

# Komponenten / Module

## Echtzeitsysteme 2 – Vorlesung/Übung

**Fabian Scheler**  
**Michael Stilkerich**  
**Wolfgang Schröder-Preikschat**

Lehrstuhl für Informatik IV  
Verteilte Systeme und Betriebssysteme  
Friedrich-Alexander Universität Erlangen-Nürnberg

<http://www4.cs.fau.de/~{scheler,mike,wosch}>  
[{scheler,mike,wosch}@cs.fau.de](mailto:{scheler,mike,wosch}@cs.fau.de)



1

# Übersicht

- Identifikation
- Spezifikation
- Modellierung
- Implementierung
- Ergebnis
- Literatur



Fabian.Scheler@cs.fau.de

2

## Teil 1: Identifikation von Modulen

- Was ist ein Modul?
- Wie teilt man ein System in Module auf?
- Welche Module gibt es in meinem System?
- Wie sind diese Module gekoppelt?
- Wie interagieren diese Module?
- Welche Module kann man zusammenfassen?



Fabian.Scheler@cs.fau.de

3

## Was ist eine Komponente / ein Modul?

- Softwarekomponente
  - Klasse
  - Prozedur
  - Package
  - Übersetzungseinheit
- hat eine wohldefinierte Funktion
- hat eine wohldefinierte Schnittstelle



Fabian.Scheler@cs.fau.de

4

# Wie gliedert man ein System in Module?

- **Algorithmus existiert nicht!**
- ist **anwendungsbezogen** – hängt vom System ab
- ist **subjektiv** – hängt vom Designer ab
- mögliche Kriterien
  - **Partitionierung** des Systems
  - **Größe** der Module
  - **Komplexität** der Module
  - **externe Schnittstelle** der Module
  - **Beziehung** und **Kommunikation** der Module untereinander
  - Bereich der **Kontrolle** und des **Einflusses** eines Moduls
- erfordert **Erfahrung**

