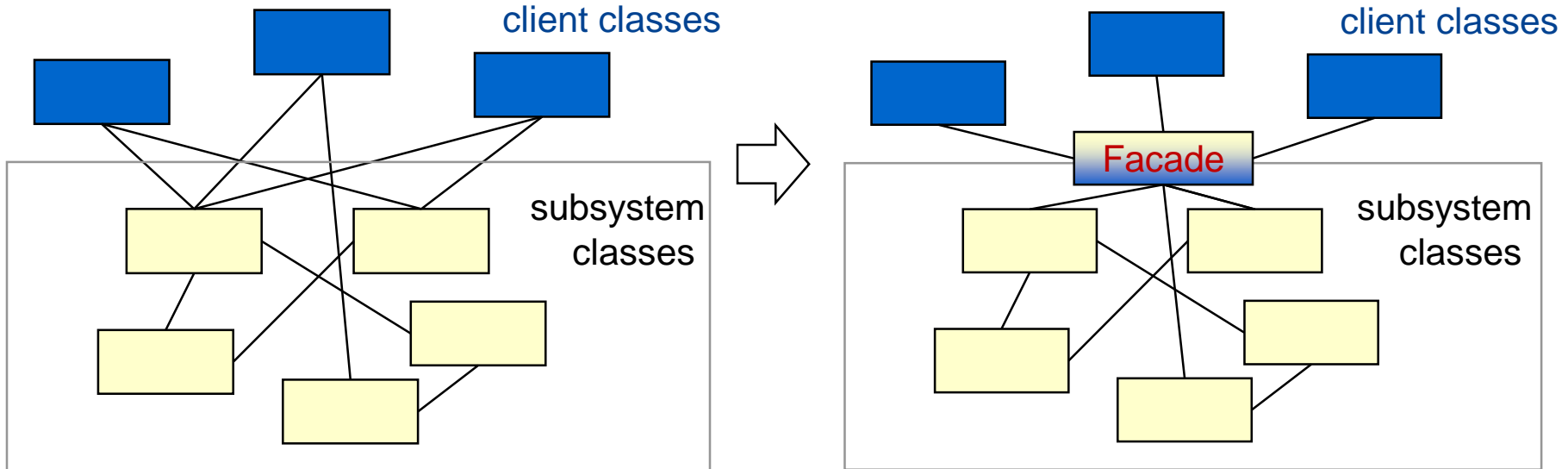
7.3.1 Das Facade Pattern

Facades als Einstiegspunkte in Subsysteme

Subsystem-Implementierung mit „Façade Pattern“

- Absicht

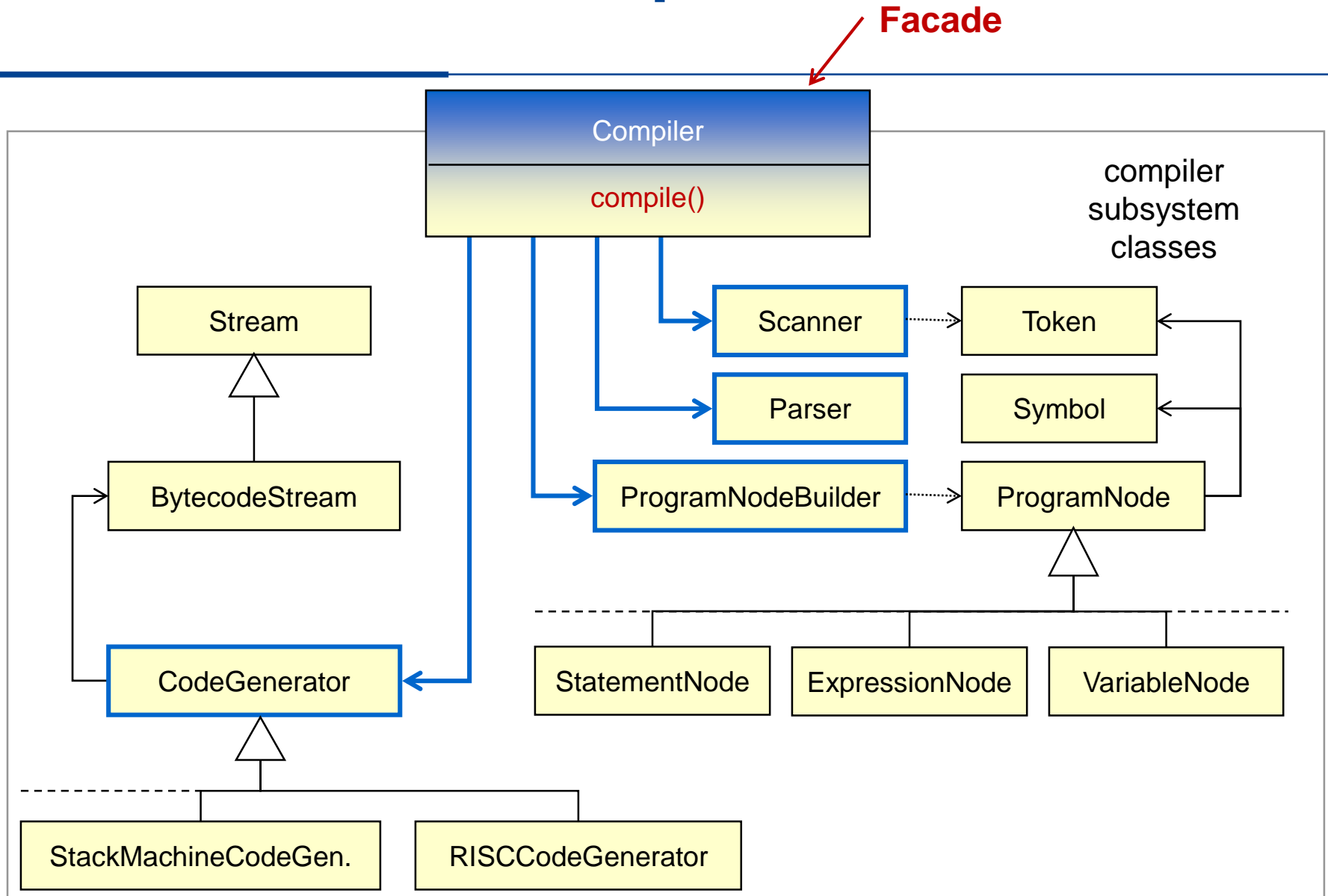
- ◆ Abhängigkeiten der Clients von der Struktur eines Subsystems reduzieren



- Idee: Façade = “Dienst”-Objekt

- ◆ Funktionen einer Menge von Klassen eines Subsystems zu einem Dienst zusammenfassen
- ◆ Objekt, das den Dienst eines Subsystems nach außen darstellt
- ◆ Bietet alle Methoden des Dienstes
- ◆ Vorteil: Clients müssen nichts über die Interna des Subsystems wissen

Facade Pattern: Beispiel



Facade Pattern: Anwendbarkeit

- Viele Abhängigkeiten zwischen Klassen
 - ◆ Reduzieren durch Facade-Objekte
- Einfaches Interface zu einem komplexen Subsystem
 - ◆ Einfache Dinge einfach realisierbar (aus Client-Sicht)
 - ◆ Anspruchsvolle Clients dürfen auch "hinter die Facade schauen"
 - ⇒ zB für seltene, komplexe Anpassungen des Standardverhaltens
- Hierarchische Strukturierung eines System
 - ◆ Eine Facade als Einstiegspunkt in jede Ebene

