

# Grundlagen der Softwarearchitektur

*Vincent Langenfeld*, Daniel Dietsch

November 23, 2020

- ▶ *Sopra-bot* ist online
- ▶ Benutzen sie *@Tutorname* oder *@Dozentenname* in Gitea
- ▶ Wir sind Donnerstags 14-18 garantiert für Fragen da (bzw. nach der Fragestunde); entweder via *@Name* oder einfach in Global
- ▶ Hausaufgabenpunkte: im Gruppentreffen und sobald verfügbar im Dashboard
- ▶ GDD Abgabe **diesen Samstag (23:59)**
  - ▶ Im Repository in Abgabeverzeichnis (siehe Wiki)
  - ▶ Feedback innerhalb der nächsten Woche (inhaltlich, handwerklich, Anforderungen erfüllt, zu viel/wenig)
- ▶ Architekturabgabe: Erklärung nächste Vorlesung

- ▶ Was ist Softwarearchitektur?
- ▶ Dokumentieren mit UML
- ▶ Softwarearchitektur bewerten
- ▶ Softwarearchitektur planen
- ▶ Metriken
- ▶ Code Review

## Was ist Softwarearchitektur?

## Definition: Softwarearchitektur <sup>1</sup>

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

---

<sup>1</sup> Helmut Balzert, *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*, 2011, ISBN 978-3-8274-1706-0

## Definition: Modell<sup>2</sup>

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

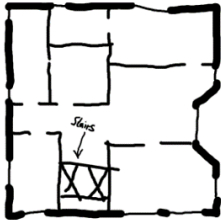
Drei Eigenschaften sind kennzeichnend für ein Modell:

- 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
- iii Die *Pragmatik*, d.h. das *Modell* wurde in einem spezifischen Kontext für einen spezifischen Zweck erstellt

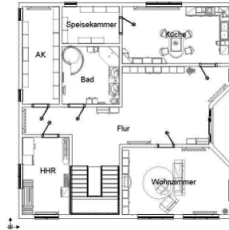
---

<sup>2</sup>Glinz, 2008

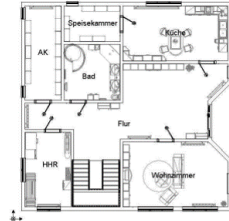
Als Skizze



Als Blaupause



Als Programmiersprache



- + wiringplan
- + windows
- + ...

## Wieso Softwarearchitektur modellieren?

- ▶ *Kommunikation* und *Dokumentation*
  - ▶ Der Realisierung
  - ▶ Von Designentscheidungen
- ▶ *Qualität* planen
  - ▶ Konzeptionelle Integrität (gleiche Probleme werden gleich gelöst)
  - ▶ Grundlage für Bewertung anhand von Szenarien
  - ▶ Wiederverwendung von Systembestandteilen
- ▶ Arbeitsteilung durch *Dekomposition*
  - ▶ Schnittstellen identifizieren und definieren
  - ▶ Durch Abstraktion parallelisieren



## Bewerten einer Softwarearchitektur

## 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

## 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

- GDD
- Techdemo
- Testing

## 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

- GDD
- Techdemo
- Testing

- Cognitive Walkthrough
- Heuristic Evaluation

## 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

- GDD
- Techdemo
- Testing

- Cognitive Walkthrough
- Heuristic Evaluation

- Profiling
- Testing (mit FPS-Anzeige)
- Szenarien

## 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

- GDD
- Techdemo
- Testing

- Cognitive Walkthrough
- Heuristic Evaluation

- Profiling
- Testing (mit FPS-Anzeige)
- Szenarien

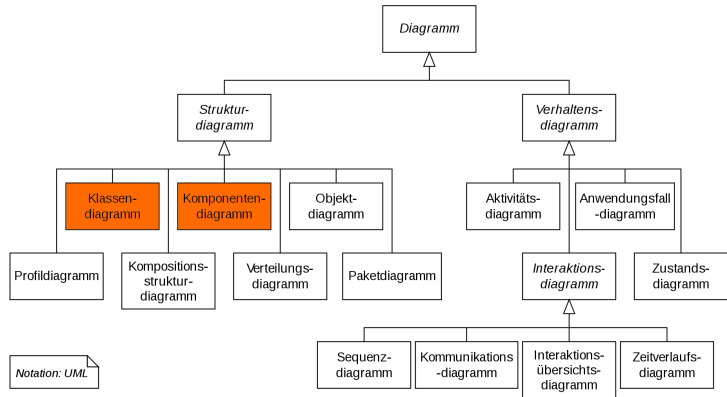
- Szenarien
- Clean Code Prinzipien
- Metriken z.B.: Efferent/Afferent Coupling, Lack of Cohesion of Methods, Type Complexity