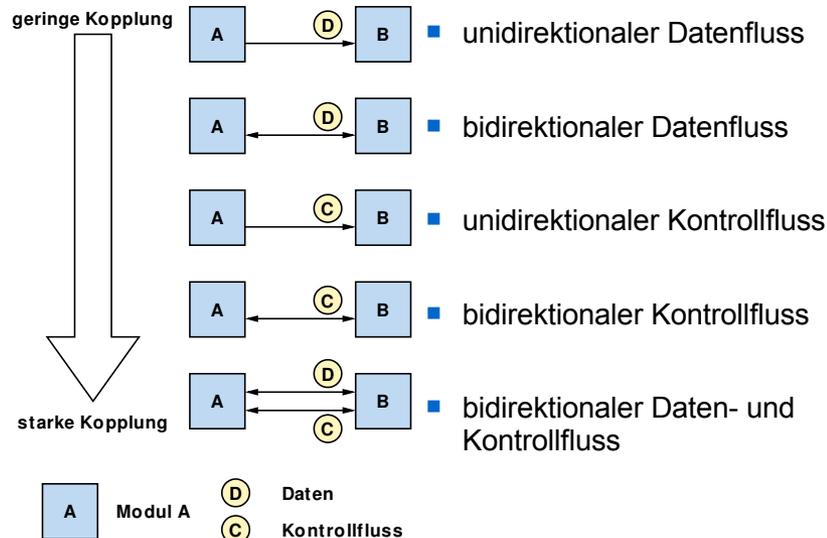


# Kopplung von Modulen

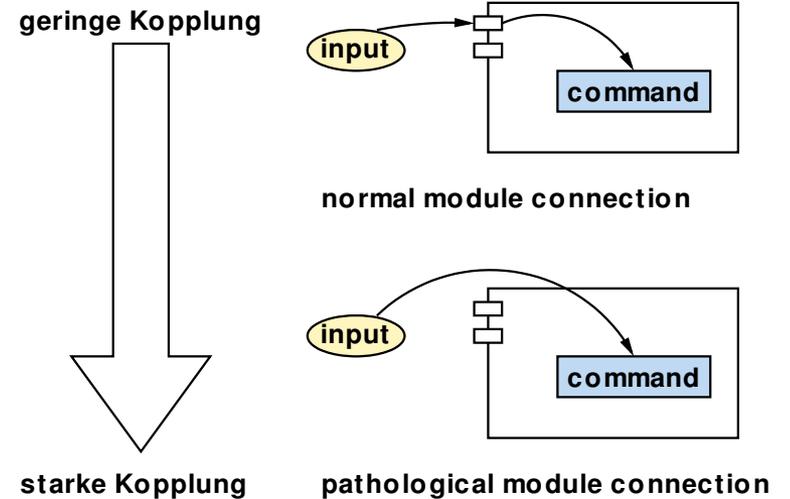
- Maß für die **Unabhängigkeit** von Modulen
- niedrige Kopplung
  - einfach zu verstehen
  - wartbar
  - verlässlich
  - stabil
- hohe Kopplung
  - hohes Maß an „Collateral Evolution“
- Hauptfaktoren
  - Komplexität von Information
  - Informationstyp
  - Kommunikationsmethode
  - Art der Kopplung



# Kopplung - Informationstyp

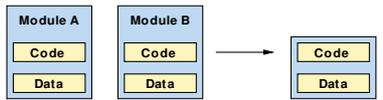


# Kopplung - Kommunikationsmethode



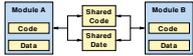
# Kopplung - Art

starker Kopplung



Concept

Implementation



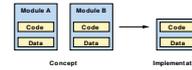
geringe Kopplung

- Content Coupling
  - Module existieren im Konzept
  - keine Trennung in der Implementierung
- Common Coupling
- Stamp Coupling
- Data Coupling – by reference
- Data Coupling – by value

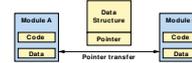
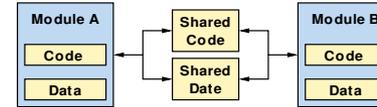


# Kopplung - Art

starker Kopplung



Concept Implementation



geringe Kopplung

- Content Coupling
- Common Coupling
  - Unterscheidung: globale vs. private Daten- und Programmbereiche
  - globale Bereiche von allen Modulen zugreifbar
- Stamp Coupling
- Data Coupling – by reference
- Data Coupling – by value

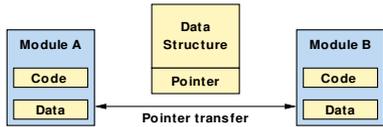
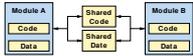


# Kopplung - Art

starker Kopplung



Concept Implementation



geringe Kopplung

- Content Coupling
- Common Coupling
- Stamp Coupling
  - spezifische Datenstruktur statt globaler Bereiche
  - nur bestimmte Bereiche werden geteilt
  - teilende Module müssen konsistente Sicht auf die Datenstruktur haben
- Data Coupling – by reference
- Data Coupling – by value

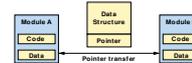
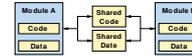


# Kopplung - Art

starker Kopplung



Concept Implementation



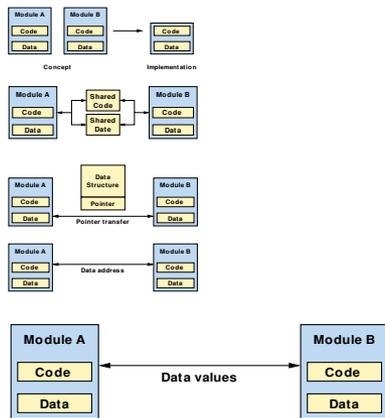
geringe Kopplung

- Content Coupling
- Common Coupling
- Stamp Coupling
- Data Coupling – by reference
  - Zugriff auf Daten per Referenzen
  - Konsistenz der Daten durch Programmiersprache sicher gestellt
- Data Coupling – by value



# Kopplung - Art

starker Kopplung

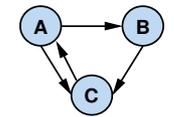
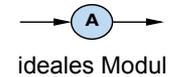


- Content Coupling
- Common Coupling
- Stamp Coupling
- Data Coupling – by reference
- Data Coupling – by value
  - nur Werte werden weiter gegeben
  - Module können Zustände anderer Module nicht ändern
  - sehr sicher
  - hoher Speicherbedarf