Facade Pattern: Konfigurierbarkeit

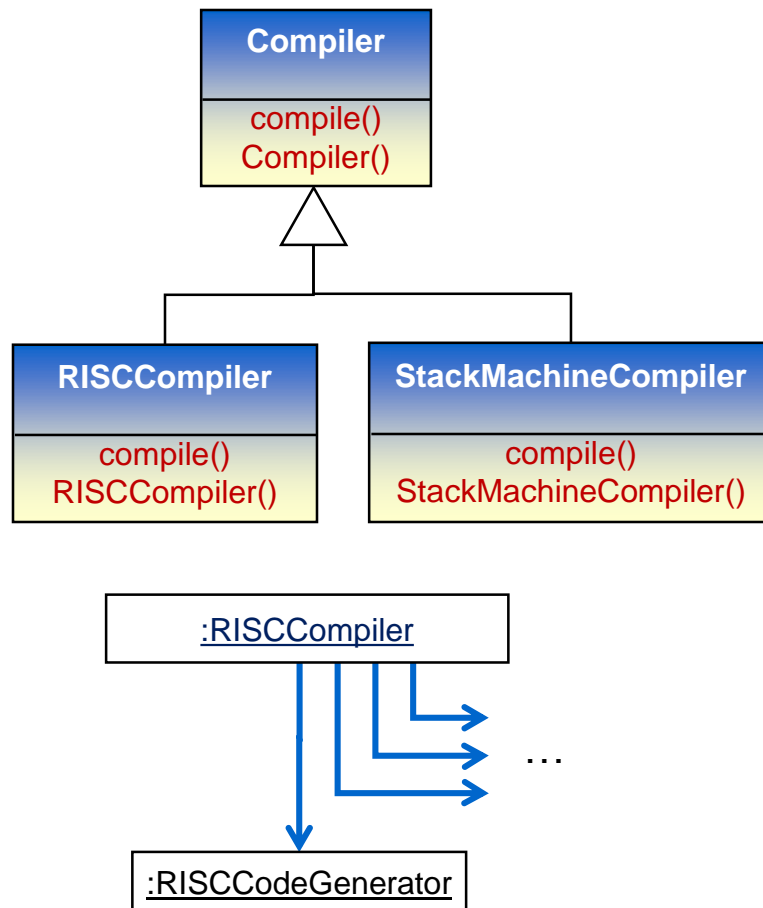
Beispiel: Verwendung des „StackMachineCodeGenerator“ versus „RISCCodeGenerator“

Realisierungsalternativen

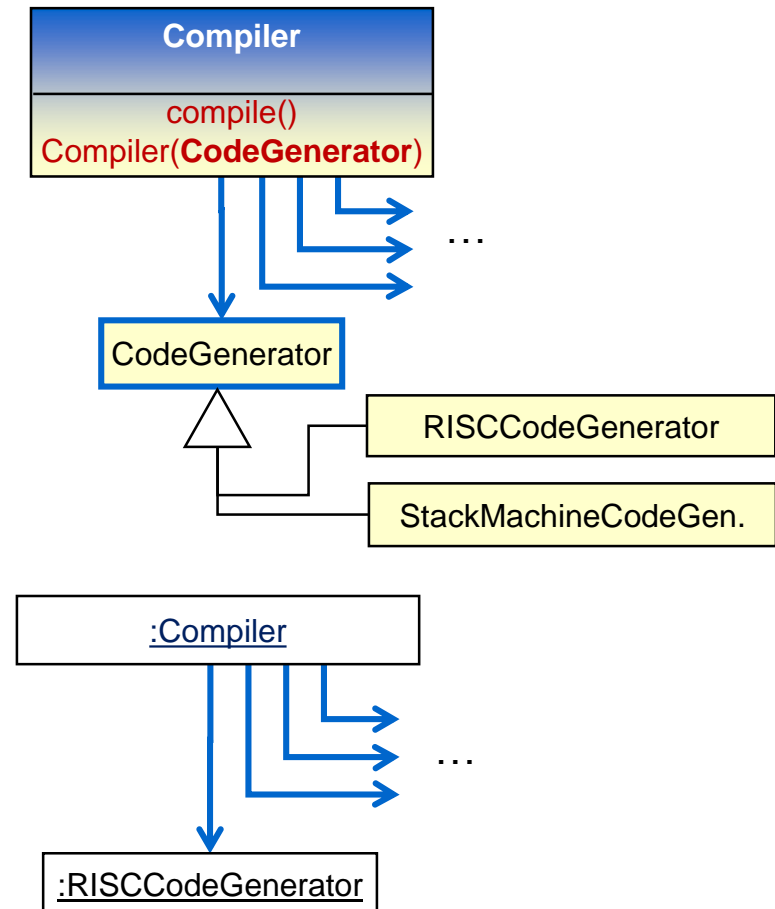
- Eigene Facade-Subklasse pro Konfiguration
oder
 - Nur eine Facade-Klasse deren Instanzen durch das explizite Setzen verschiedener Subsystem-Objekte konfiguriert werden
- Grafik hierzu siehe nächste Seite

Facade Pattern: Konfigurierbarkeit

Konfiguration = Subklasse



Konfiguration = Parameter

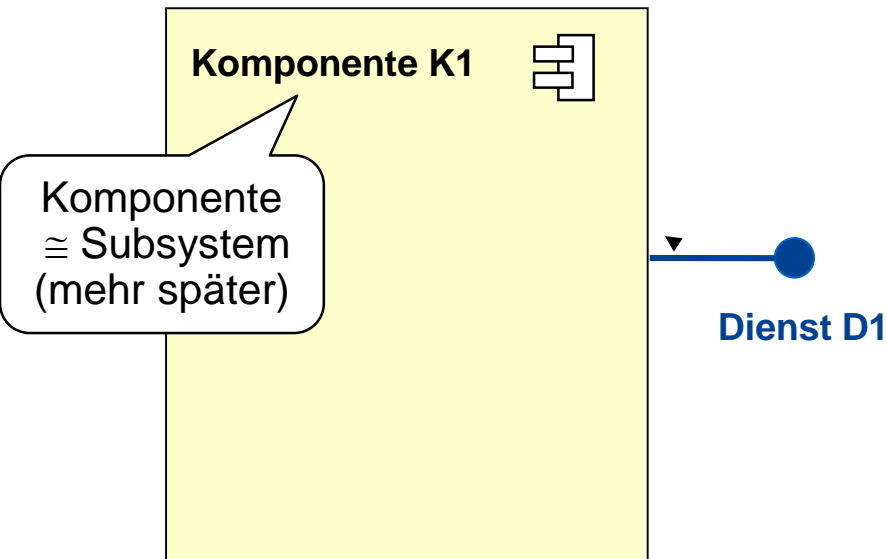


Façade als Realisierung eines Dienstes

Black-Box Sicht

Komponente bietet der Außenwelt einen Dienst

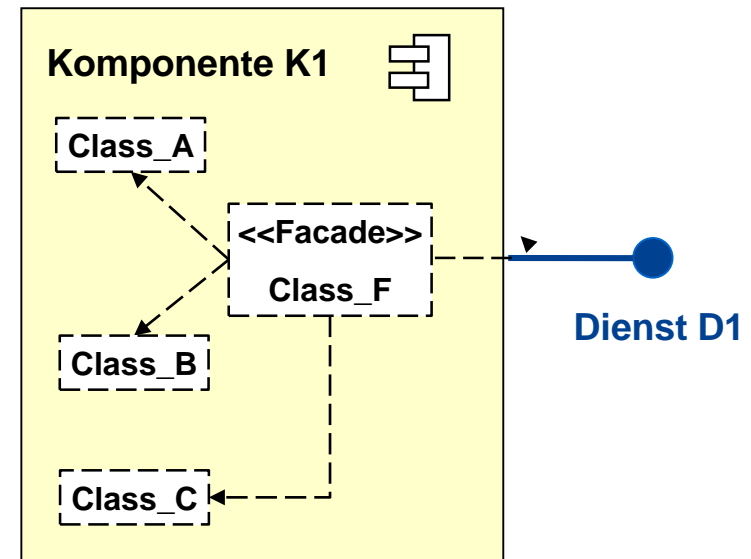
- K1 bietet D1
- Wie das geschieht ist egal



