Grundlagen der Softwarearchitektur

Daniel Dietsch, *Vincent Langenfeld*, Frank Schüssele

November 3, 2021

Organisatorisches

- Benutzen sie *@Tutorname* oder *@Dozentename* in Gitea und Mattermost
- Wir sind Donnerstags 14-18 (garantiert) für Fragen da (bzw. nach der Fragestunde)
- Hausaufgabenpunkte: im Gruppentreffen und sobald verfügbar im Dashboard
- GDD Abgabe diesen Samstag (23:59)
  - Im Repository in Abgabeverzeichnis (siehe Wiki)
  - Feeback innerhalb der nächsten Woche (inhaltlich, handwerklich, Anforderungen erfüllt, zu viel/wenig)
- Architekturpräsentation: Erklärung nächste Vorlesung

Agenda

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

Was ist Softwarearchitektur?

Softwarearchitektur

Definition: Softwarearchitektur [1]

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

[1] Helmut Balzert, *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*, 2011, ISBN 978-3-8274-1706-0

Definition: Modell[2]

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:

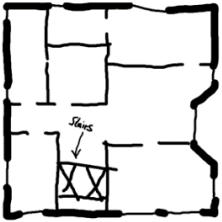
i Das *Abbildungsmerkmal*, d.h. es gibt eine Entität (ein Original) dessen Abbild oder Vorbild das Modell ist

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

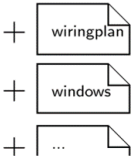
[2] Glinz, 2008

Als Skizze



Als Blaupause

Als Programmiersprache



Softwarearchitektur

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

Als Skizze

Als Blaupause

Qualitätsmerkmale

ISO 9126 (bzw. 25000)

- Funktionalität
  Angemessenheit, *Richtigkeit*, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- Zuverlässigkeit
  Reife, Fehlertoleranz, Widerherstellbarkeit
- *Benutzbarkeit*
  Verständlichkeit, Erlernbarkeit, Bedienbarkeit
- *Effizienz*
  Zeitverhalten, Verbrauchsverhalten
- Änderbarkeit
  Analysierbarkeit, *Modifizierbarkeit*, *Stabilität*, Prüfbarkeit

Qualitätsmerkmale

ISO 9126 (bzw. 25000)

- Funktionalität
  Angemessenheit, *Richtigkeit*, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- Zuverlässigkeit
  Reife, Fehlertoleranz, Widerherstellbarkeit
- *Benutzbarkeit*
  Verständlichkeit, Erlernbarkeit, Bedienbarkeit
- *Effizienz*
  Zeitverhalten, Verbrauchsverhalten
- Änderbarkeit
  Analysierbarkeit, *Modifizierbarkeit*, *Stabilität*, Prüfbarkeit

- GDD
- Techdemo
- Testing

Qualitätsmerkmale

ISO 9126 (bzw. 25000)

- Funktionalität
  Angemessenheit, *Richtigkeit*, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- Zuverlässigkeit
  Reife, Fehlertoleranz, Widerherstellbarkeit
- *Benutzbarkeit*
  Verständlichkeit, Erlernbarkeit, Bedienbarkeit
- *Effizienz*
  Zeitverhalten, Verbrauchsverhalten
- Änderbarkeit
  Analysierbarkeit, *Modifizierbarkeit*, *Stabilität*, Prüfbarkeit

- GDD
- Techdemo
- Testing

- Cognitive Walkthrough
- Heuristic Evaluation

Qualitätsmerkmale

ISO 9126 (bzw. 25000)

- Funktionalität
  Angemessenheit, *Richtigkeit*, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- Zuverlässigkeit
  Reife, Fehlertoleranz, Widerherstellbarkeit
- *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

Qualitätsmerkmale

ISO 9126 (bzw. 25000)

- Funktionalität
  Angemessenheit, *Richtigkeit*, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- Zuverlässigkeit
  Reife, Fehlertoleranz, Widerherstellbarkeit
- *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
- Codemetriken

