



Grundlagen der Softwarearchitektur (im Softwarepraktikum)



ORGANISATORISCHES

Inhalt

- Was ist Softwarearchitektur?
- Dokumentieren mit UML
- Wie bewerte ich eine Softwarearchitektur?
- Wie plane ich eine Softwarearchitektur?
- Metriken



DISCLAIMER



WAS IST SOFTWAREARCHITEKTUR?

Was ist Softwarearchitektur?

Definition. Eine **Softwarearchitektur** beschreibt die **Strukturen** eines Softwaresystems durch **Architekturbausteine** und ihre **Beziehungen** und **Interaktionen** untereinander sowie ihre physikalische Verteilung. Die **extern sichtbaren** Eigenschaften eines Architekturbausteins werden durch **Schnittstellen** spezifiziert.

- Verschiedene **Sichten**:
Statisch, **Dynamisch**, Verteilung, Kontext
- Verschiedene **Bausteinararten**:
Subsysteme, **Komponenten**, Frameworks, **Pakete**, **Klassen**
- Softwarearchitektur ist ein **Modell** eines Softwaresystems.

Was ist ein Modell?

Definition. [Glinz, 2008, 425]

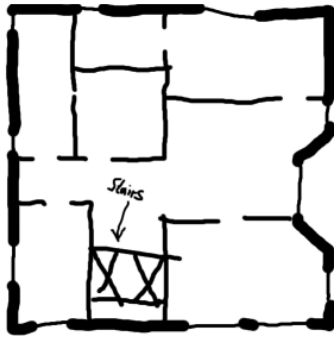
Ein **Modell** ist ein konkretes oder mentales **Abbild** von etwas oder ein konkretes oder mentales **Vorbild** für etwas.

Drei Eigenschaften kennzeichnen ein Modell:

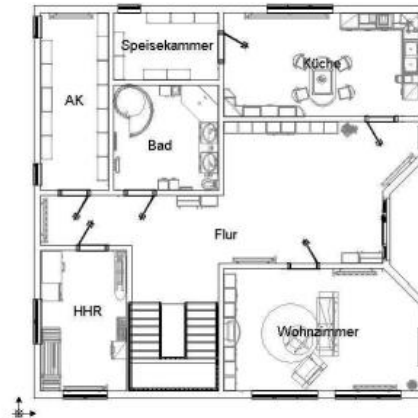
- (i) Das **Abbildungsmerkmal**, d.h. es gibt eine Entität (ein **Original**) dessen Abbild oder Vorbild das Modell ist,
- (ii) das **Verkürzungsmerkmal**, d.h. nur die Eigenschaften des Originals die für den Modellierungskontext relevant sind werden repräsentiert, und
- (iii) die **Pragmatik**, d.h. das Modell wurde in einem spezifischen Kontext für einen spezifischen Zweck erstellt.

Modell-Modi

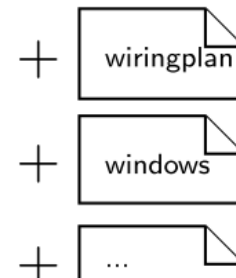
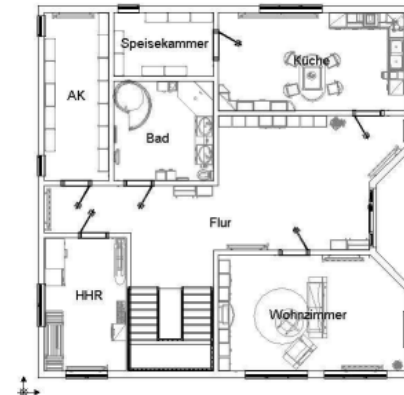
... als Skizze



... als Blaupause



... als Programmiersprache



[Westphal, 2012/13, UML]

[<http://martinfowler.com/bliki>]

Wieso Softwarearchitektur planen?

- **Kommunikation** und **Dokumentation**
 - der Realisierung.
 - der Entwurfsentscheidungen.
- **Dekomposition**
 - Schnittstellen identifizieren.
 - Aufgaben parallelisieren.
- **Qualität** planen
 - Konzeptionelle Integrität.
 - Grundlage für **Bewertung** anhand von **Szenarien**.
 - Vereinfachung der **Wiederverwendung** von Systembestandteilen.