Interne Sicht

Dienst wird intern durch eine Façade implementiert

- Class_F implementiert D1 und agiert als Façade



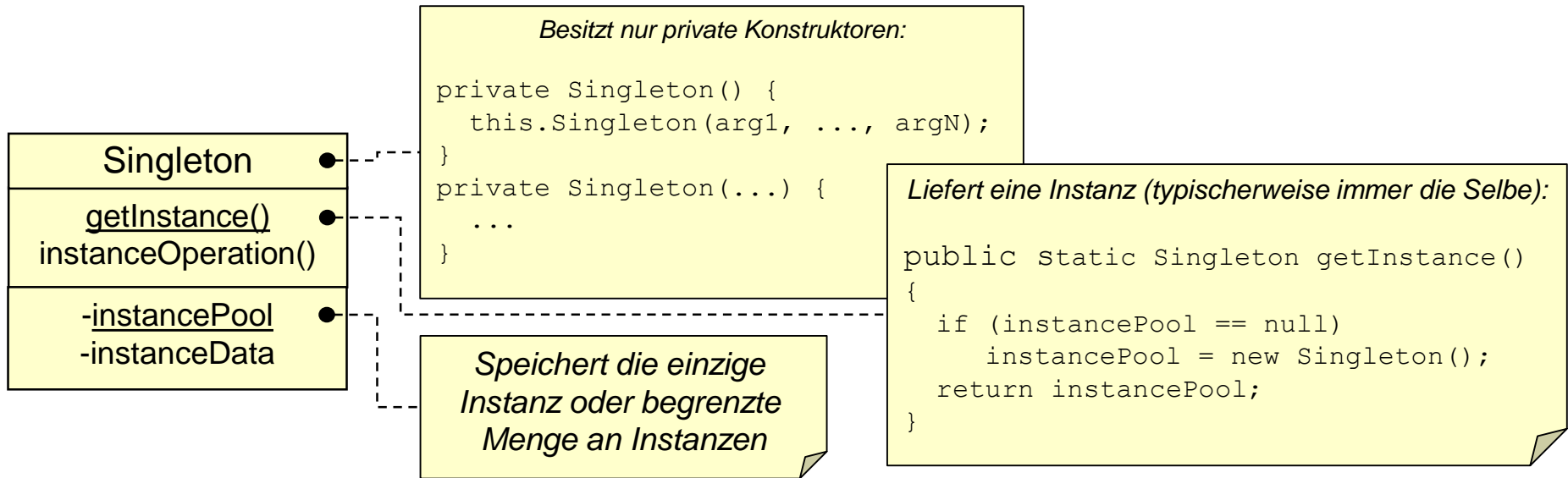
7.3.2 Das Singleton Pattern

Facades sind typischerweise Singletons

Singleton: Motivation

- Beschränkung der Anzahl von Exemplaren zu einer Klasse
- Meist: nur ein einzelnes Exemplar
 - ◆ Motivation: Zentrale Kontrolle → Z.B. Facade, Repository, Abstract Factory
- Aber auch: feste Menge von Exemplaren
 - ◆ Motivation 1: begrenzte Ressourcen (z.B. auf mobilen Geräten)
 - ◆ Motivation 2: Teure Objekterzeugung durch „Object Pool“ vermeiden
→ z.B. 1000 Enterprise Java Beans vorhalten, nach Nutzung zurück in den Pool

Singleton: Struktur + Implementierung



- Nur private Konstruktoren
 - ◆ dadurch wird verhindert, dass Clients beliebig viele Instanzen erzeugen können
 - ◆ in Java muss **explizit ein privater** Konstruktor mit leerer Argumentliste implementiert werden, damit **kein impliziter öffentlicher** Konstruktor vom Compiler erzeugt wird
- `instancePool` als Registry für alle Singleton-Instanzen
 - ◆ lookup-Mechanismus erforderlich um gezielt eine Instanz auszuwählen

7.4 Beispiel: Vom Analysemodell zur Systemdekomposition

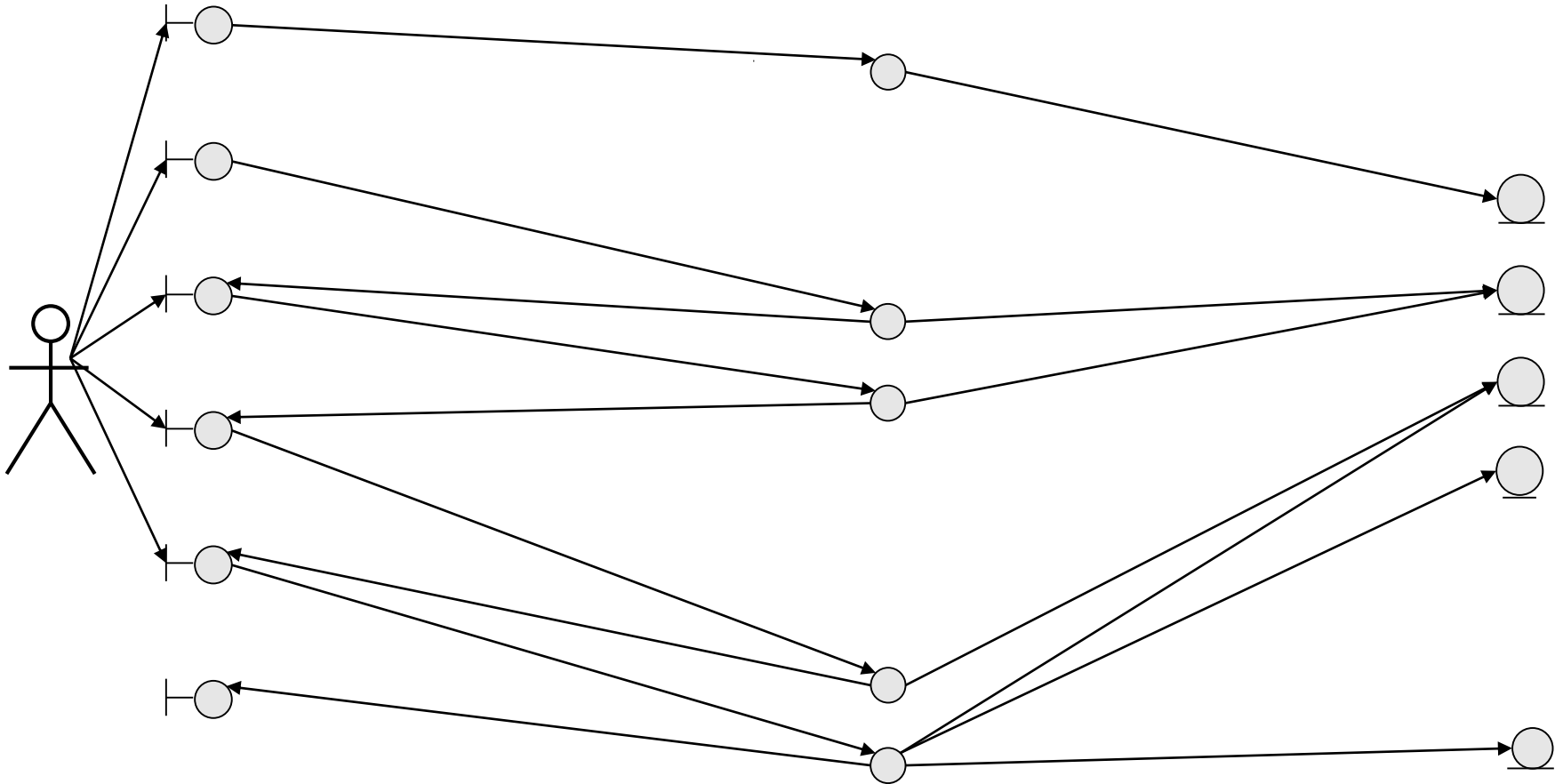
Gruppieren nach ähnlichen Funktionalitäten