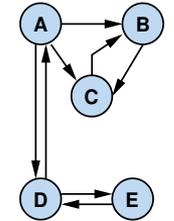
geringe Kopplung

# Kohäsion

- ideales Module
  - sehr einfach
  - 1 Eingang, 1 Ausgang
- einfache Systeme
  - übersichtlich
- komplexere Systeme
  - Systemkomplexität steigt schnell an
  - unübersichtlich
- Kompromiss
  - einfache Module vs. einfaches System



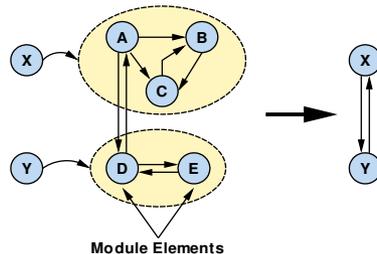
einfaches System



komplexeres System

# Kohäsion

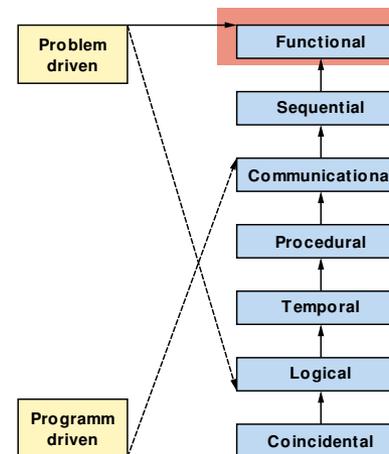
- fasse Module zu Super-Modulen zusammen
  - Module werden Elemente des Super-Moduls



- Kohäsion: ein Maß für den Zusammenhalt der Elemente
  - Anzahl der Verbindungen untereinander
  - Anzahl der Verbindungen nach außen

# Kohäsion – Arten (Sommerville)

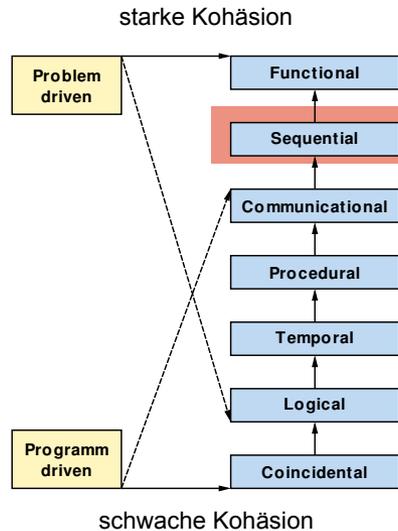
starke Kohäsion



schwache Kohäsion

- Funktional
  - fasst Elemente nach ihrer Funktion zusammen
  - ein funktionales Modul erfüllt genau eine Funktion
  - kein Element ist redundant
  - schlecht definierbar

## Kohäsion – Arten (Sommerville)

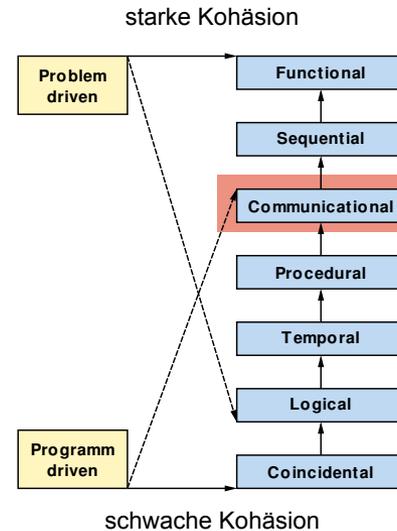


### ■ Sequentiell

- Elemente formen eine lineare Sequenz
- Ausgabe des einen Elements ist die Eingabe des nächsten Elements
- gut geeignet für die Abbildung von Datenflüssen