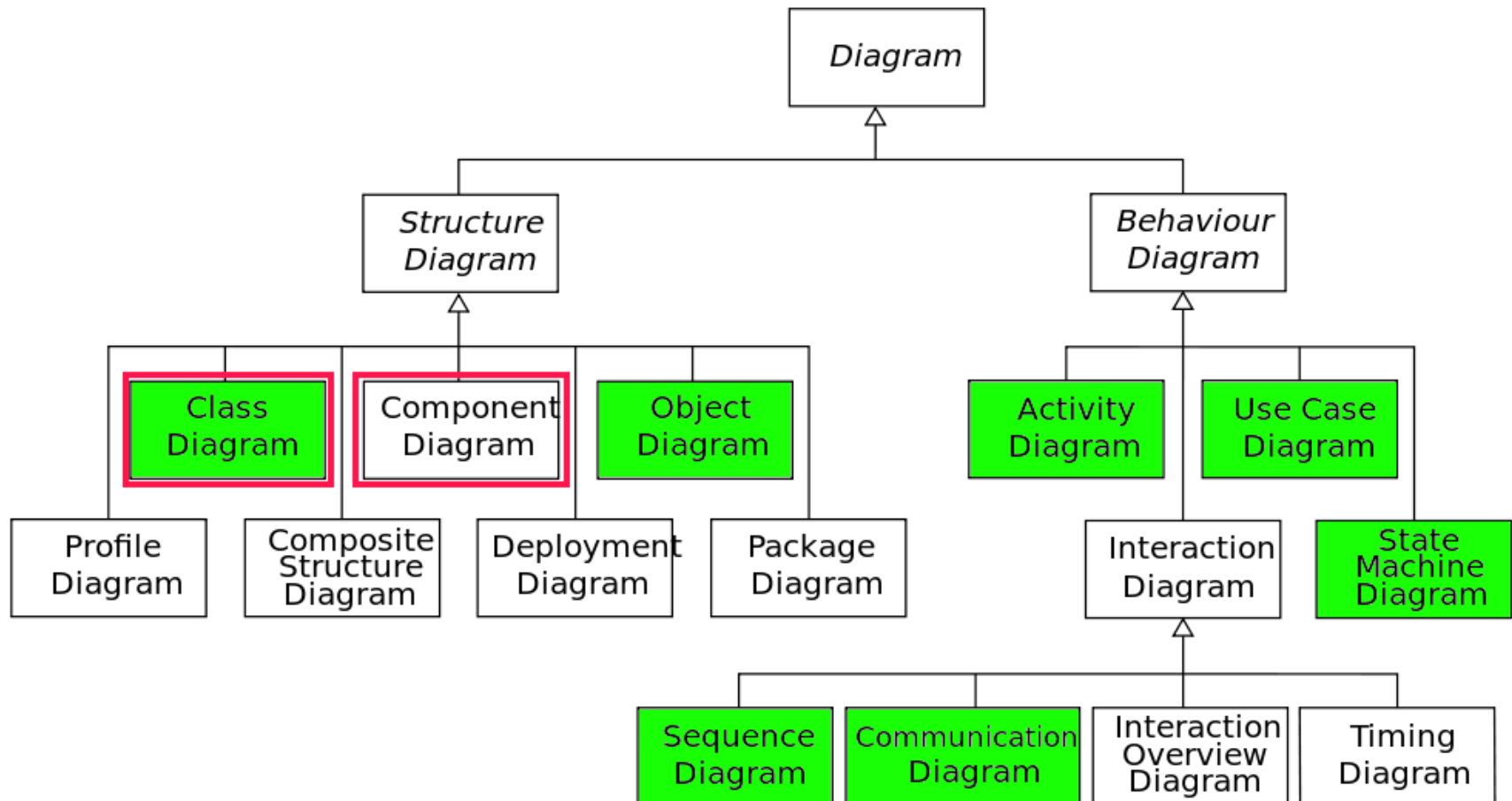


UML

Was ist UML?

- UML ist eine **Modellierungssprache** mit graphischen Notationen.
- UML ist die Abkürzung für **Unified Modeling Language**.
- UML ist standardisiert, der Standard wird von der **Object Management Group (OMG)** verwaltet.