Angebotene und genutzte Dienste identifizieren (Schnittstellen)

Subsysteme einführen (Komponenten)

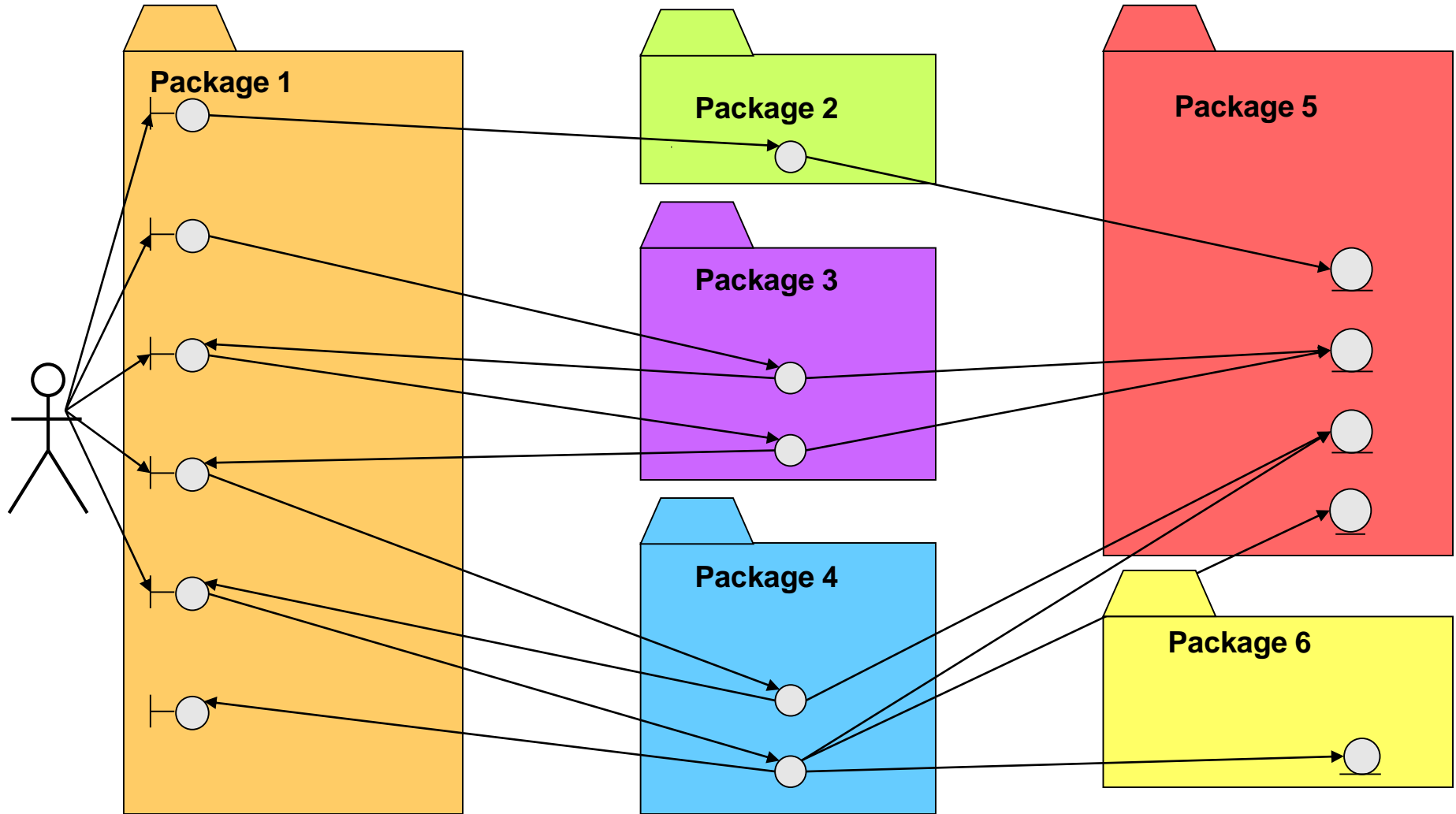
Facades als Einstiegspunkte in die Subsysteme hinzufügen