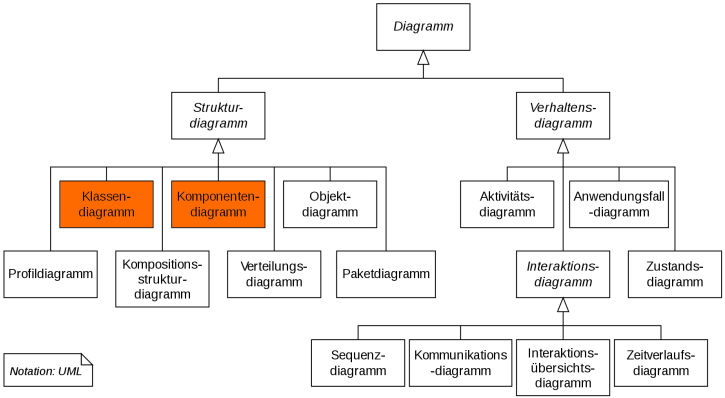
UML

UML

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

UML

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



UML · Klassendiagram

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

UML · Klasse

Potion
+ Name: string + Usable: bool - mLevel: int
+ Potion() + Use(user: Character): bool

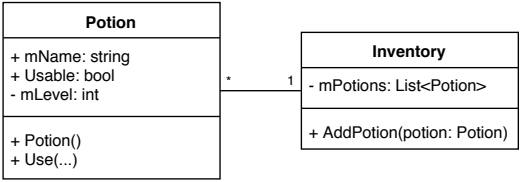
```
class Potion
{
    public string mName;
    public bool Usable
    {
        get { return mLevel > 0; }
    }
    //liquid level in percent
    private int mLevel = 100;


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


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

UML · Klasse

Potion
+ Name: string + Usable: bool - mLevel: int - mEffect: List<Effect>
+ Potion(effects: List<Effect>) + Use(user: Character): bool

```
class Potion
{
    public string mName;
    public bool Usable
    {
        get { return mLevel > 0; }
    }
    //liquid level in percent
    private int mLevel = 100;
    private var mEffects = new List<Effect>();

    public Potion(List<Effect>
        effects) { /*...*/ }

    // apply each effect to User
    public bool Use(Character user) {
        /*...*/ }
}
```

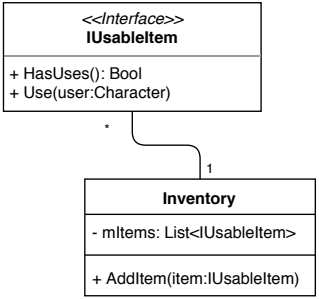
UML · Assoziation




```
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;

   /*  ... */
}
```

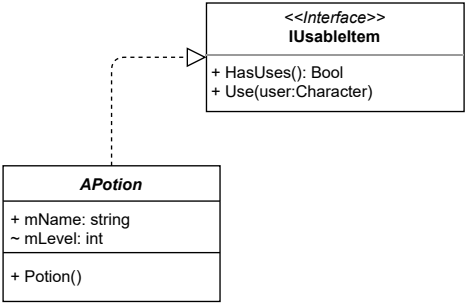
UML · Interface




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

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

    /*  ... */
}
```