# UML

- ▶ Unified Modelling Language
- ▶ Modellierungssprache mit graphischer Notation
- ▶ Standardisiert, verwaltet von der *Object Management Group*



- Unified Modelling Language
- Modellierungssprache mit graphischer Notation
- Standardisiert, verwaltet von der *Object Management Group*



- ▶ *Statische* Sicht
- ▶ *Klassen*, *Interface*, *Pakete* und deren *statische Beziehung*
- ▶ Sehr nahe an der Implementierung

Potion
+ mName: string + Usable: bool - mLevel: int
+ Potion() + Use(...)

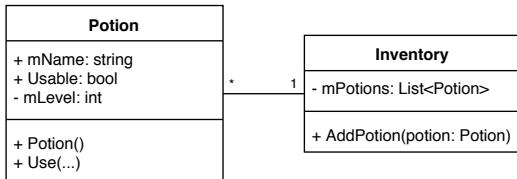
---

```
class Potion
{
    public string mName;
    public bool Usable
    {
        get { return mLevel > 0; }
    }
    //liquid level in percent
    private int mLevel = 100;

    public Potion() { /*...*/ }

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

---

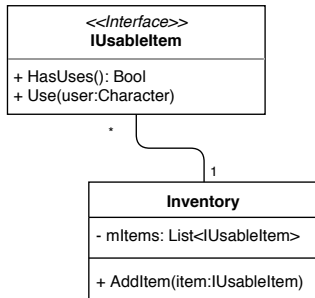


```
class Inventory
{
    private List<Potion> mPotions;

    /* ... */
}
```

```
class Potion
{
    public string mName;
    public bool Usable
    {
        get { return mLevel > 0; }
    }
    //liquid level in percent
    private int mLevel = 100;

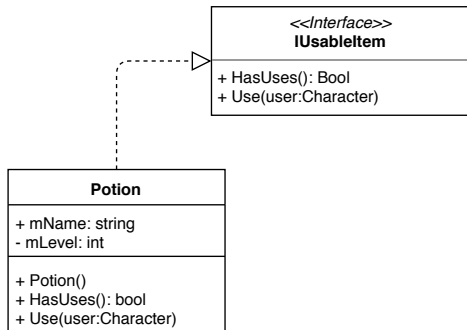
    /* ... */
}
```



```
interface IUsableItem
{
    bool HasUses();
    void Use(Character user);
}
```

```
class Inventory
{
    private List<IUsableItem>
        mInventoryItems;

    /* ... */
}
```



```

interface IUsableItem
{
    bool HasUses();
    void Use(Character user);
}
    
```

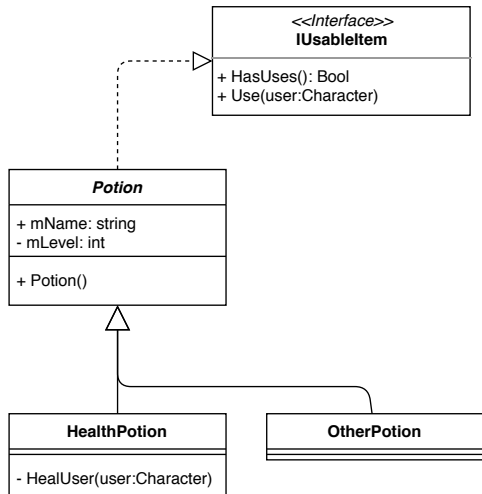
```

class Potion: IUsableItem
{
    public string mName;
    private int mLevel = 100;