Ausgangspunkt: Objektmodell der Analyse

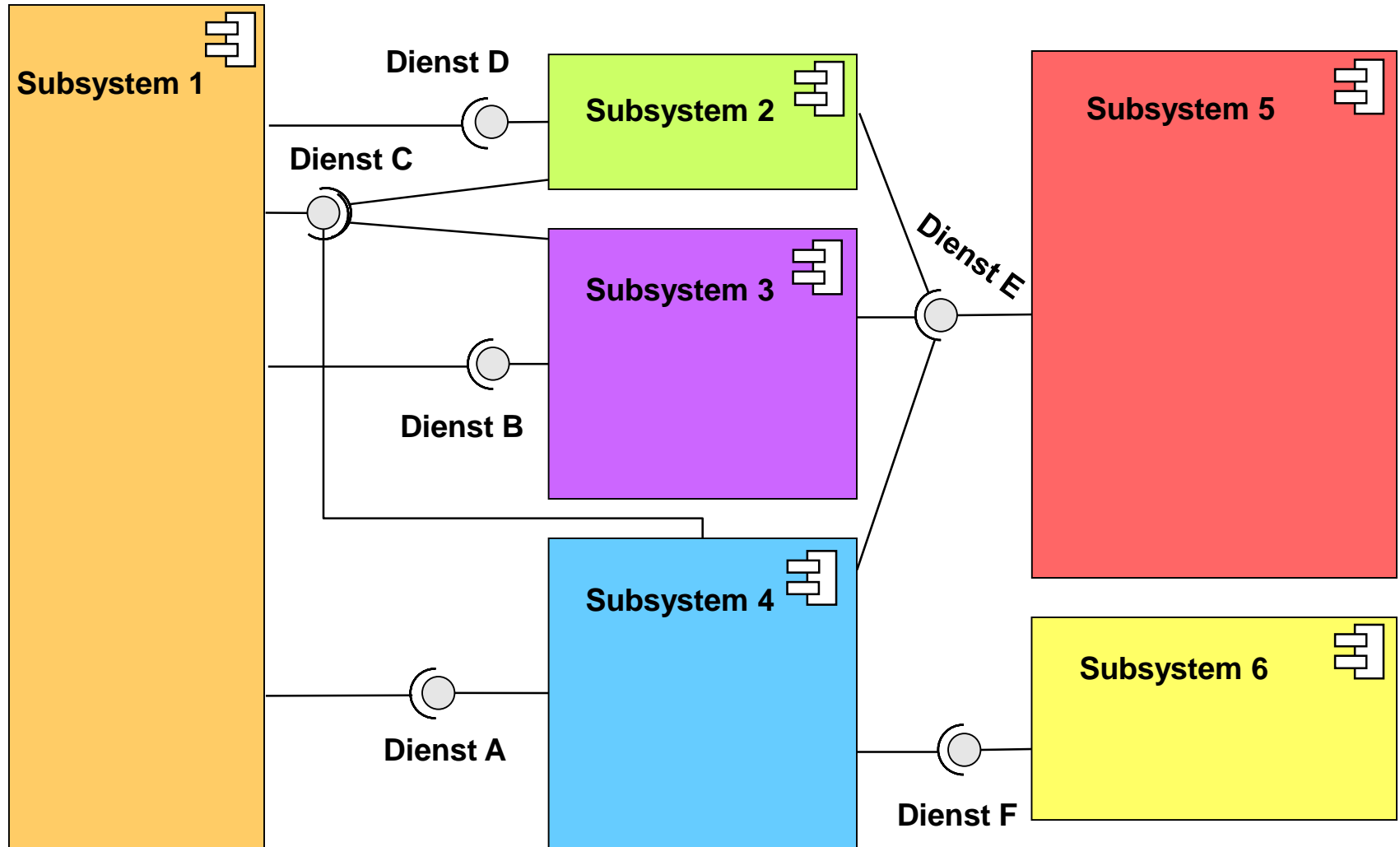
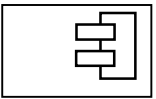


Gruppierung in Packages

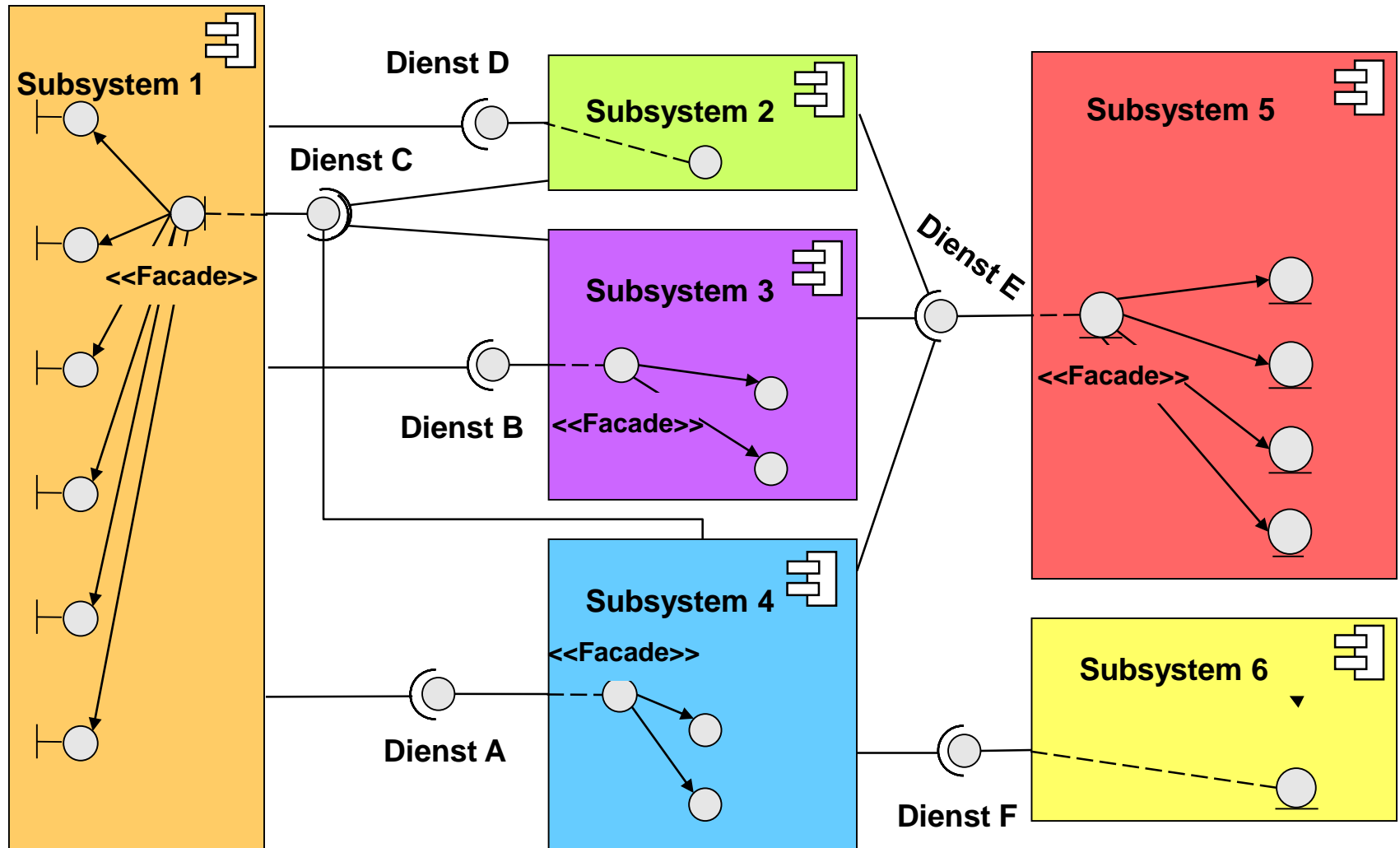
reduziert keine Abhängigkeiten ☹️



System-Dekomposition: Komponenten bieten und nutzen Dienste



System-Dekomposition: Dienste-Realisierung mit Facades



Namensräume versus Subsysteme

Die zwei vorherigen Folien illustrieren, dass Subsysteme viel mehr sind als Packages

- **Packages** sind nur Namensräume, keine Kapselungseinheiten
 - ◆ Sie verhindern zufällige Namensgleichheit, erlauben aber Zugriff (via import-Mechanismus)
 - ◆ Sie haben keine eigene Kapselungsgrenze (keine Schnittstelle)
 - ◆ Sie reduzieren somit nicht die Abhängigkeiten (siehe vorvorherige Folien)
- **Subsysteme** werden als Komponenten (s. nächster Abschnitt) realisiert
 - ◆ Sie haben klar definierte Kapselungsgrenzen (Schnittstellen)
 - ◆ Sie begrenzen somit die möglichen Abhängigkeiten (da nur über die Schnittstellen zugegriffen werden kann)

Aufgabe (Diskussion mit Kollegen)