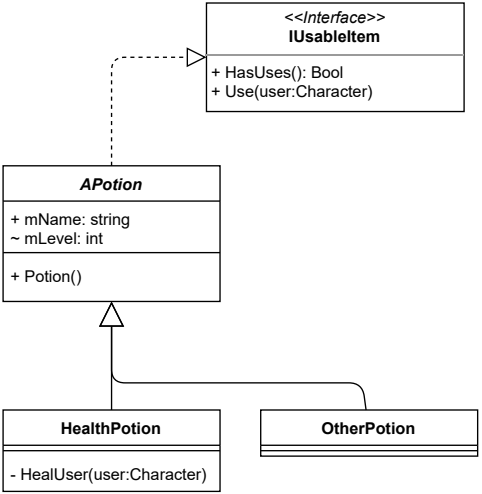
UML · Realisierung




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

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

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

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

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

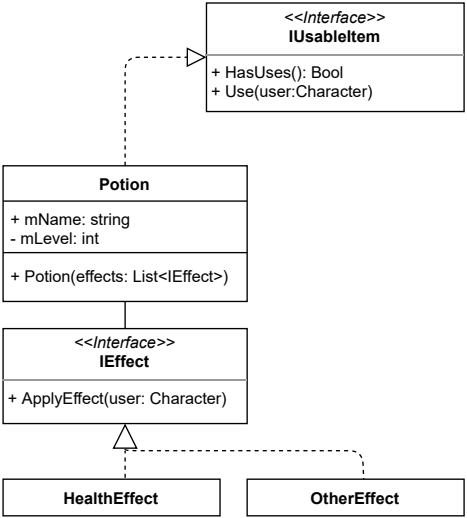
UML · Vererbung




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

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

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

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

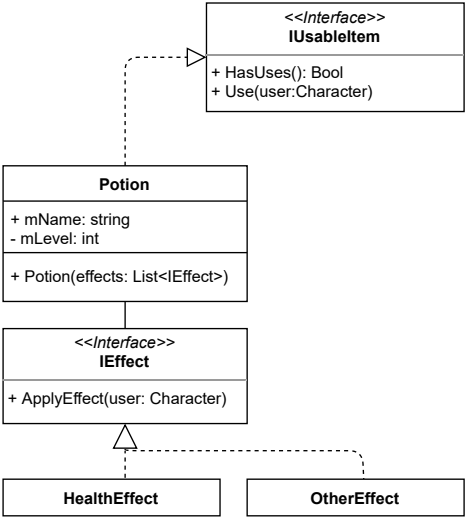
```
class HealthPotion : Potion
{
   private void
       HealUser(/*...*/){/*...*/}
}
```

UML · Das Beispiel ohne Vererbung




```
class Potion
{
    /* ... */
    private var mEffects = new
        List<IEffect>();

    public Potion(List<IEffect> effects)
        { /*...*/ }

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

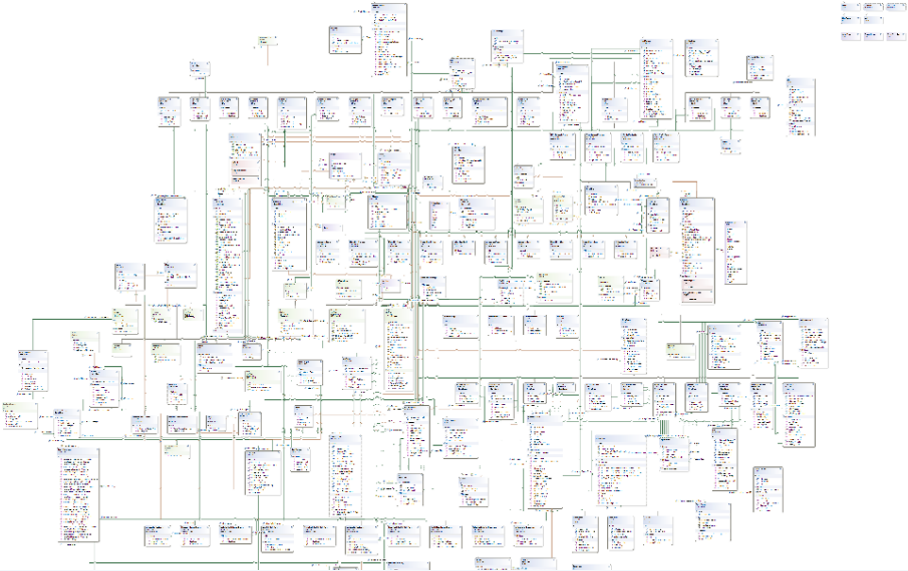
UML · Das Beispiel ohne Vererbung



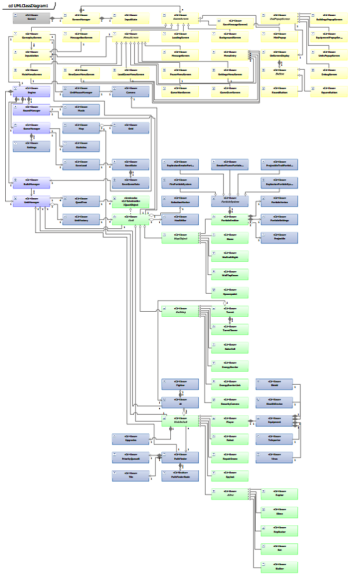