    public Potion(/*...*/) { /*...*/ }

    public bool HasUses() { /*...*/ }

    public void Use(Character user)
    { /*...*/ }
}
    
```



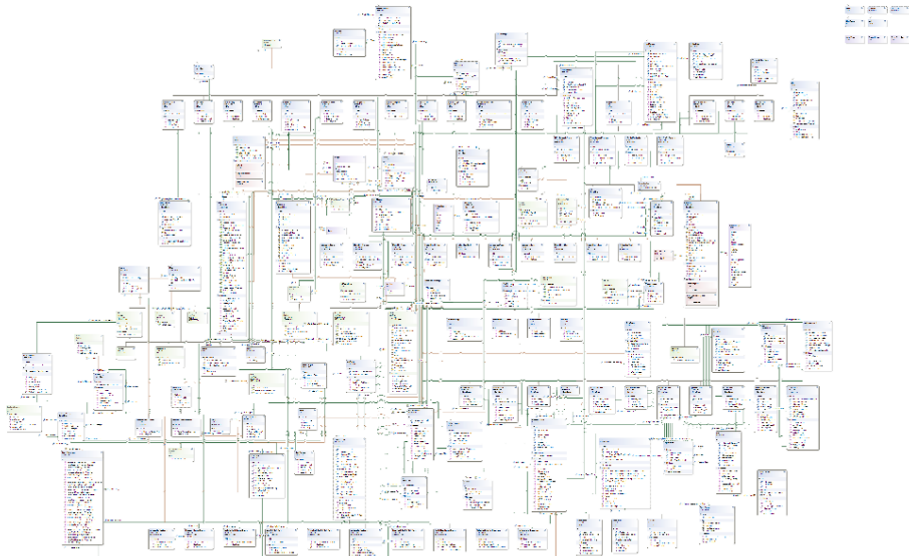
```
abstract class Potion: IUsableItem
{
    public string mName;
    private int mLevel = 100;