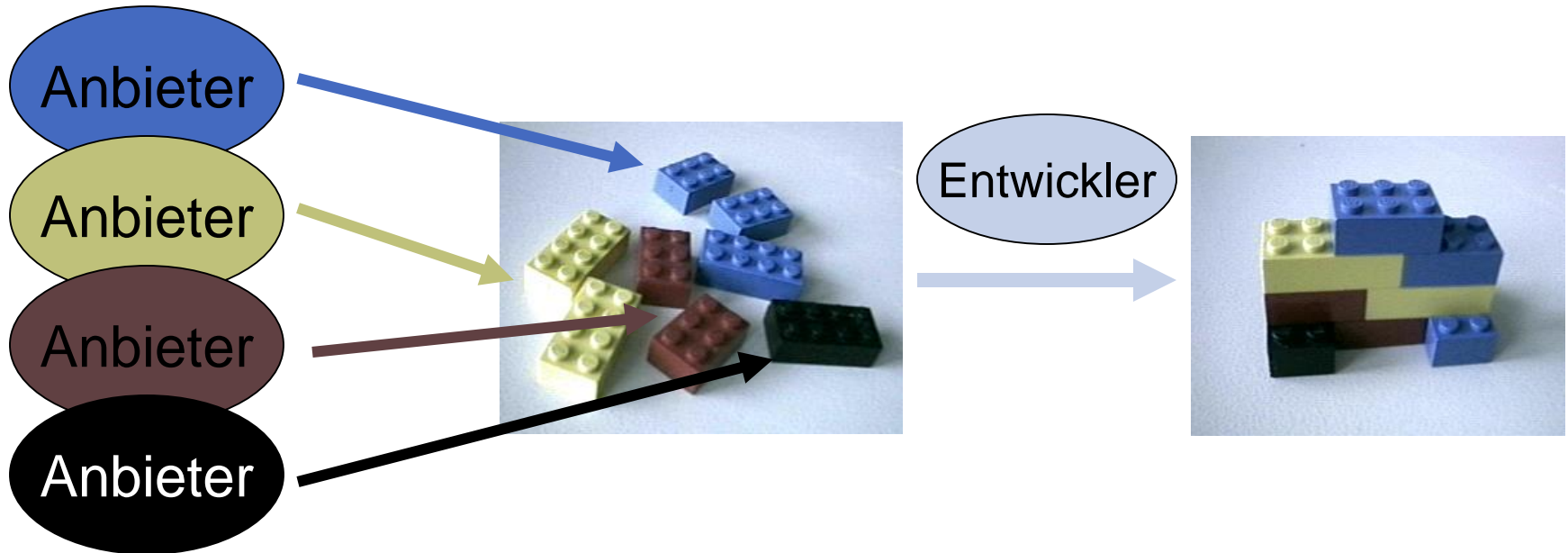
- Überlegen, Sie ob das vorherige Beispiel eine gute oder schlechte Dekomposition darstellt.
- Diskutieren Sie, was für eine Systemdekomposition „gut“ oder „schlecht“ ist.
- Kategorisieren Sie die Dekomposition aus dem Beispiel als eine der nachfolgend vorgestellten Software-Architekturen.
- Passt es genau? Brauchen Sie Änderungen damit es passt?

Patterns für Subsysteme

- Facade
 - ◆ Subsystem abschirmen (gerade vorgeführt)
- Singleton
 - ◆ Nur eine einzige Facade-Instanz erzeugen
- Proxy
 - ◆ Stellvertreter für entferntes Subsystem
- Adapter
 - ◆ Anpassung der realen an die erwartete Schnittstelle
- Bridge
 - ◆ Entkopplung der Schnittstelle von der Implementierung

7.5 Von Subsystemen zu Komponenten

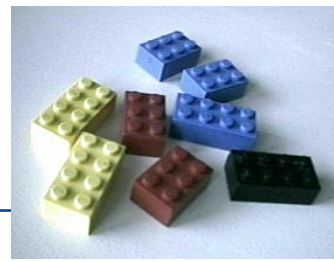
Intuitive Vorstellung



Komponenten-Definition



- Clemens Szyperski , WCOP 1996
 - ◆ „Eine Softwarekomponente ist eine **Kompositionseinheit** mit **vertraglich spezifizierten Schnittstellen** und **nur expliziten Kontextabhängigkeiten**.“
 - ◆ „Eine Softwarekomponente kann **unabhängig eingesetzt** werden und wird **von Dritten zusammengesetzt**.“
- Literatur
 - ◆ Workshop on Component-Based Programming (WCOP) 1996
 - ◆ Clemens Szyperski:
„Component Software – Beyond Object-Oriented Programming“, Addison Wesley Longman, 1998.
 - ◆ Clemens Szyperski, Dominik Gruntz, Stephan Murer:
„Component Software – Beyond Object-Oriented Programming“, Second Edition, Pearson Education, 2002.



- Plattformunabhängige Wiederverwendung von Komponenten
 - günstigere,
 - bessere und
 - schnellere Softwareentwicklung.
- Unterstützung für flexibel anpassbare Geschäftsprozesse
 - ◆ Einfach existierende Dinge zu einem neuen Verbund zusammensetzen
- Fokus auf intelligente Anwendung anstatt der wiederholten (Neu-)Entwicklung des gleichen Basisfunktionalitäten
- Märkte für Komponenten
 - ◆ Möglichkeit Komponenten von Drittanbietern zu kaufen
 - ◆ Möglichkeit Komponenten an andere zu verkaufen

Charakteristika Komponentenbasierter Softwareentwicklung

- Strikte Trennung zwischen Schnittstellen und Implementierung
 - ◆ Die Schnittstellenspezifikation enthält alle Informationen die ein potentieller Benutzer kennen muss. Es gibt keine anderen Kontextabhängigkeiten.
- Verfügbarkeit als Binärcode
 - ◆ Komponenten werden in ausführbarer, binärer Form für viele Plattformen geliefert. Quellcode ist nicht erforderlich.
- Plattformunabhängigkeit
 - ◆ Komponenten können auf einer Vielzahl von Rechnerumgebungen / Betriebssystemen eingesetzt werden.
- Ortstransparenz
 - ◆ Komponenten verwenden oft in Verbindung mit Middleware-Systemen eingesetzt, so dass man nicht wissen braucht, wo sich einzelne Komponenten zur Laufzeit befinden.

Charakteristika Komponentenbasierter Softwareentwicklung

- Wohldefinierter Zweck, der mehr als ein einzelnes Objekt umfasst
 - ◆ Eine Komponente ist auf ein spezifisches Problem spezialisiert
- Wiederverwendbarkeit
 - ◆ Als domänenspezifische Abstraktionen erlauben Komponenten Wiederverwendung auf Ebene von (Teil-)Anwendungen
- Kontextfreiheit
 - ◆ Die Integration von Komponenten sollte unabhängig von einschränkenden Randbedingungen sein.
- Portabilität und Sprachunabhängigkeit
 - ◆ Es sollte möglich sein, Komponenten in (fast) jeder Programmiersprache zu entwickeln.