## Kohäsion – Arten (Sommerville)

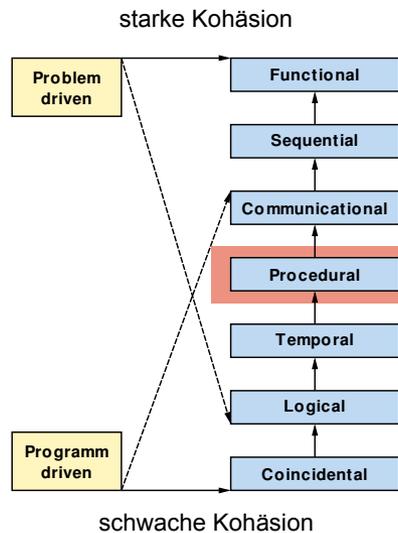


### ■ Kommunikativ

- Elemente arbeiten auf gemeinsamen Daten
- Verknüpfung der Elemente entsteht durch die Operationen auf den Daten



## Kohäsion – Arten (Sommerville)

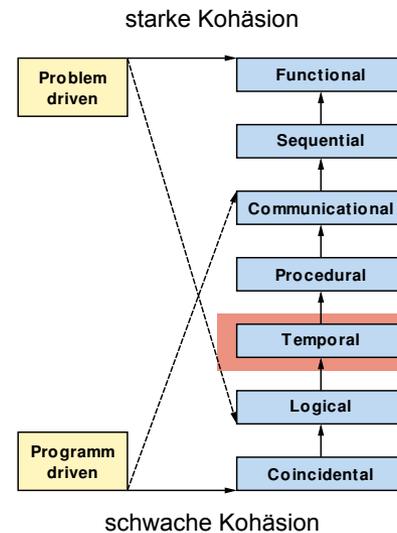


### ■ Prozedural

- jedes Element ist Teil einer Prozedur
- alle Elemente sind notwendig, wenn die Prozedur ausgeführt wird



## Kohäsion – Arten (Sommerville)

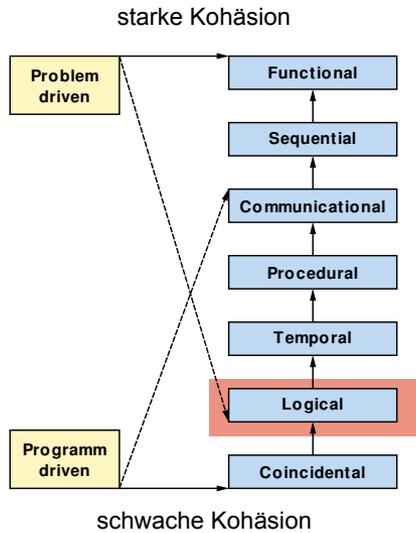


### ■ Zeitlich

- alle Elemente hängen zeitlich voneinander ab (z.B. Initialisierung)
- Elemente ohne Relation werden zusammen gruppiert

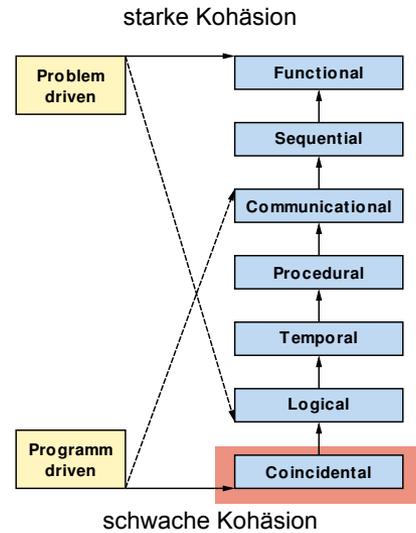


# Kohäsion – Arten (Sommerville)



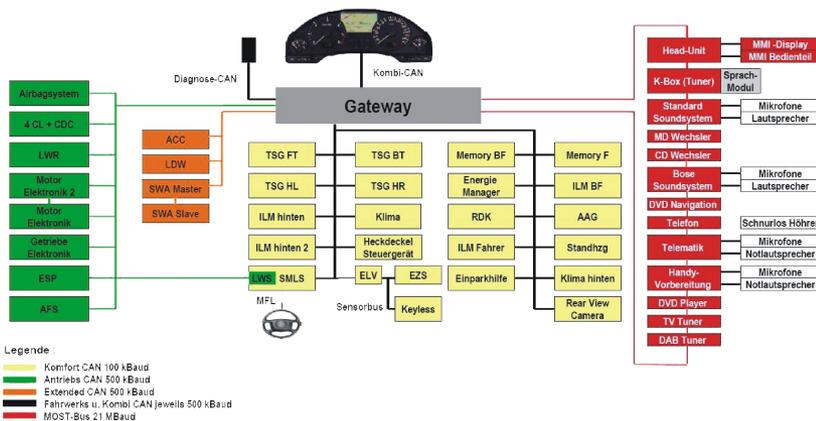
- Logisch
  - Elemente hängen logisch zusammen (z.B. ein Filter besteht aus einer Gruppe von Prozeduren)
  - Vorsicht: Module enthält mehr als Funktionalität → schlechtere Wartbarkeit
  - relativ gut geeignet für eingebettete Systeme