    public Potion(/*...*/) { /*...*/ }

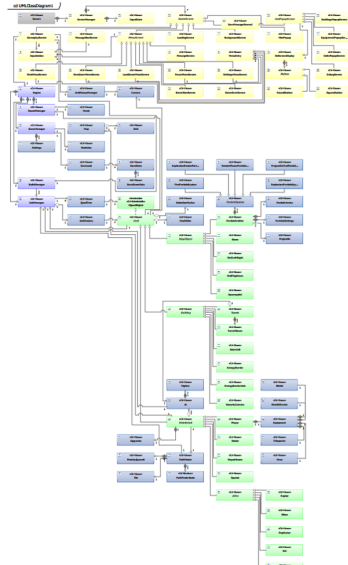
    public bool HasUses() { /*...*/ }

    public void Use(Character user)
    { /*...*/ }
}
```

```
class HealthPotion : Potion
{
    private void
        HealUser(/*...*/){/*...*/}
}
```





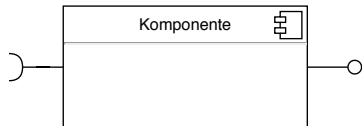


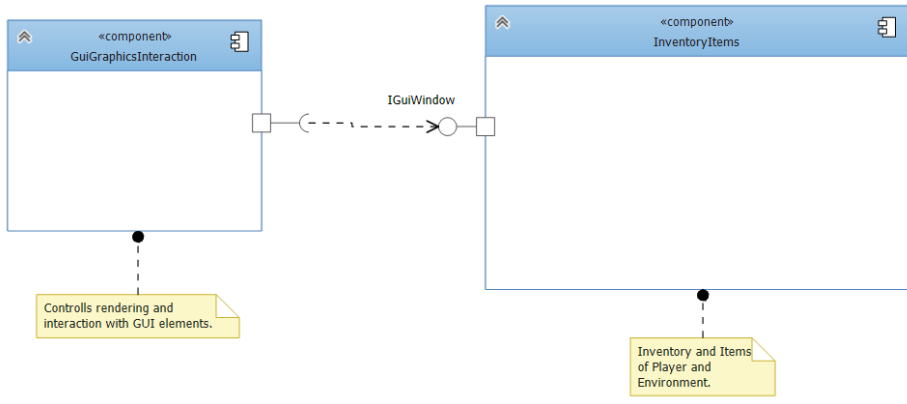
- ▶ *Statische* Sicht
- ▶ *Komponenten*, *Interfaces*, *Parts* und deren *statische Beziehung* als Bausteine
- ▶ Abstrakter als Klassendiagramm
  - ▶ Beschreibt die Schnittstellen zwischen den Komponenten

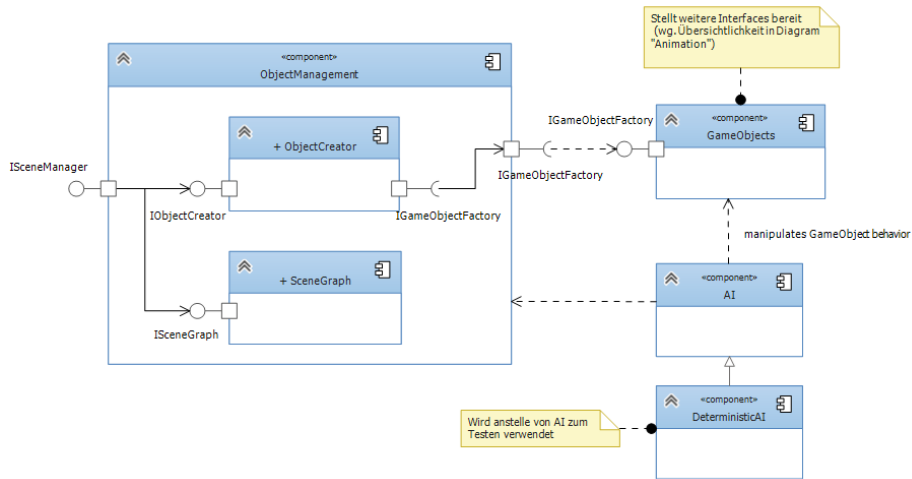
## Komponente

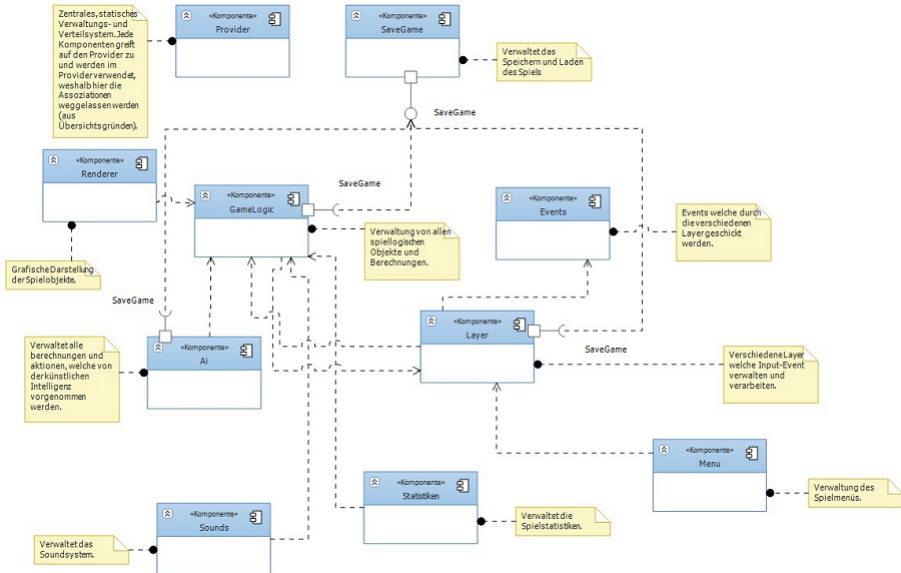
Eine Komponente ist ein *Softwarebaustein*, der Dritten Funktionalität über Schnittstellen zur Verfügung stellt und nur explizite Abhängigkeiten nach außen besitzt.

- ▶ Eine Komponente enthält z.B.: Klassen, Interfaces, Enumerations, ...
- ▶ Kann eine Schnittstelle anbieten
- ▶ Und Schnittstellen fordern









Wie plane ich eine Architektur?

Es gibt kein deterministisches Verfahren, das auf jeden Fall in einer guten Architektur resultiert.\*

\* Es gibt grundlegende  
Aktivitäten und  
Heuristiken, etc.



Informationen über das *Problem* sammeln

- ▶ Domänenwissen (Bücher, Webseiten, Zeitschriften, nächste Vorlesung)
- ▶ Fachbegriffe sammeln
  - ▶ Kernaufgabe des Systems in wenigen Sätzen beschreiben
  - ▶ Gemeinsame Sprache schaffen
- ▶ Anforderungen
- ▶ Rahmenbedingungen und technischer Kontext
- ▶ GDD

Informationen über die *Lösung* sammeln

- ▶ Gibt es schon Lösungen für ähnliche Aufgaben?
- ▶ Gibt es Lösungen für Teilaufgaben
  - ▶ Architekturstile
  - ▶ Entwurfsmuster
- ▶ Quellen: Literatur, Internet, eigene Projekte,...

Arbeiten Sie *iterativ* und *inkrementell*