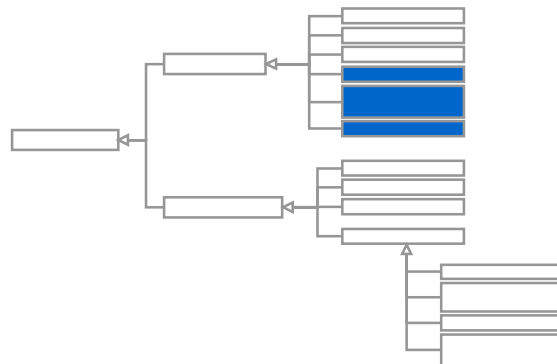
Charakteristika Komponentenbasierter Softwareentwicklung

- Reflektive Fähigkeiten
 - ◆ Komponenten sollten Reflektion unterstützen, so dass die von Ihnen angebotenen und benötigten Dienste durch Introspektion bestimmt werden können.
- Plug & Play
 - ◆ Komponenten sollten leicht einzusetzen sein.
- Konfiguration
 - ◆ Komponenten sollten parametrisierbar sein, damit sie leicht neuen Situationen angepasst werden können.
- Zuverlässigkeit
 - ◆ Komponenten sollten ausgiebig getestet werden.

Charakteristika Komponentenbasierter Softwareentwicklung

- Eignung für Integration / Komposition
 - ◆ Es sollte möglich sein, Komponenten zu komplexeren Komponenten zusammensetzen. Komponenten müssen miteinander interagieren können.
 - ◆ Unterstützung für visuelle Kompositionswerkzeuge

7.5.1 Komponentendiagramme



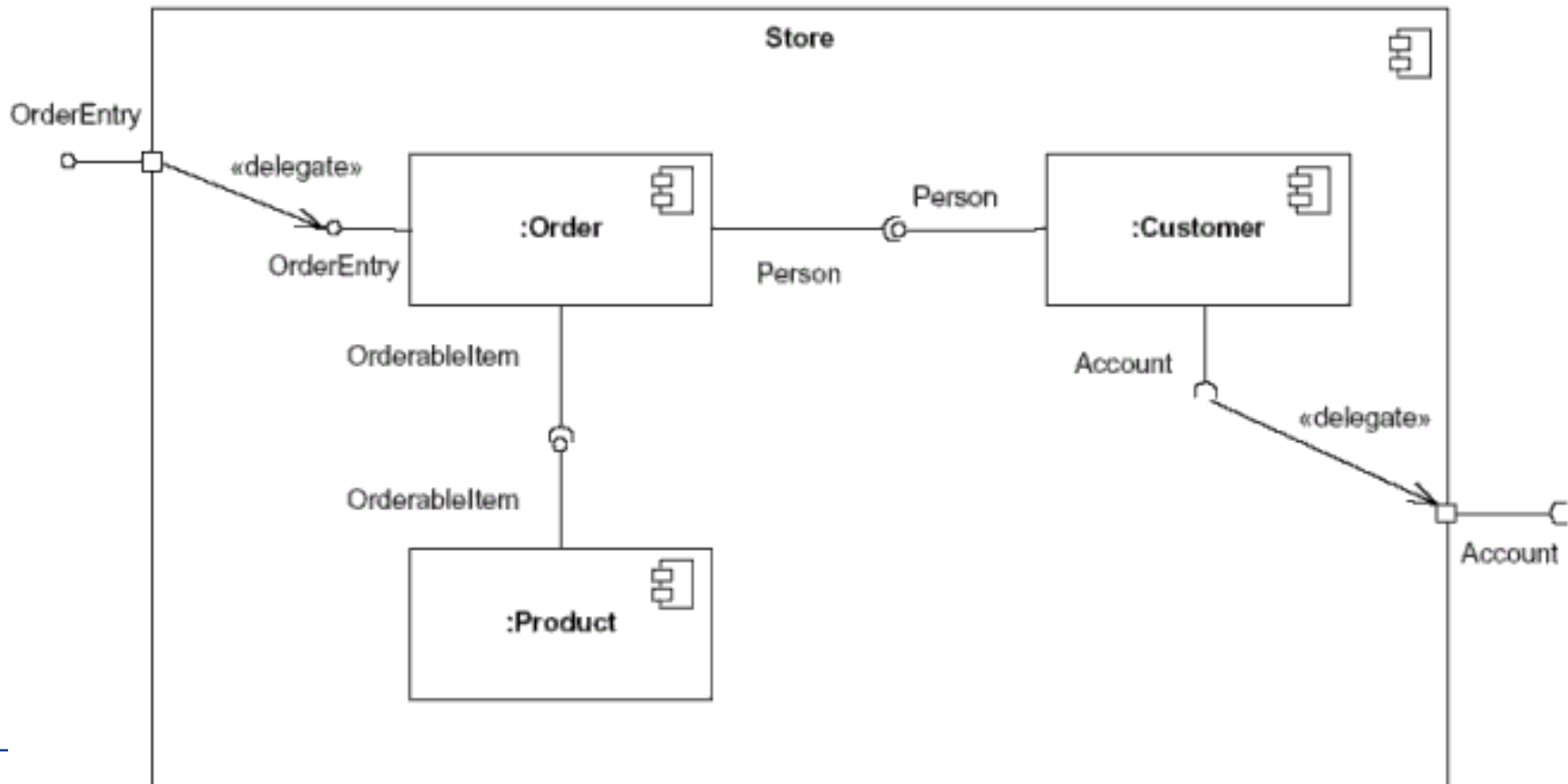
Komponenten

- Kernidee → Nur explizit spezifizierte Kontextabhängigkeiten
 - ◆ Daher auch „benutzte Schnittstellen“ beschreiben!.
- Beispiel
 - ◆ Die Komponente „Bestellung“ (Order) braucht einen „Person“-Dienst um die eigenen Dienste anbieten zu können.



Komponenten: Beispiel

- Komposition
 - ◆ Die „Order“-Komponente nutzt den „Person“-Dienst von „Customer“
- Hierarchische Komponenten
 - ◆ Die „Store“-Komponente besteht ihrerseits aus drei Unterkomponenten