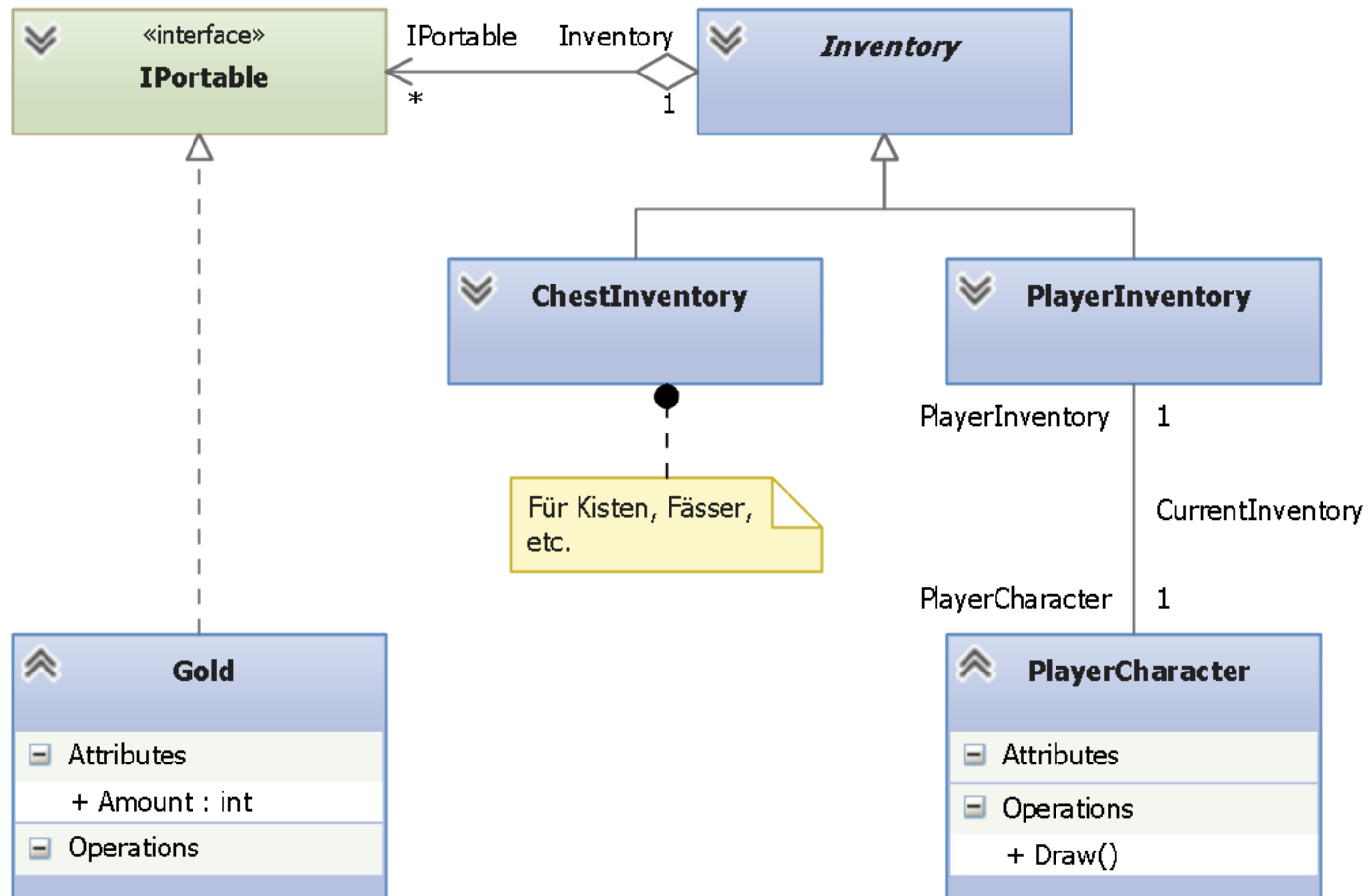
Diagrammarten in UML



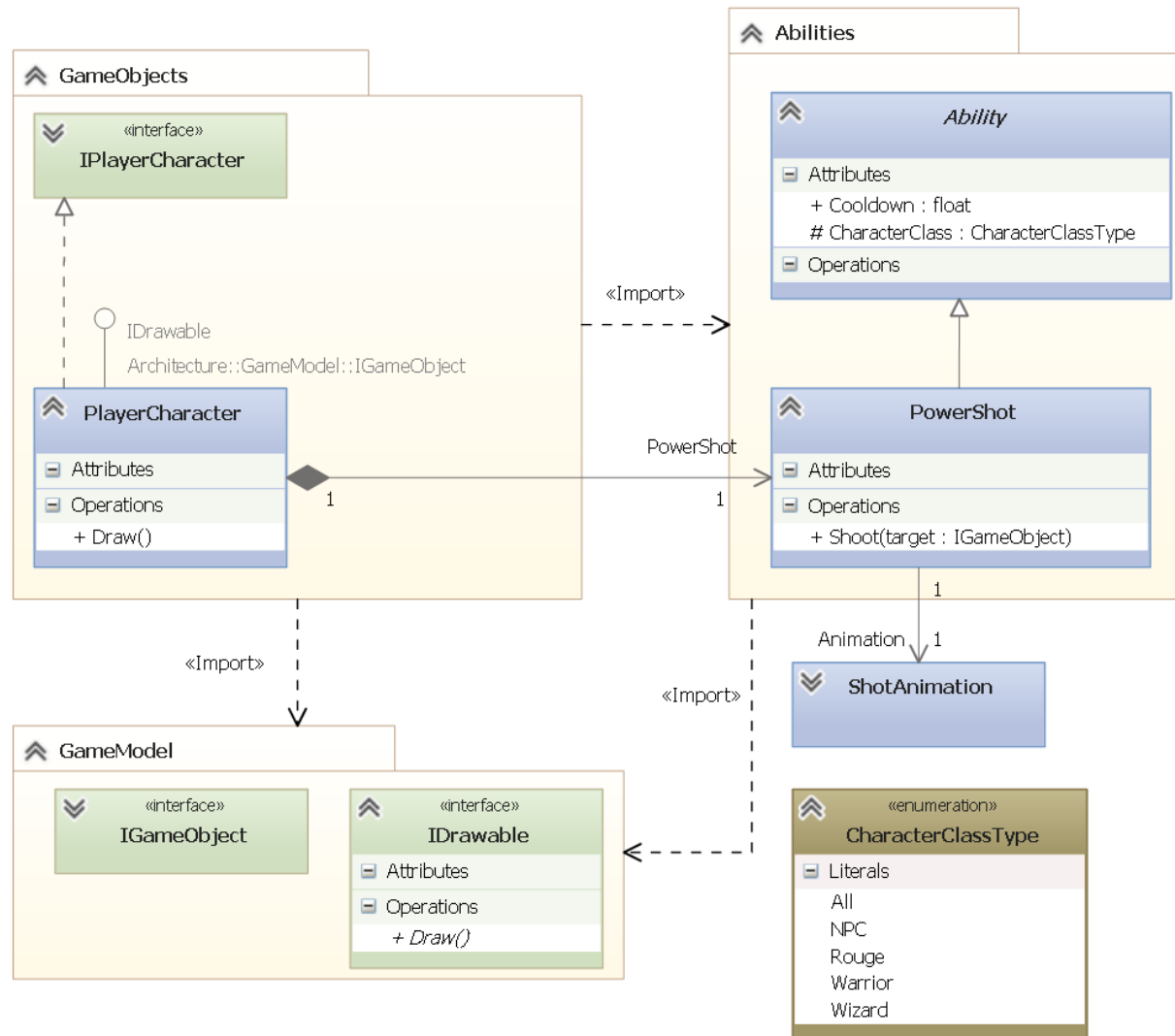
Klassendiagramm

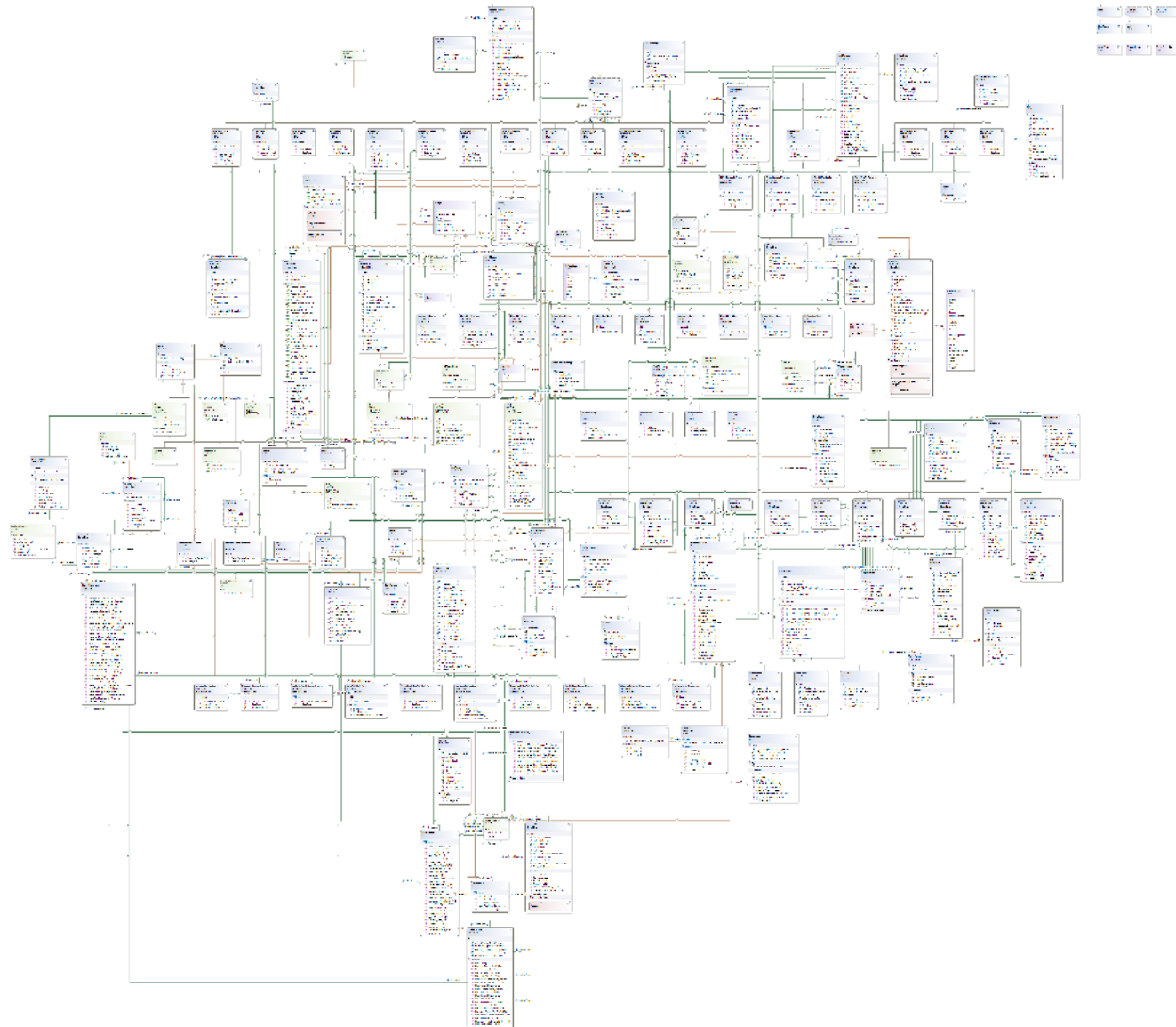
- Statische Sicht.
- Klassen, Interfaces, Pakete und deren statische Beziehungen als Bausteine.
- Sehr nah an der Implementierung.
- Hier: Klassendiagramm à la Microsoft Visual Studio.