- ▶ Einfache Lösungen bauen und weiterentwickeln
- ▶ Frühzeitig validieren und dann erweitern
  - ▶ User Stories
  - ▶ Szenarien
- ▶ Vorsicht vor Optimierung

## Szenario

Ein Szenario ist eine Ablaufbeschreibung mit den folgenden Eigenschaften

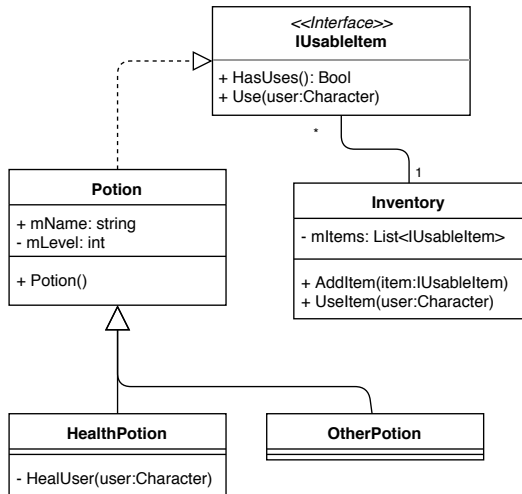
- ▶ Quelle (z.B. intern, extern, Benutzer, Betreiber, Angreifer)
- ▶ Auslöser (z.B. Modellierer, Spieler)
- ▶ Umgebung (z.B. Endlosmodus, Entwicklung)
- ▶ Systembestandteil (z.B. alle, Input, KI)
- ▶ Antwort (z.B. Modell erfolgreich eingebunden, gleichzeitige Darstellung, 45FPS)
- ▶ Antwortmetrik (z.B. mehr oder weniger als 45FPS)

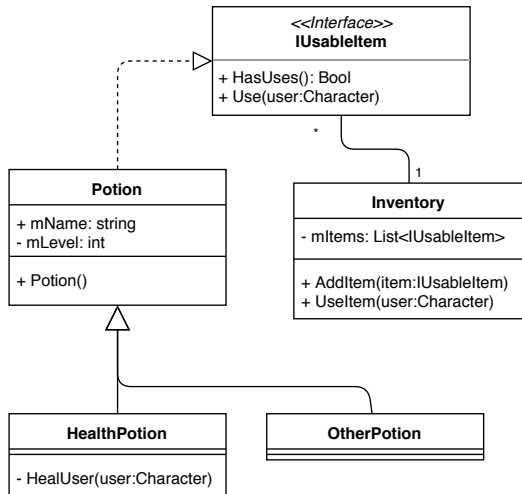
### Modell Einbinden

Die Erstellung und Einbindung eines neuen 3D-Modells durch einen Modellierer muss innerhalb von 6h abgeschlossen sein.

### Aktive Spielobjekte

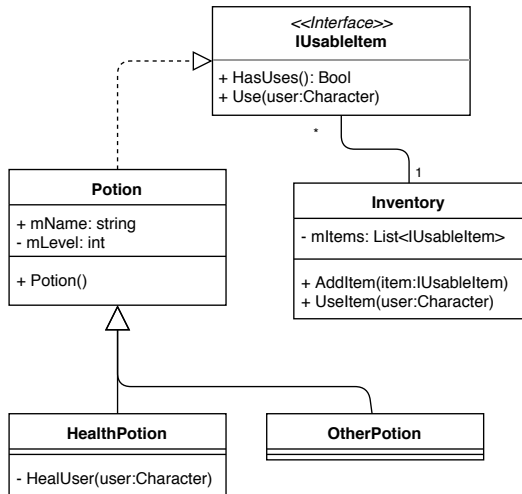
Das Spiel muss im Endlosmodus 1000 aktive, durch den Spieler steuerbare Spielobjekte gleichzeitig mit mindestens 45 FPS auf der Referenzhardware darstellen können.





## Szenario: Heiltrank heilt Spieler

Wenn der Spieler im Inventar einen Heiltrank benutzt, wird die Spielfigur geheilt.



## Szenario: Heiltrank heilt Spieler

Wenn der Spieler im Inventar einen Heiltrank benutzt, wird die Spielfigur geheilt.

## Szenario: Namen anzeigen

Im Inventar sollen die Namen aller Items in sortierter Reihenfolge angezeigt werden.

*Clean Code* gibt Hinweise

- ▶ Viele Clean Code Prinzipien beziehen sich auf die Architektur

Grundidee

- ▶ Abhängigkeit zwischen Bausteinen verringern:  
Niedrige *Kopplung*, hohe *Kohäsion*
- ▶ Open-Closed Principle  
Offen für Erweiterungen, geschlossen gegenüber Änderungen





## Single Responsibility Principle <sup>2</sup>

A class should have only one reason to change