```
class Potion
{
    /* ... */
    private var mEffects = new
        List<IEffect>();

    public Potion(List<IEffect> effects)
        { /*...*/ }

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

```
public static Potion GetHealAndFly(){
    return new Potion(new List<Effect>()
    {new HealEffect(),
       new FlyingEffect()}
    )
}
```

UML



UML



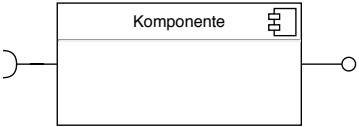
UML · Komponentendiagramm

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

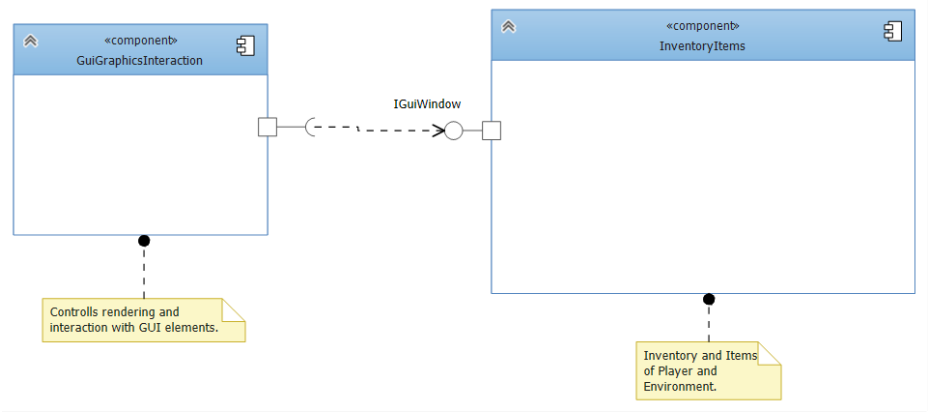
Komponente

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

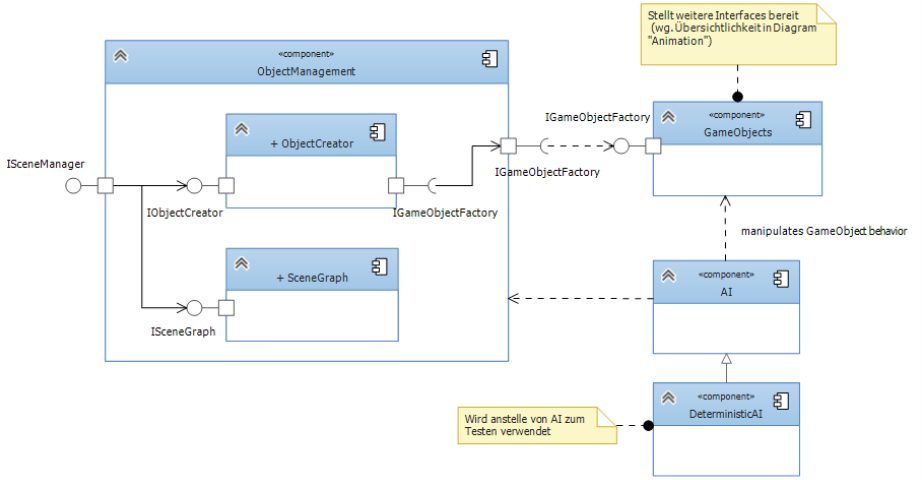
- Eine Komponente enthält z.B.: Klassen, Interfaces, Enumerations, ...
- Kann eine Schnittstelle anbieten
- Und Schnittstellen fordern