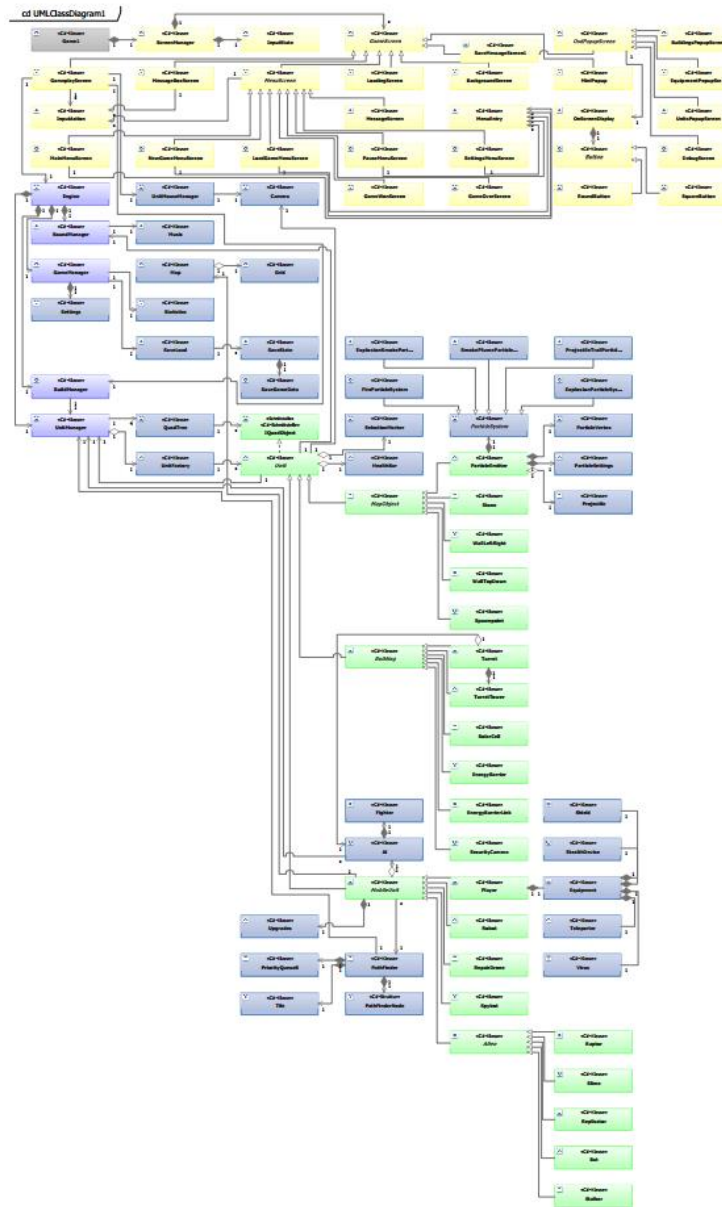
Klassendiagramm



Klassendiagramm







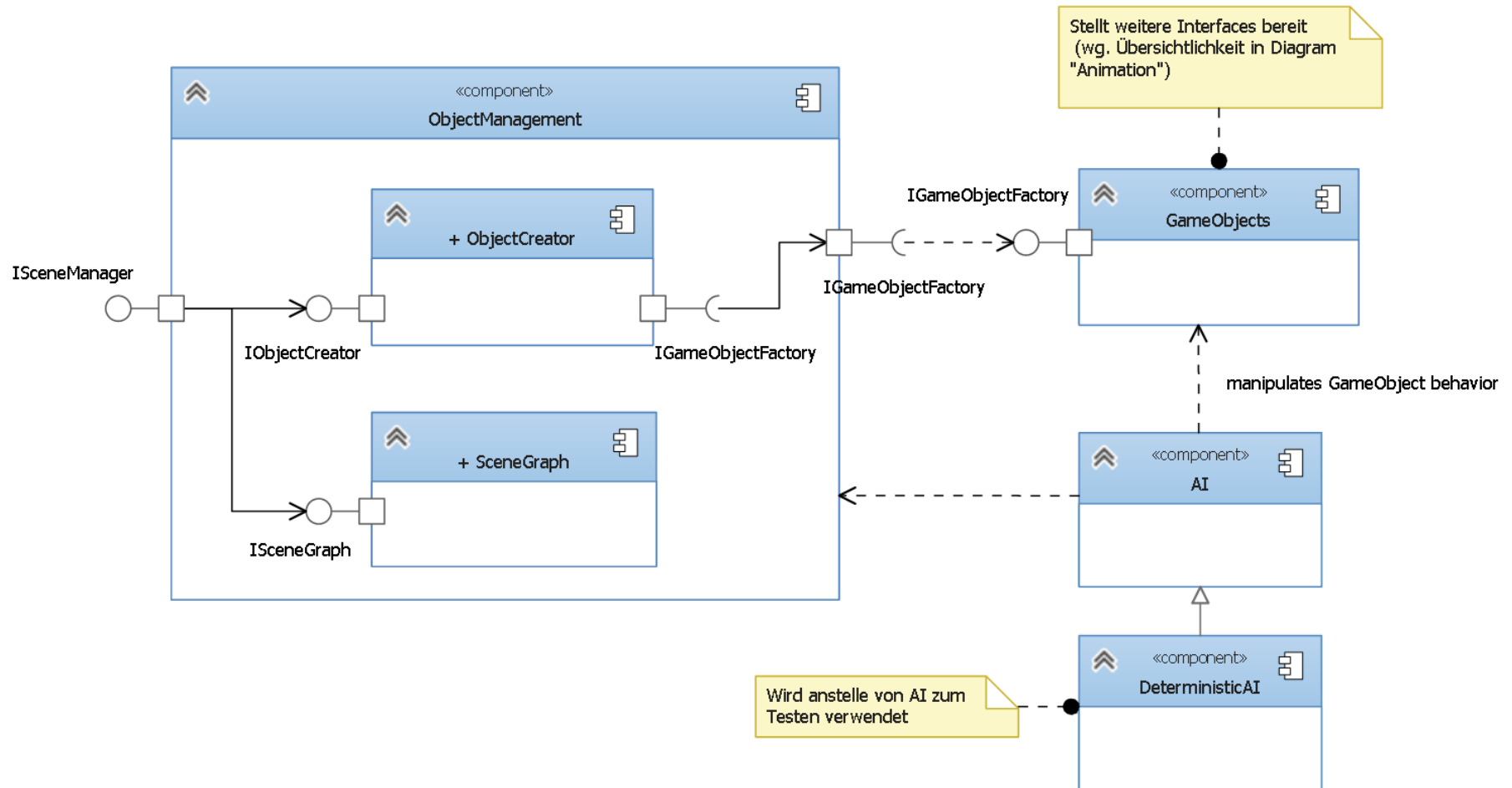
Komponenten

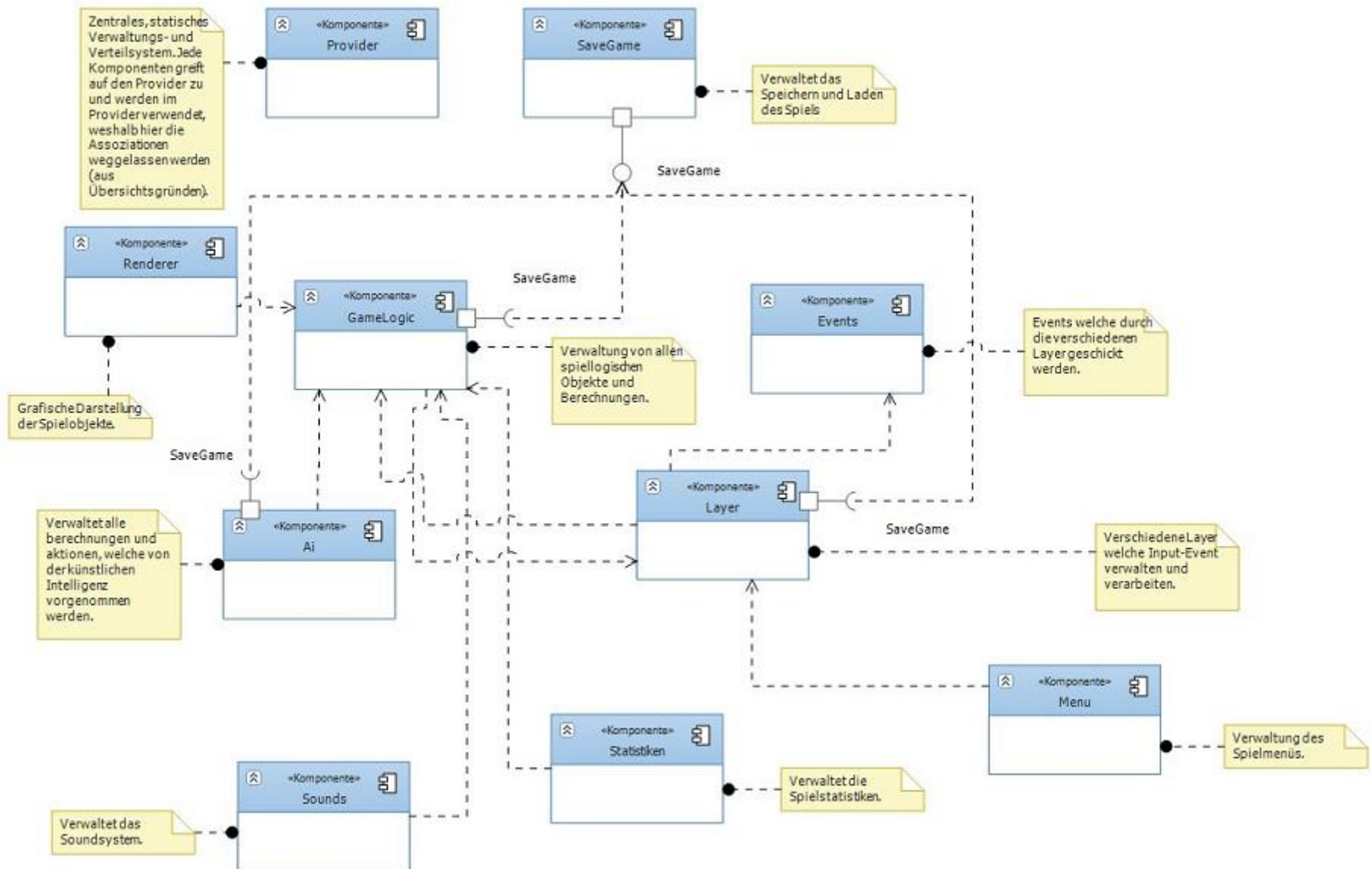
- Eine **Komponente** ist ein **Softwarebaustein**, der Dritten **Funktionalität** über **Schnittstellen** zur Verfügung stellt und **nur explizite Abhängigkeiten** nach außen besitzt.

Komponentendiagramm

- **Statische** Sicht.
- **Komponenten**, **Interfaces**, **Parts** und deren statische Beziehungen untereinander als Bausteine.
- **Abstrakter** als Klassendiagramm.
- Beschreibt „Verdrahtung“ von **Komponenten**.
- **Hier**: Komponentendiagramm à la Microsoft Visual Studio.

Komponentendiagramm





Werkzeugunterstützung

- Sie können **jede Art von Werkzeug** verwenden. Z.B.:
 - Visual Studio Ultimate
 - ArgoUML mit C# Importer
 - ModelMaker
 - Grafikprogramme
 - Papier, Stifte, Scanner
- **Visual Studio Ultimate**
 - Klassen- und Komponentendiagramme können mit Visual Studio **gezeichnet** werden.
 - Klassendiagramme können **generiert** werden (sehen dann aber anders aus).
 - Aus den Diagrammen kann auch **Code erzeugt** werden (ist aber nicht bzw. nur mit hohem Aufwand nützlich).



WIE BEWERTE ICH EINE SOFTWAREARCHITEKTUR?