# Kohäsion – Arten (Sommerville)



- Zufällig
  - Elemente stehen nicht in einer bestimmten Beziehung
  - entsteht z.B. durch Code-Sharing
  - Indikator für schlechtes Design

# Beispiel: KFZ



# Identifikation – Zusammenfassung

- minimiere Kopplung
- maximiere Kohäsion
- Unterstützung durch die Programmiersprache
  - Modulkonzept in C: schwach ausgeprägt
  - Besser: C++ - Klassen
  - explizites Modulkonzept: Modula
- orientiere dich am realen System

## Teil 2: Spezifikation

- informelle Beschreibung des Moduls
- Abhängigkeiten
  - von welchen anderen Modulen hängt dieses Modul ab
  - welche anderen Module hängen von diesem Modul ab
- Schnittstelle
  - Syntax
  - Semantik
  - Vor- und Nachbedingungen
- Beschreibung der Schnittstelle
  - Zustandsdiagramme
  - Sequenzdiagramme
  - Aktivitätsdiagramme
  - informell

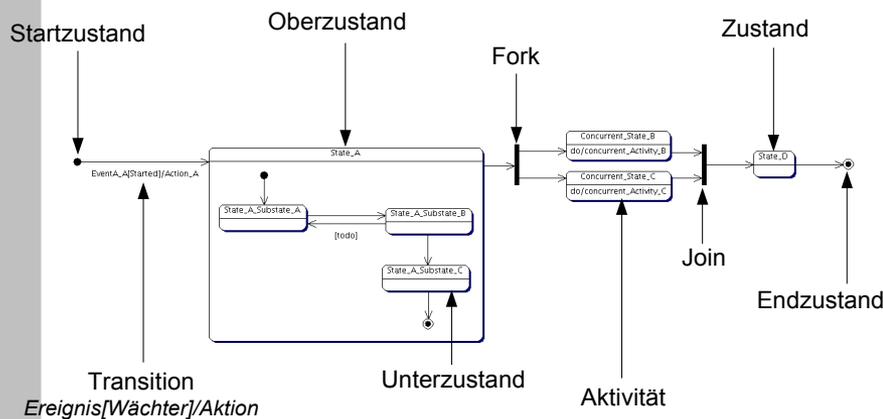


## Zustandsdiagramme

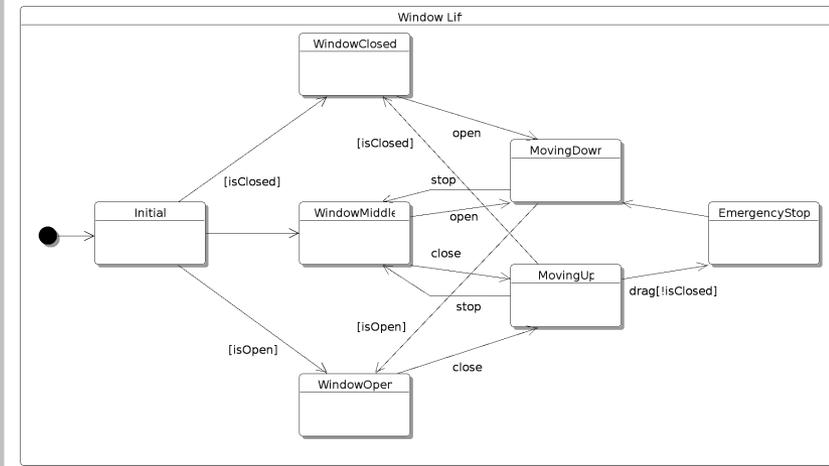
- beschreiben das Verhalten eines Systems
  - mögliche Zustände
  - Zustandsübergänge in Verbindung mit bestimmten Ereignissen
- ✓ gut geeignet für die Beschreibung eines Moduls
- ✗ weniger geeignet für die Beschreibung mehrerer, interagierender Module