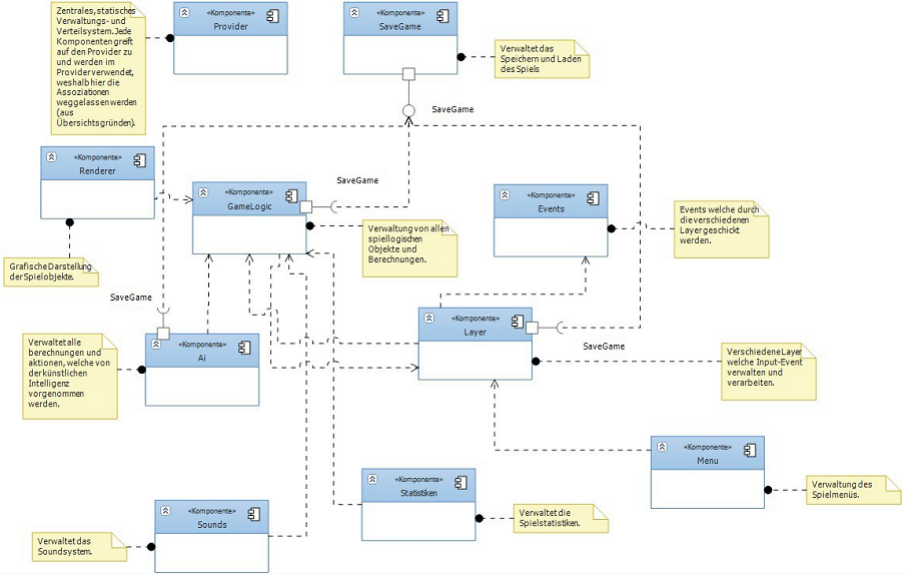
UML · Komponentendiagramm



UML · Komponentendiagramm



UML · Komponentendiagramm



Wie plane ich eine Architektur?

Planung

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

* Es gibt grundlegende Aktivitäten und Heuristiken, etc.

Planung · Das Problem verstehen

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

Planung · Den Lösungsraum verstehen

Informationen über die *Lösung* sammeln

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

Planung · Sich der Lösung annähern

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

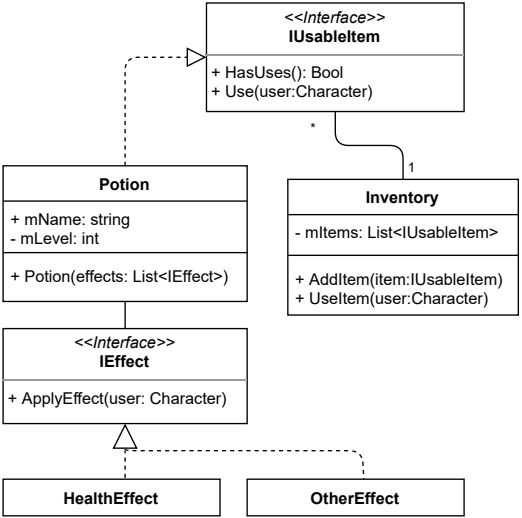
- Quelle
  (z.B. Auftraggeber)
- Auslöser
  (z.B. Modellierer, Spieler, Angreifer)
- Umgebung
  (z.B. im Endlosspielmodus, im Spielverzeichnis)
- Beschreibung
- Antwortmetrik
  (z.B. Modell erfolgreich eingebunden, gleichzeitige Darstellung, 45FPS)

- Quelle: Auftraggeber
- Auslöser: Spieler

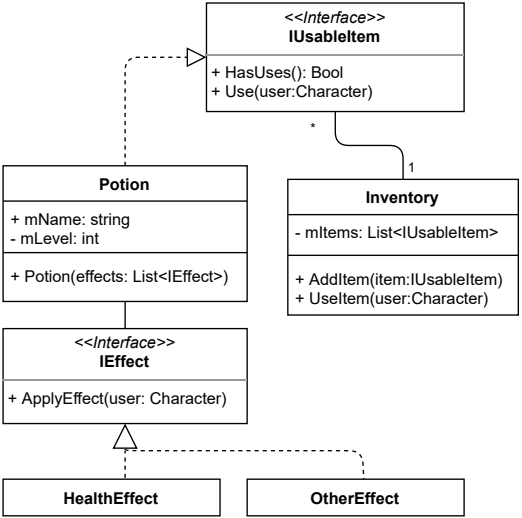
Ich bin bereits im Spiel und selektiere eine Einheit:
Es werden auf sinnvolle weise Spielobjekte erzeugt.