Qualitätsmerkmale nach ISO 9126 (bzw. 25000)

- Funktionalität
Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- Zuverlässigkeit
Reife, Fehlertoleranz, Wiederherstellbarkeit
- *Benutzbarkeit*
Verständlichkeit, Erlernbarkeit, Bedienbarkeit
- *Effizienz*
Zeitverhalten, Verbrauchsverhalten
- Änderbarkeit
Analysierbarkeit, Modifizierbarkeit, Stabilität, Prüfbarkeit
- Übertragbarkeit
Anpassbarkeit, Installierbarkeit, Konformität, Austauschbarkeit

Qualitätsmerkmale im Softwarepraktikum

- **Funktionalität**
 - Testing
- **Benutzbarkeit**
 - Cognitive Walkthrough
 - Heuristic Evaluation
- **Effizienz**
 - Profiling
 - Testing (FPS mit fraps)
 - Szenarien
- **Änderbarkeit**
 - Szenarien
 - Abschätzung mit Metriken, z.B. Efferent / Afferent **Coupling**, Lack of **cohesion** of methods (**LCOM**), Type **complexity**



WIE PLANE ICH EINE SOFTWAREARCHITEKTUR?

Wie plane ich eine Softwarearchitektur?

- **Schlechte Nachricht:**
 - Es gibt **kein** deterministisches Verfahren, das in jedem Fall zu guten Softwarearchitekturen führt.
- Was nun?
 - Es gibt grundlegende Aktivitäten, Heuristiken, etc.
 - Z.B. Softwaretechnik-Vorlesung „**Object-Oriented Analysis / UML Diagrams**“.