## Zustandsdiagramme



## Beispiel: elektrischer Fensterheber

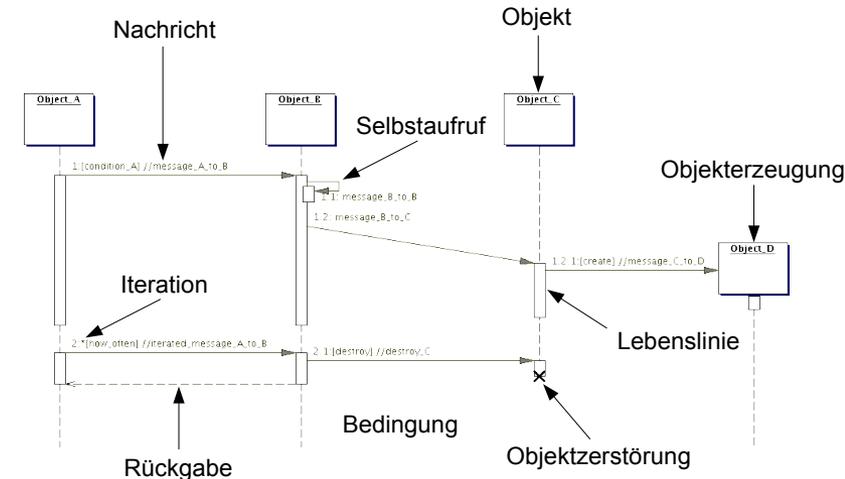


# Sequenzdiagramme

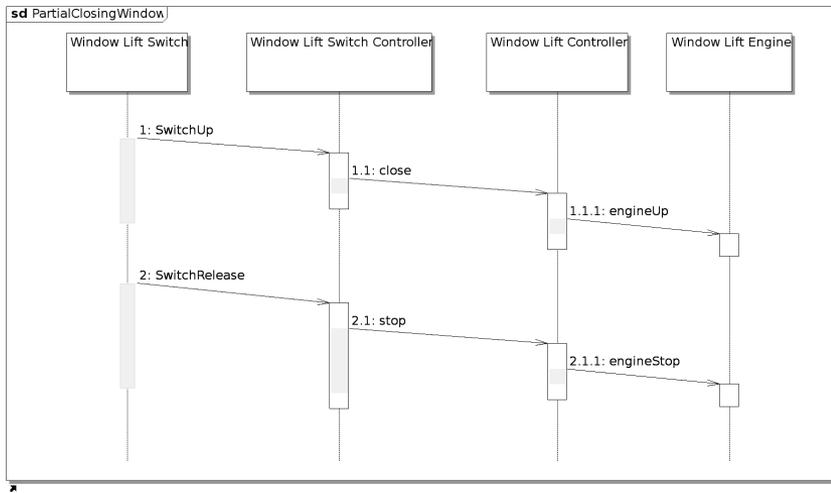
- Darstellung von
  - Kontrollflussinformation
  - nebenläufigen Prozessen
- ✓ Sequenzdiagramme sind gut geeignet, um
  - das Verhalten mehrerer Objekte/Modulen in einem Anwendungsfall darzustellen
  - die Zusammenarbeit zwischen mehreren Objekten/Modulen aufzuzeigen
  - zeitliches Verhalten präzise zu definieren
- ✗ Sequenzdiagramme sind weniger geeignet, ...
  - um das Verhalten einfacher Objekte/Module in mehreren Anwendungsfälle darzustellen (→ Zustandsdiagramme, → Aktivitätsdiagramme)