Ich bin in der Techdemo:
Das Spiel kann 1000 aktive, durch den Spieler steuerbare Spielobjekte gleichzeitig mit mindestens 45 FPS auf der Referenzhardware darstellen.

Planung · Beispiel

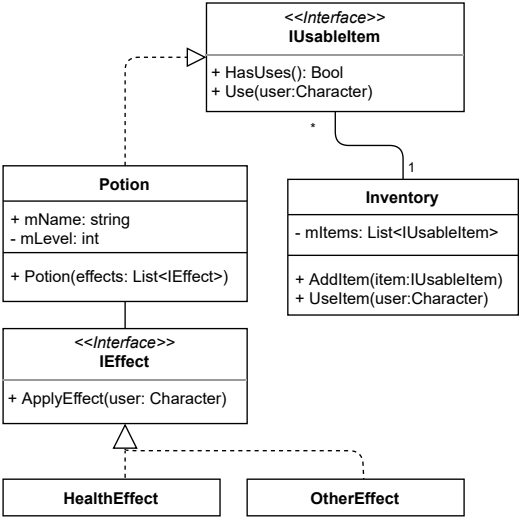


Planung · Beispiel



Ich befinde mich (im Spiel) im Inventarfenster:
Die Namen aller Items werden in sortierter Reihenfolge dargestellt.

Planung · Beispiel

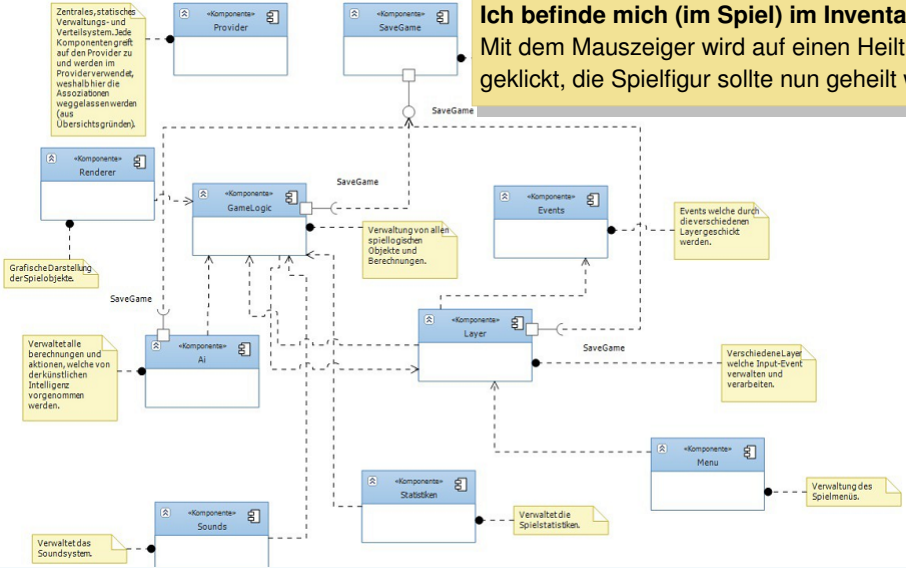


Ich befinde mich (im Spiel) im Inventarfenster: Die Namen aller Items werden in sortierter Reihenfolge dargestellt.

Ich befinde mich (im Spiel) im Inventarfenster: Mit dem Mauszeiger wird auf einen Heiltrank geklickt, die Spielfigur sollte nun geheilt werden.

Planung · Beispiel

Ich befinde mich (im Spiel) im Inventarfenster:
Mit dem Mauszeiger wird auf einen Heiltrank geklickt, die Spielfigur sollte nun geheilt werden.