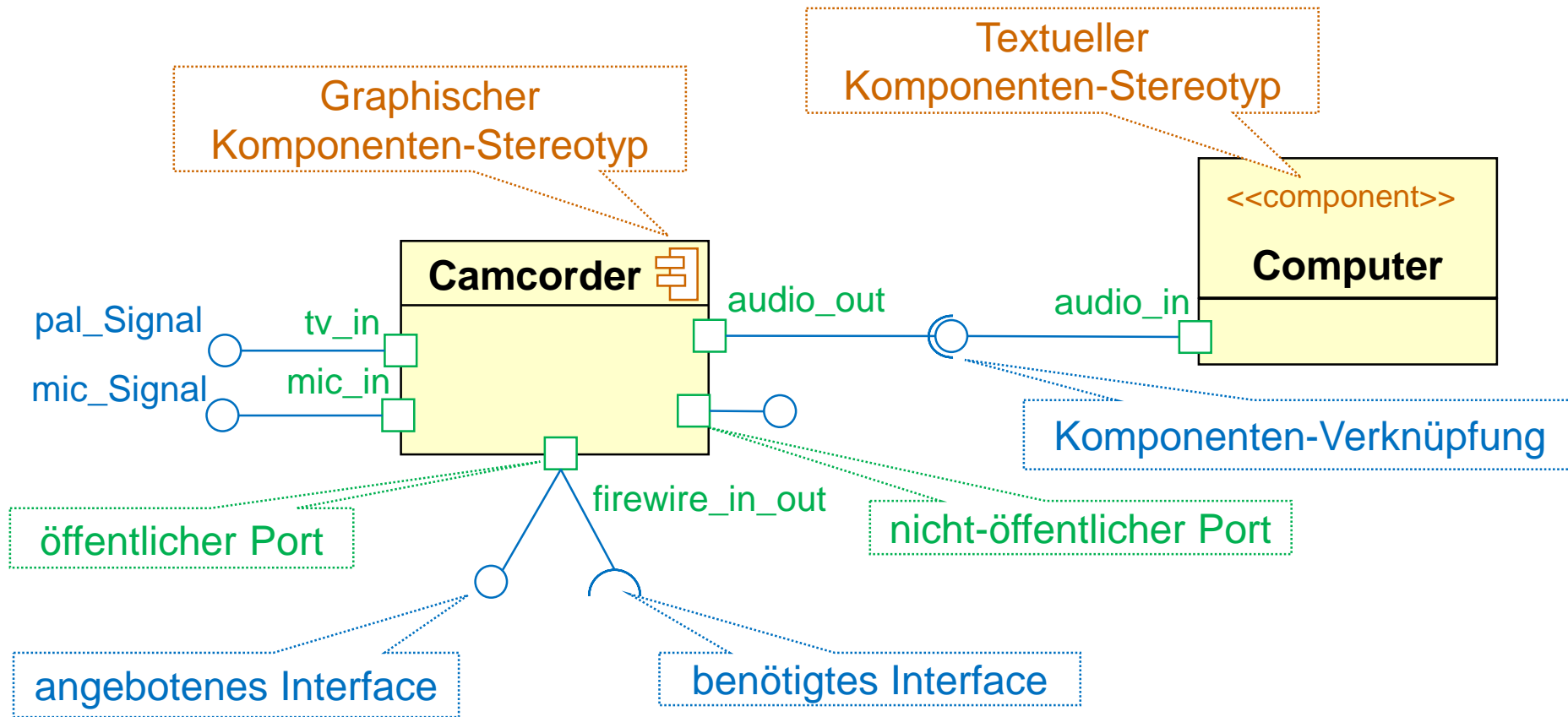


Komponentendiagramme (ab UML 2.0)

Komponentendiagramm zeigt Komponenten und deren Abhängigkeiten

- **Komponenten kapseln beliebig komplexe Teilstrukturen**
 - ◆ Klassen, Objekte, Beziehungen oder ganze Verbände von Teilkomponenten (→ hierarchische Komposition)
 - ◆ Quellcode, Laufzeitbibliotheken, ausführbare Dateien, ...
- **Komponenten haben wohldefinierten Schnittstellen**
 - ◆ Angebotene Schnittstellen (,provided interfaces‘)
 - ◆ Benötigte/benutzte Schnittstellen (,required interfaces‘)
- **Komponenten bieten ‚Ports‘**
 - ◆ Ein Port ist Name für eine Menge zusammengehöriger Schnittstellen
 - ◆ Verschiedene Ports (Namen) für mehrfach vorhandene gleiche Schnittstelle (z.B. mehrere USB-Schnittstellen am gleichen Gerät)

Komponentendiagramm-Elemente an einem Beispiel



Komponenten und Rollen („Parts“)

Komponenten können in Klassendiagrammen verwendet werden und umgekehrt

- Rollen („Parts“)

- ◆ Der gleiche Typ (Interfaces oder Klasse) kann in verschiedenen Komponenten verschiedene Rollen spielen.

- ⇒ „Das engl. Wort „Part“ heißt in diesem Kontext „Rolle“, nicht „Teil“!

- ◆ Notation

- ⇒ Rollen mit Multiplizität

Rolle:Typ [Multiplizität]

Rolle:Typ ^{Multiplizität}

- ⇒ Rolleninstanzen

instanz/Rolle:Typ

b1/Benutzer:Typ

7.6 Spezielle Komponentenmodelle

Verfeinerung von Schnittstellen durch „Behaviour Protocols“
Übersicht von Komponentenmodellen

Interaktionsspezifikation durch „Behaviour Protocoll“

● Gegeben

- ◆ Folgende Schnittstelle

DB_Interface
open(DB_descr) : Connection
close(Connection)
query(Connection,SQL) : ResultSet
getNext(resultSet) : Result

● Problem

- ◆ Wir wissen trotzdem nicht, wie das beabsichtigte Zusammenspiel der einzelnen Methoden ist.
- ◆ Kann man die Methoden in jeder beliebigen Reihenfolge aufrufen?

● Lösung

- ◆ Zu jedem Typ wird sein „Verhaltensprotokoll“ mit angegeben
- ◆ Es ist ein regulärer Ausdruck der legale Aufrufsequenzen und Wiederholungen spezifiziert

● Beispiel

- ◆ „**Erst** Verbindung zur Datenbank erstellen, **dann beliebig oft** anfragen und in jedem Anfrageergebnis **beliebig oft** Teilergebnisse abfragen, **dann** Verbindung wieder schließen.“

```
protocoll DB_Interface_Use =  
  open(DB_descr) ,  
  ( query(Connection,SQL) : ResultSet ,  
    ( getNext(resultSet) : Result )*  
  )* ,  
  close(Connection)
```

SOFA (Software Appliances) Component Model

- SOFA Component (<http://dsrg.mff.cuni.cz/sofa>)
 - ◆ 1. provided and required interfaces
 - ◆ 2. frame (black-box view)
 - ◆ 3. architecture (gray-box view)
 - ◆ 4. connectors (abstract interaction)
 - ◆ 5. behavior protocols associated with 1., 2., 3.
- Behaviour Protocol
 - ◆ incoming event (!)
 - ◆ outgoing event (?)
 - ◆ regular expression describing legal event sequences
- Example
 - ◆ !open, [!query, [!getNext]*]*, !close
- Behaviour protocols enable verification of composition

Weiterführende Literatur zu „Behaviour Protocols“

„SOFA / DCUP“ Projekt an der Karls-Universität Prag, Prof. Plasil und Mitarbeiter

- <http://dsrg.mff.cuni.cz/projects.phtml?p=sofa&q=0>
- Hierarchische Komponenten mit „angebotenen“ und „benutzten“ Schnittstellen
- Spezifikation von Behaviour Protocols für beide Arten von Schnittstellen
- Automatische Verifikation der Protokolleinhaltung bei der Komposition von Komponenten
 - ◆ Horizontale Verbindung von „frames“ untereinander
 - ◆ Vertikale Verbindung von „frame“ mit seiner „architecture“
- Das ganze sogar bei dynamischen Updates der Komposition (d.h. Ersetzung von Komponenten zur Laufzeit)

Übersicht über existierende Komponentenmodelle

	Modell	IDL	Schnittstellen	Ereignisse	Konfiguration	Komposition
Microsoft	COM/DCOM	ja	nur provided	durch Schnittstellen	nein	nein
	COM+	ja	nur provided	Ereignisdienst	Kataloge	nein
	.NET	Gemeinsames Typsystem	nur provided	Nachrichtenserver	Einsatzbeschreibung	nein
Java	JavaRMI	nein	nur provided	durch Schnittstellen	nein	nein
	JavaBeans	nein	nur provided	durch Schnittstellen	Binärdatei	nein
	EJB	nein	nur provided	durch Schnittstellen	Einsatzbeschreibung	nein
OMG	CORBA	ja	nur provided	Ereignisdienst	nein	nein
	CCM	ja	provided und required	Quellen und Verbraucher	Komponentenbeschreibung	Kompositionsbeschreibung
	SOFA	ja	provided und required und behaviour protocols	Quellen und Verbraucher	Komponenten- und Einsatzbeschreibung	Kompositionsbeschreibung