# Sequenzdiagramme



# Beispiel: elektrischer Fensterheber

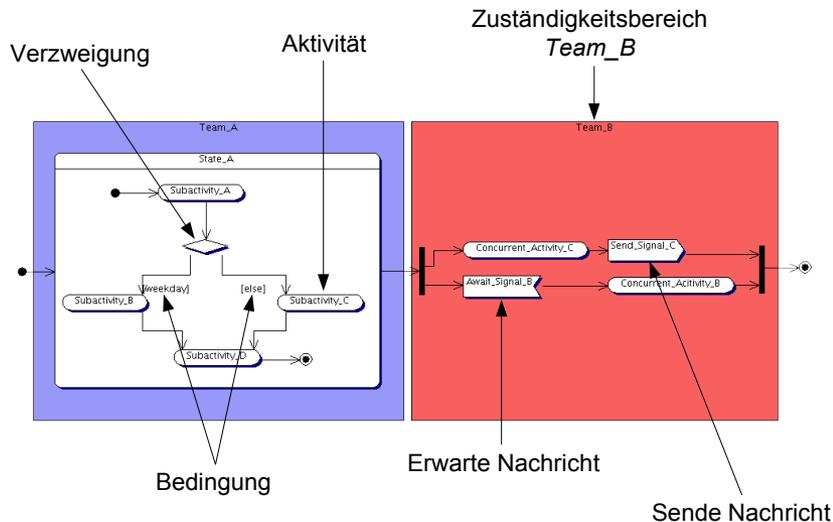


# Aktivitätsdiagramme

- Mischung aus
  - SDL
  - Workflow-Modellierung
  - Petri-Netzen
- Stärke: nebenläufiges Verhalten
- unklare Zuordnung zwischen Aktivität und Objekt
- ✓ gut mit Aktivitätsdiagrammen darzustellen:
  - Workflows
  - nebenläufige Aktivitäten
  - komplizierte, sequentielle Algorithmen
- ✗ weniger gut mit Aktivitätsdiagrammen darzustellen:
  - Kooperation mehrerer Objekte / Module (→ Sequenzdiagramme)
  - Verhalten eines Objekts / Moduls (→ Zustandsdiagramme)



## Aktivitätsdiagramme



## Teil 3: Modellierung

- ein Modell ist ...
- **keine** Spezifikation!
- **abstrakte** Implementierung der Spezifikation
  - oft basierend auf formalen Methoden
    - Zustandsautomaten, Petri-Netze
    - formale Modellierungssprachen
    - Logik
  - geeignet für die Verifizierung des Systems
  - geeignet für die automatische Erzeugung einer Implementierung
- kommt dem Verhalten des realen Systems sehr nahe
- **nicht** Bestandteil dieser Veranstaltung



## Teil 4: Implementierung

- Verwendete Programmiersprache: C
- Modulkonzept in C: schwach ausgeprägt
- hohe Disziplin des Entwicklers notwendig
- muss überprüft / sicher gestellt werden
  - Metriken
  - Coding-Guidelines
  - Code Reviews



## Module in C

- Schnittstelle: Header
- Implementierung: Übersetzungseinheit
- private Programmstrukturen: Schlüsselwort `static`
- globale Programmstrukturen: Schlüsselwort `extern`
- Wie implementiert man
  - Content Coupling?
  - Common Coupling?
  - Stamp Coupling?
  - Data Coupling – by reference?
  - Data Coupling – by value?



## Ergebnis

---

- Gliederung des Systems in Komponenten
- Spezifikation der Komponenten
  - informelle Beschreibung der Komponenten
  - syntaktische und semantische Beschreibung der Schnittstelle
  - Interaktion mit und Abhängigkeiten von anderen Komponenten
- Implementierung der Komponenten in C



## Literatur

---

- *Grady Booch, James Rumbaugh, and Ivar Jacobson.*  
**The Unified Modeling Language User Guide.**  
Object Technology Series. Addison-Wesley, Reading/MA, 1999.
- *Jim Cooling.*  
**Software Engineering for Real-Time Systems.**  
Addison-Wesley, 2003.
- *Bruce Powel Douglass.*  
**Real-Time UML.**  
Addison-Wesley, Reading, USA, 1998.
- *Martin Fowler and Kendall Scott.*  
**UML konzentriert.**  
Addison-Wesley, 2nd edition, 2000.