Wie plane ich eine Softwarearchitektur?

- Informationen über das **Problem** sammeln
 - **Anforderungen** (GDD)
 - **Randbedingungen** und **technischer** Kontext (MSDN zu C#, XNA, eventl. F#)
 - **Domänenwissen** (Bücher, div. Websites, Zeitschriften, nächste Vorlesung)
- **Fachbegriffe** sammeln und **verstehen**
 - **Kernaufgabe** des Systems (vgl. „Zusammenfassung des Spiels“) in wenigen Sätzen und mit eigenen Begriffen beschreiben.
 - **Gemeinsame Sprache** schaffen

Wie plane ich eine Softwarearchitektur?

- Informationen über **Lösungen** sammeln
 - Wer hat eine ähnliche Aufgabe schon vorher gelöst, und wie?
 - **Architekturstile** und **Entwurfsmuster**
 - **Quellen**: eigene Projekte, Literatur, Internet, etc.
- Arbeiten Sie **iterativ** und **inkrementell**.
- **Validieren** Sie **frühzeitig** durch Implementierungen.
- Verwenden Sie **Szenarien**.

Szenarien

- Beispiel:
 - „Die Erstellung und Einbindung eines neuen 3D-Modells durch einen Modellierer muss innerhalb von 6h abgeschlossen sein.“
 - „Das Spiel muss im Endlosmodus 1000 aktive, durch den Spieler steuerbare Spielobjekte gleichzeitig auf einem Screen mit mindestens 45 FPS auf der Referenzhardware darstellen können.“
- Szenarien sind **Ablaufbeschreibungen** mit den folgenden **Eigenschaften**:
 - Auslöser (z.B. Modellierer, Spieler)
 - Quelle (z.B. intern, extern, Benutzer, Betreiber, Angreifer)
 - Umgebung (z.B. Endlosmodus, Entwicklung)
 - Systembestandteil (z.B. alle, Input, KI)
 - Antwort (z.B. Modell erfolgreich eingebunden, gleichzeitige Darstellung 45FPS)
 - Antwortmetrik (z.B. mehr oder weniger als 45FPS)

Wie plane ich eine Softwarearchitektur?

- **Clean Code** enthält viele Hinweise
 - Nahezu alle Clean Code Prinzipien beziehen sich auf Architektur.
- Grundideen
 - **Abhängigkeiten** zwischen Bausteinen verringern:
Niedrige **Kopplung**, hohe **Kohäsion**
 - **Open-Closed Principle**:
Offen gegen Erweiterungen, geschlossen gegen Änderungen.



ENTWURFSMUSTER

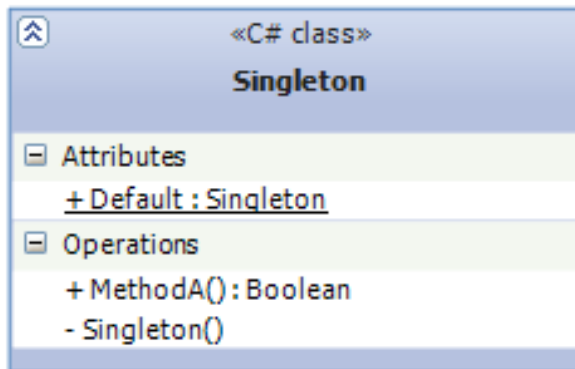
Entwurfsmuster

- Siehe Softwaretechnik-Vorlesung „Design Patterns“.
- Für Spiele:
 - Sehr nützlich:
Composite, (Abstract) Factory, Builder, Flyweight, Observer, Visitor, Iterator
 - Vielleicht nützlich:
Object Pool, Proxy, Prototype, Composite, Decorator, Command, Strategy, ...

Entwurfsmuster

- **Vorsicht!**
 - Entwurfsmuster können eine Architektur auch **unnötig kompliziert** machen.
 - Erst verstehen, wozu ein Muster gut ist, dann das Muster verwenden.
 - **Nicht**: Wie kann ich bloß dieses Muster verwenden?
- Siehe auch Anti-Pattern **Cargo Cult Programming**.
- Beliebttes Beispiel: **Singleton**.

Singleton



```

public class Singleton {
    private static Singleton sInstance;

    public static Singleton Default {
        get {
            if (sInstance == null) {
                sInstance = new Singleton();
            }
            return sInstance;
        }
    }

    private Singleton() { /* ... */ }

    public bool MethodA() { /* ... */ }
}
  
```

Nachteile des Singleton-Patterns

- **Versteckt Abhängigkeiten.**
 - Aufrufe von Methoden eines Singletons sind Aufrufe eines statischen Objekts.
 - Man kann sie überall verwenden.
- **Schwierig zu testen.**
 - Wie verändere ich alle Aufrufe zu einem Singleton in meinen Tests?
- **Schwierig abzuleiten.**
 - Da die Initialisierung statisch geschieht, kann man sie in abgeleiteten Klassen nicht einfach überschreiben.
- **Sprachabhängig.**
 - Z.B. in Java gibt es nicht einen statischen Kontext pro VM, sondern pro Classloader.
- **Schwierig zu verändern.**
 - Was ist, wenn man doch plötzlich zwei Objekte braucht?



SOFTWAREMETRIKEN

Softwaremetriken

- **Metriken** sind z.B. Länge, Fläche, Gewicht, Geschwindigkeit.
 - Gemessen z.B. in Einheiten wie cm, km, m², kg, km/h, ...
- Eine **Softwaremetrik** ist eine Funktion, die eine **Eigenschaft einer Software** auf eine **Zahl** abbildet.
- Softwaremetriken werden benutzt zum
 - **Vergleichen**
 - **Bewerten**
- Verschiedene **Tools** berechnen Metriken
 - Sonar, NDepend, Visual Studio, ...