```
class Potion: IUsableItem
{
    public string mName;
    private int mLevel = 100;

    public Potion(/*...*/) { /*...*/ }

    public void DrawInInventory
        (/*...*/) { /*...*/ }

    public bool HasUses() { /*...*/ }

    public void Use(/*...*/) { /*...*/ }

    public bool SaveToFile
        (Path targetFile) { /*...*/ }
}
```

<sup>2</sup>Robert C. Martin, *Clean Architecture*, 2018, Prentice Hall

## Single Responsibility Principle <sup>2</sup>

A class should have only one reason to change

Class `Potion` muss geändert werden, wenn:

- ▶ Sich das Verhalten von Potions ändert
- ▶ Sich ändert wie der Inventar gezeichnet wird
- ▶ Sich das Speichern/Laden ändert
- ▶ Sich Dateisystemdetails ändern
- ▶ (Sich `IUsableItem` ändert)

```
class Potion: IUsableItem
{
    public string mName;
    private int mLevel = 100;

    public Potion(/*...*/) { /*...*/ }

    public void DrawInInventory
        (/*...*/) { /*...*/ }

    public bool HasUses() { /*...*/ }

    public void Use(/*...*/) { /*...*/ }

    public bool SaveToFile
        (Path targetFile) { /*...*/ }
}
```

<sup>2</sup>Robert C. Martin, *Clean Architecture*, 2018, Prentice Hall

# Entwurfsmuster

Nächstes Jahr in der Softwaretechnikvorlesung *Design Patterns*

- ▶ Nützlich für Spiele:  
Composite, (Abstract) Factory, Builder, Flyweight, Observer, Visitor, Iterator
- ▶ Vielleicht nützlich:  
Object Pool, Proxy, Prototype, Decorator, Command, Strategy, ...

- ▶ <https://gameprogrammingpatterns.com/contents.html>
- ▶ [https://en.wikipedia.org/wiki/Category:Software\\_design\\_patterns](https://en.wikipedia.org/wiki/Category:Software_design_patterns)

## 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?

Singleton
+ Default: Singleton
+ MethodA():Boolean - 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

- ▶ Versteckte Abhängigkeiten
  - ▶ Aufrufe von Methoden eines Singleton sind Aufrufe eines statischen Objekts.
  - ▶ Überall verwendbar
- ▶ Schwierig zu testen
  - ▶ Wie tausche ich das Singleton gegen ein Mockup aus?
- ▶ 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, wenn ich plötzlich zwei Objekte brauche?



## Metriken

## Metrik<sup>3</sup>

A quantitative measure of the degree to which a system, component, or process possesses a given attribute.

- ▶ Gemessen in z.B.: in Einheiten wie cm, km, kg, km/h, ...
- ▶ *Softwaremetrik* ist eine Funktion, die eine Eigenschaft der Software auf eine Zahl abbildet
- ▶ Vergleichen, Bewerten
- ▶ Verschiedene Tools wie Sonar, NDepend, Visual Studio

---

<sup>3</sup>IEEE Std. 610.20 (1990)

## Lines Of Code

Anzahl an Zeilen einer Klasse bzw. des ganzen Projektes

- ▶ *Logisch:*  
Nur tatsächliche Codezeilen (keine  
Kommentare, Leerzeilen, ...)
- ▶ *Physikalisch:*  
Alle Zeilen

## Lines Of Code

Anzahl an Zeilen einer Klasse bzw. des ganzen Projektes

- ▶ *Logisch*:  
Nur tatsächliche Codezeilen (keine  
Kommentare, Leerzeilen, ...)
- ▶ *Physikalisch*:  
Alle Zeilen

*Grundsätzlich*: Jede Methode sollte  
vollständig auf einen Bildschirm  
passen

## Cyclomatic Complexity (CC)

Die Cyclomatic Complexity ist die Anzahl der Ausführungspfade innerhalb einer Methode

- ▶ Methoden: Kanten - Knoten + 2
- ▶ Klassen: je nach Tool
  - ▶ Durchschnitt aller Methoden
  - ▶ Summe aller Methoden
- ▶  $15 < CC < 30$ : kompliziert aber ok

```
class Class1
{
    private const SOMECONST = 5;

    void MethodA()
    {
        if (mFieldA == SOMECONST)
        {
            DoSomething();
        }
    }
}
```

## Lack of Cohesion of Methods (LCOM)

### Kohäsion einer Klasse

- ▶  $LCOM = 0$ : Klasse hat keine Methoden
- ▶  $LCOM = 1$ : Klasse ist zusammenhängend
- ▶  $LCOM > 1$ : Klasse kann geteilt werden
  - ▶ Durchschnitt aller Methoden
  - ▶ Summe aller Methoden