Planung

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

Planung


verstehen
prinzip
astonishment
klasse
least
principle
arbeiten
andere
anderen
klassen
funktion
möglichst
programmteile
schreiben
muss
haben
sollten
namen
simple
conventions
kiss
verhalten
immer
projekt
development
team
stupid
funktionen
einfacher
genau
viele
viel
leichter
weniger
besser
einfach
mehrere
somit
clean
cleancode
codes
erwartet
finde
yourself
keine
problem
aufgabe
verständlich
verwenden
schnell
welche
sondern
wahrscheinlich
gerade
arbeit
klar
denke
großen
repeat
yagni
gebraucht
programm
nachvollziehen
teilprobleme
coding
möglich
weil
gut
hat
dafür
spiel
separation
zukunft
dabei
don't
keep
the
fürs
oft
teil
dazu

Planung

Single Responsibility Principle [2]

Eine Klasse sollte sich nur aus einem einzigen Grund ändern müssen.

```
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) {/*...*/}

    public static Inventory SortInventory
        (Inventory inventory) {/*...*/}
}
```

[2] Robert C. Martin, *Clean Architecture*, 2018, Prentice Hall

Planung

Single Responsibility Principle [2]

Eine Klasse sollte sich nur aus einem einzigen Grund ändern müssen.

Class `Potion` muss geändert werden, wenn:

- (Sich `IUsableItem` ändert)
- Sich das Verhalten von Potions ändert
- Sich ändert wie der Inventar gezeichnet wird
- Sich das Speichern/Laden ändert
- Sich Dateisystemdetails ändern
- Sich die Inventarsortierung 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) {/*...*/}

    public static Inventory SortInventory
        (Inventory inventory) {/*...*/}
}
```

[2] Robert C. Martin, *Clean Architecture*, 2018, Prentice Hall

Entwurfsmuster

Entwurfsmuster

- 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

Entwurfsmuster

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

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?

- https://gameprogrammingpatterns.com/contents.html
- https://en.wikipedia.org/wiki/Category:Software_design_patterns

Entwurfsmuster · Anitpattern: Singleton

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(){/* ... */}
}
```

UNI FREIBURG

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

Metriken

Metrik[3]

A quantitative measure of the deree 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: Sonar, NDepend, Visual Studio

[3] IEEE Std. 610.20 (1990)

Metriken

Lines Of Code

Anzahl an Zeilen einer Klasse bzw. des ganzen Projektes

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

```
var x = 4 + 5;
```

```
/*
add four and five
*/
// do not write such comments!
var x
=
4
+
5;
```

Metriken

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

```
var x = 4 + 5;
```

```
/*
add four and five
*/
// do not write such comments!
var x
=
4
+
5;
```

Metriken

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 SOMECNOST = 5;

    11 void MethodA()
    {
    12  if (mFieldA == SOMECONST)
        {
        13      DoSomething();
        }
    14  return something;
    }
}
```

Metriken

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 SOMECNOST = 5;

    11  void MethodA()
    {
    12    if (mFieldA == SOMECONST)
          {
    13        DoSomething();
          }
    14    return something;
    }
}
```

Metriken

Lack of Cohesion of Methods (LCOM)

Lack of Cohesion of Methods ist eine Metrik für die Kohäsion innerhalb einer Klasse.

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

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

    void  MethodA()
    {
        mFieldA = 1;
    }

    void MethodB()
    {
        mFieldB = 2;