Lines of Code

- Anzahl an Zeilen einer Klasse bzw. des ganzen Projektes.
- **Logisch**
Nur der eigentliche Code wird gezählt.
- **Physikalisch**
Alle Zeilen (inkl. leere Zeilen, Kommentare, etc.).

Cyclomatic Complexity (CC)

- CC bestimmt, wie komplex eine Methode oder Klasse ist.
- **Methode**
Anzahl der Ausführungspfade (`if`, `while`, `switch`, ...)
- **Klassen**
Durchschnittliche Komplexität aller Methoden oder Summe der Komplexität aller Methoden (je nach Tool).
- **Schwellenwert**
Methoden mit $CC < 15$ sind kompliziert, bis $CC < 30$ aber ok

Lack of cohesion of methods (LCOM)

- Beschreibt die **Kohäsion** einer **Klasse**
- **Berechnung**

$$\text{LCOM3}(c) = \frac{m(c) - \left(\frac{\sum m_f(c)}{f(c)} \right)}{m(c) \cdot f(c)}$$

$m(c)$:= Anzahl an Methoden in c

$f(c)$:= Anzahl an Feldern in c

$m_f(c)$:= Anzahl an Methoden in c die auf das Feld f in c zugreifen

- **Schwellenwert**
 - LCOM3 = 0 bedeutet maximale Kohäsion
 - LCOM3 = 1 bedeutet keine Kohäsion
 - LCOM3 > 1 bedeutet Dead Code oder Variablen, die nur von außen benutzt werden.
 - LCOM3 < 1 ok

Lack of cohesion in methods (LCOM)

```
class Class1
```

```
{
```

```
    private int mFieldA;
```

```
    private int mFieldB;
```

```
    void MethodA()
```

```
{
```

```
        mFieldA = 1;
```

```
}
```

```
    void MethodB()
```

```
{
```

```
        mFieldB = 2;
```

```
}
```

```
}
```

- $m(C1) = 1$
- $f(C1) = 1$
- $m_{m_{FA}}(C1) = 1$
- $m_{m_{FB}}(C1) = 1$
- $LCOM3(C1)=1$

Lack of cohesion in methods (LCOM)

```
class Class1
{
    private int mFieldA;

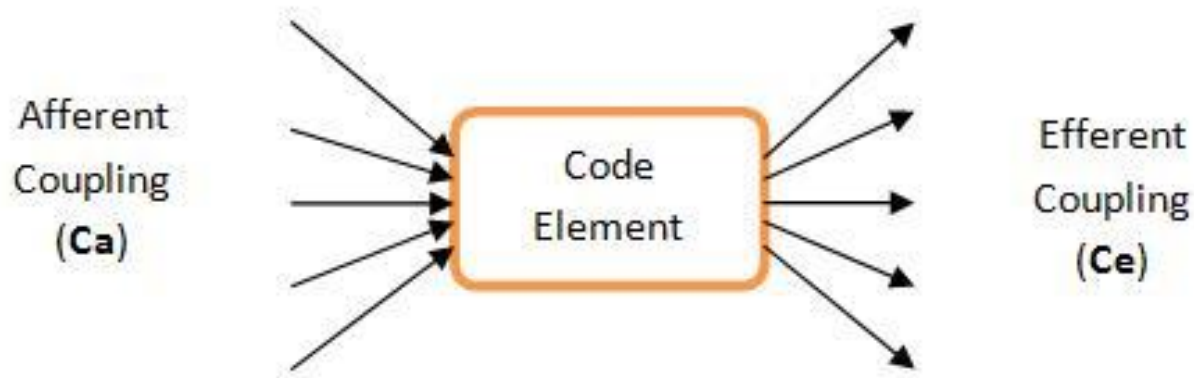
    void MethodA()
    {
        mFieldA = 1;
    }
}
```

```
class Class2
{
    private int mFieldB;

    void MethodB()
    {
        mFieldB = 2;
    }
}
```

Coupling

- Beschreibt wie viele **Typen** von einer Klasse (oder Modul, Namespace, Assembly) **abhängen** oder umgekehrt.



Softwaremetriken

- Metriken geben **Hinweise**.
- Metriken sagen **nichts** über Funktionalität und Qualität zur Laufzeit aus.
- Metriken benötigen **Kontext**.
- Metriken können **Details verschleiern**.
- **Erreichen einer bestimmten Metrik ist kein Ziel!**



FRAGEN?

Quellen

- [Thiemann, 2013, SWT] Thiemann, P. (18.4.2013). Softwaretechnik. Vorlesung. Prozessmodelle. Universität Freiburg.
- [Westphal, 2012/13, UML] Westphal, B. (23.10.2012). Software Design, Modelling, and Analysis in UML. Vorlesung. Introduction. Universität Freiburg.
- [Balzert, 2011] Balzert, H. (2011). Lehrbuch Der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb. Springer. ISBN 3827422469, 9783827422460.
- [Glinz, 2008] Glinz, M. (2008). Modellierung in der Lehre an Hochschulen: Thesen und Erfahrungen. Informatik Spektrum, 31(5):425–434.
- [Gregory, 2009] Gregory, J. (2009). Game Engine Architecture. A K Peters Limited. ISBN 1568814135, 9781568814131.
- [Starke, 2011] Starke, G. (2011). Effektive Softwarearchitekturen: Ein praktischer Leitfaden. Hanser Fachbuch. ISBN 3446427287, 9783446427280.