```
class Class1
{
    private int mFieldA;
    private int mFieldB;

    void MethodA()
    {
        mFieldA = 1;
    }

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

## Lack of Cohesion of Methods (LCOM)

### Kohäsion einer Klasse

- ▶  $LCOM = 0$ : Klasse hat keine Methoden
- ▶  $LCOM = 1$ : Klasse ist zusammenhängend
- ▶  $LCOM > 1$ : Klasse kann geteilt werden
  - ▶ Durchschnitt aller Methoden
  - ▶ Summe aller Methoden

```
class Class1
{
    private int mFieldA;
    private int mFieldB;

    void MethodA()
    {
        mFieldA = 1;
    }

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

## Coupling (Afferent)

Zahl der Klassen, die von dieser Klasse abhängig sind

- ▶ Nicht schlimm wenn die Klasse stabil ist (z.B.: standard Libraries)
- ▶ Hohes Coupling: Änderungen an dieser Klasse verursachen Änderungen an viele anderen Stellen

## Coupling (Efferent)

Zahl der Klassen von denen die Klasse abhängig ist

- ▶ Hinweise auf eine *Gottklasse* (macht alles und muss dafür jeden kennen)



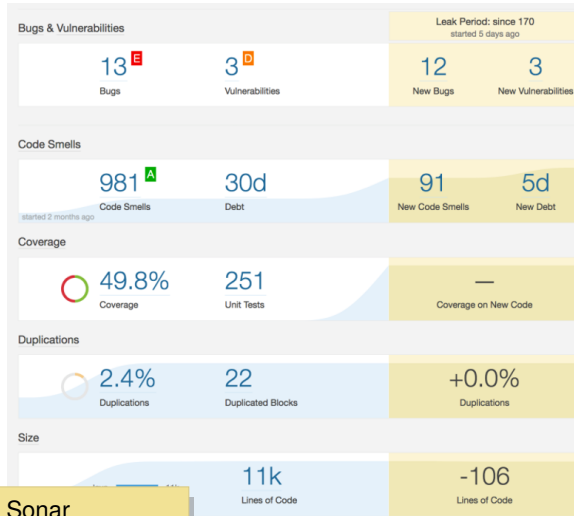


Codemetrikergebnisse

Filter: Keine    Min.:    Max.:    [Icons]

Hierarchie ▲	Wartbarkeitsindex	Zyklomatische Komp...	Vererbungstiefe	Klassenkopplung	Zeilen von Quellcode	Zeilen von ausführbarem...
▲ ■■ Irrgarten (Debug)	84	168	3	86	1.344	398
▷ {} Irrgarten	77	33	3	47	323	80
▷ {} Irrgarten.Componentes	88	43	2	29	283	53
▷ {} Irrgarten.Rendering	75	30	2	22	195	68
▲ {} Irrgarten.Scenegraph	93	39	2	16	216	48
▷ 🍄 Component	99	5	1	5	18	2
▷ 🍄 ComponenteSystem	85	8	1	4	41	12
▷ 🍄 GameObject	82	17	1	11	95	29
▷ 🍄 GenericUpdateSystem	94	4	1	4	22	4
▷ 🔗 ISystem	100	3	0	2	6	0
▷ 🔗 Item	100	1	0	0	4	0
▷ 🍄 WorldZero	95	1	2	2	8	1
▷ {} Irrgarten.Screen	94	12	2	9	90	16
▷ {} Irrgarten.Serialization	75	1	1	5	16	4
▷ {} Irrgarten.Utils	70	10	1	18	221	129

VS → Analyse → Calculate Metrics



services.sopranium.de → Sonar

- ▶ Metriken geben *Hinweise*
- ▶ Metriken sagen nichts über die *Funktionalität* zur Laufzeit aus
- ▶ Metriken benötigen *Kontext*
- ▶ Metriken verschleiern *Details*
- ▶ Das Erreichen einer Metrik ist **kein Ziel**

## Recurring Tasks

## Qualitätssicherung (ab Woche 3)

Code Reviews machen um die Codequalität zu verbessern. Ergebnisse im Gruppentreffen (Sprint Review) präsentieren.

Vorgehen:

- ▶ Ein interessantes Stück Code suchen
- ▶ **Code lesen und verstehen**
- ▶ Verständnisprobleme und gefundene Fehler festhalten
- ▶ Verbesserungen vornehmen
- ▶ Probleme berichten

## Qualitätssicherung (ab Woche 3)

Code Reviews machen um die Codequalität zu verbessern. Ergebnisse im Gruppentreffen (Sprint Review) präsentieren.

Vorgehen:

- ▶ Ein interessantes Stück Code suchen
- ▶ **Code lesen und verstehen**
- ▶ Verständnisprobleme und gefundene Fehler festhalten
- ▶ Verbesserungen vornehmen
- ▶ Probleme berichten

Ein *interessantes* Stück Code suchen:

1. Autor hat um Review gebeten und interessanten Code verfasst
2. Code ist im Sprinttreffen negativ aufgefallen
3. Sonar markiert den Code als komplex oder schwer wartbar
4. Reviewer interessiert sich für die Funktionalität (z.B.: Quadtree, A\*, ...)

## Qualitätssicherung (ab Woche 3)

Code Reviews machen um die Codequalität zu verbessern. Ergebnisse im Gruppentreffen (Sprint Review) präsentieren.

Vorgehen:

- ▶ Ein interessantes Stück Code suchen
- ▶ **Code lesen und verstehen**
- ▶ Verständnisprobleme und gefundene Fehler festhalten
- ▶ Verbesserungen vornehmen
- ▶ Probleme berichten

### *Pfadfinderregel*

Verlasse den Code immer besser  
als Du ihn vorgefunden hast.

Ein *interessantes* Stück Code suchen:

1. Autor hat um Review gebeten und interessanten Code verfasst
2. Code ist im Sprinttreffen negativ aufgefallen
3. Sonar markiert den Code als komplex oder schwer wartbar
4. Reviewer interessiert sich für die Funktionalität (z.B.: Quadtree, A\*, ...)

## Checkliste:

- ▶ Kommentare, Commitmessages und Naming



## Checkliste:

- ▶ **Kommentare**, Commitmessages und Naming

---