    }
}
```

Metriken

Lack of Cohesion of Methods (LCOM)

Lack of Cohesion of Methods ist eine Metrik für die Kohäsion innerhalb einer Klasse.

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

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

    void  MethodA()
    {
        mFieldA = 1;
    }

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

Metriken

Coupling (Afferent)

Zahl der Klassen, die von dieser Klasse abhänig 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)

Metriken · Visual Studio

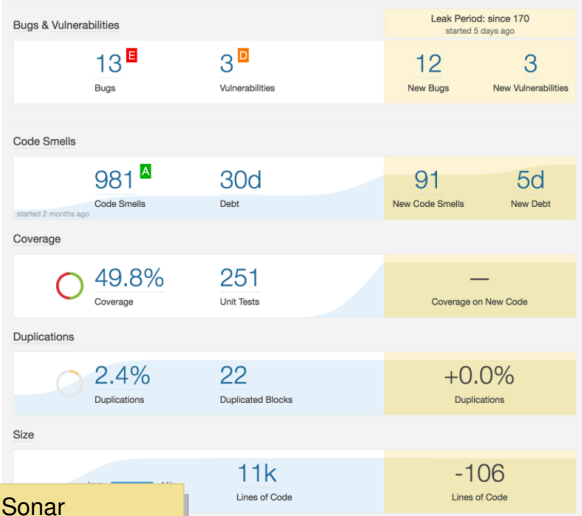
Codemetrikergebnisse

Filter: Keine | Min.: | Max.:

Hierarchie	Wartbarkeitsindex	Zyklomatische Komp...	Vererbungstiefe	Klassenkopplung	Zeilen von Quellcode	Zeilen von ausführbarem...
Irrgaerten (Debug)	84	168	3	86	1.344	398
Irrgaerten	77	33	3	47	323	80
Irrgaerten.Componentes	88	43	2	29	283	53
Irrgaerten.Rendering	75	30	2	22	195	68
Irrgaerten.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
Irrgaerten.Screen	94	12	2	9	90	16
Irrgaerten.Serialization	75	1	1	5	16	4
Irrgaerten.Utils	70	10	1	18	221	129

VS → Analyze → Calculate Metrics

Metriken · SONAR



services.sopranium.de → Sonar

Metriken · Warnung

- 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

UNI FREIBURG

Qualitätssicherung (ab Woche 3)

Code Reviews machen um die Codequalität zu verbessern, damit der Code intuitiv verständlich und damit leichter zu warten und erweitern ist.

Vorgehen:

- Ein interessantes Stück Code suchen
- Code lesen und verstehen
- Verständnisprobleme und gefundene Fehler festhalten
- Verbesserungen vornehmen
- Probleme berichten
  (Als Kommentar im entsprechenden Item)

Qualitätssicherung (ab Woche 3)

Code Reviews machen um die Codequalität zu verbessern, damit der Code intuitiv verständlich und damit leichter zu warten und erweitern ist.

Vorgehen:

- Ein interessantes Stück Code suchen
- Code lesen und verstehen
- Verständnisprobleme und gefundene Fehler festhalten
- Verbesserungen vornehmen
- Probleme berichten
  (Als Kommentar im entsprechenden Item)

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, damit der Code intuitiv verständlich und damit leichter zu warten und erweitern ist.

Vorgehen:

- Ein interessantes Stück Code suchen
- Code lesen und verstehen
- Verständnisprobleme und gefundene Fehler festhalten
- Verbesserungen vornehmen
- Probleme berichten
  (Als Kommentar im entsprechenden Item)

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*, ...)

Pfadfinderregel
Verlasse den Code immer besser als Du ihn vorgefunden hast.

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

```
> git blame InterestingCode.cs
a3b62bf (Tina Test ... 1) Woololo 0.o
959f313 (Paul Probe ... 2) fixed
> git log -p a3b62bf1
[...] Author Tina Teststudent [...]
Woololo
+ var someVar = SomeCode();
```

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

```
> git blame InterestingCode.cs
a3b62bf (Tina Test ... 1) Woololo 0.o
959f313 (Paul Probe ... 2) fixed
> git log -p a3b62bf1
[...] Author Tina Teststudent [...]
Woololo
+ var someVar = SomeCode();
```

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

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

```
> git blame InterestingCode.cs
a3b62bf (Tina Test ... 1) Woololo O.o
959f313 (Paul Probe ... 2) fixed
> git log -p a3b62bf1
[...] Author Tina Teststudent [...]
Woololo
+ var someVar = SomeCode();
```

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

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

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare 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 GameWorlds{
   private const int maxLayers = 5;
   public WorldLayer[] mWorldLayers;

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

   public void ManageLayer
      (object layer) /*...*/

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

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

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare 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[] mWorldLayers;

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

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

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

Clean Code

Habe `GameWorld.cs` angeschaut. @Tinat, schau dir mal `957baf47` an. War fast nur Codestyle. Bei einer Sache bin ich mir unsicher ...

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare 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;
  }
}
```

```
for (x in someList){
   if (!x.Initalised) return;
   somethingAwesome(x);
}
```

Qualitätssicherung

Checkliste:

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

Was Nun?

- Fragen zur Vorlesung: Donnerstags 14 Uhr
- GDD Abgabe diesen Samstag (23:59)
- Fangen Sie mit Programmieren an