```
//recognises sopra estimate labels  
re.compile("est(imate)*\s*:\s*(  
?P<estimate>[0-9]+((.|\,)[0-9]+){0,1})\s*")
```

---

## Checkliste:

- Kommentare, Commitmessages und Naming

---

```
//recognises sopra estimate labels
re.compile("est(imate)*\s*:\s*(
?P<estimate>[0-9]+((.|,)[0-9]+){0,1})).*")
```

---

---

```
> git blame InterestingCode.cs
a3b62bf (Tina Test [...] 1) using [...];
959f313 (Paul Probe [...] 2) [...]
[...]
959f313 (Paul Probe [...] 147) [...]
> git log -p a3b62bf1
[...] Author Tina Teststudent [...]
    Woololo
```

---

## Checkliste:

- Kommentare, Commitmessages und **Naming**

---

```
//recognises sopra estimate labels  
re.compile("est(imate)*\\s*:\\s*(  
?P<estimate>[0-9]+((\\.|,)[0-9]+){0,1}).*")
```

---

---

```
> git blame InterestingCode.cs  
a3b62bf (Tina Test [...] 1) using [...];  
959f313 (Paul Probe [...] 2) [...]  
[...]  
959f313 (Paul Probe [...] 147) [...]  
> git log -p a3b62bf1  
[...] Author Tina Teststudent [...]  
    Woololo
```

---

---

```
//integer of the index  
private int mTheIndex;  
//Max Tmp in Fl.  
private int mMaxTmpInFl;  
private const int DamnNumber = 42;  
protected void GetLost(GameObject o)
```

---

## Checkliste:

- ▶ Kommentare, Commitmessages und Naming
- ▶ **Codestyle und Struktur**
  - ▶ Restriktive Zugriffsmodifikatoren
  - ▶ Konstanten nicht im Quellcode (in Variablen)
  - ▶ Konsistente Benennungen (Get, Set, ...)
  - ▶ Klassenname im Singular
  - ▶ Mengen von Objekten im Plural
  - ▶ Methoden als Verb + Substantiv

---

```
class GameWorld{

    private const int MAX_LAYERS = 5;
    private WorldLayer[] worldLayers;

    public GameWorld(int layerCount)
        /*...*/

    public void SetLayer
        (WorldLayer layer) /*...*/

    public WorldLayer GetLayer
        (int layerNumber) /*...*/

    public List< /*...*/>
        GetGameObjects() /*...*/

}
```

---

## Checkliste:

- ▶ Kommentare, Commitmessages und Naming
- ▶ Codestyle und Struktur
- ▶ **Funktionalität**
  - ▶ Keine offensichtlichen Fehler
  - ▶ Sinnvolle Implementierung
  - ▶ Mögliche Vereinfachungen

---

```
protected bool PreviousResultExists()  
{  
    try  
    {  
        int resultId =  
            Results[SelectedResultId - 1];  
        return true;  
    }  
    catch  
    {  
        return false;  
    }  
}
```

---

---

```
protected bool  
    PreviousResultExists() =>  
        SelectedResultId > 0;
```

---

## Checkliste:

- ▶ Kommentare, Commitmessages und Naming
- ▶ Codestyle und Struktur
- ▶ Funktionalität
- ▶ **Clean Code Prinzipien**
  - ▶ z.B.: Single Responsibility Principle
  - ▶ Wiki, Bücher, ...

