Grundlagen der Softwarearchitektur

Frank Schüssele, Tobias Kolzer, Annie-Isabell Langenfeld

November 12, 2026

Organisatorisches

- Benutzen sie @Tutorname oder @Dozentenname in Gitea und Mattermost
- Donnerstags im Pool (14-16ish Uhr, Gebäude 76)
- Hausaufgabenpunkte
  - Im Gruppentreffen und sobald verfügbar im Dashboard
  - Leider ca. 30% Plagiate
- GDD Abgabe diesen Samstag (23:59)
  - Im Repository in Abgabeverzeichnis (siehe Wiki)
  - Feedback innerhalb der nächsten Woche (inhaltlich, handwerklich, Anforderungen erfüllt, zu viel/wenig)
- Postersession (Idee)
  - Erklärung nächste Vorlesung

Agenda

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

Was ist 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

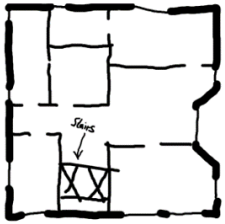
ii Das Verkürzungsmerkmal, d.h. nur die Eigenschaften des Originals, die für den Modellierungskontext relevant sind, werden repräsentiert

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

[2] Glinz, 2008

Softwarearchitektur

Als Skizze

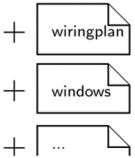


Als Blaupause



Als Programmiersprache

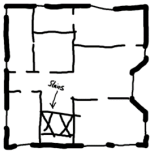




Wieso Softwarearchitektur modellieren?

- 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, Wiederherstellbarkeit
- 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, Wiederherstellbarkeit
- 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, Wiederherstellbarkeit
- 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, 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

Qualitätsmerkmale

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
- Codemetriken

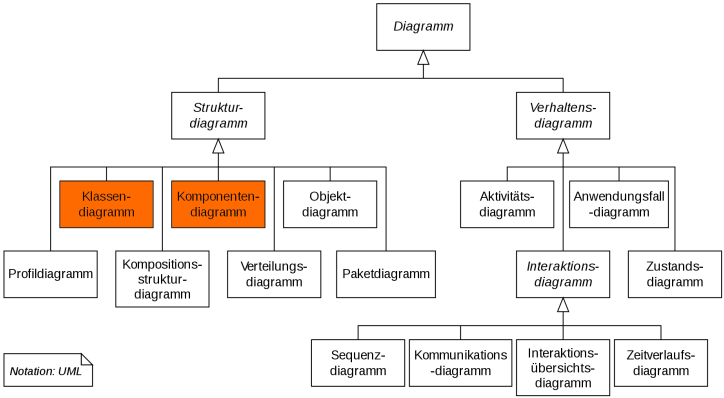
UML

UML

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

UML

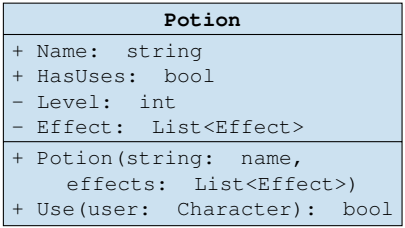
- Unified Modelling Langugage
- Modellierungssprache mit grafischer Notation
- Standardisiert, verwaltet von der Object Management Group



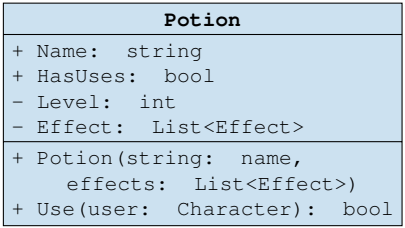
Klassendiagram

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

Klasse

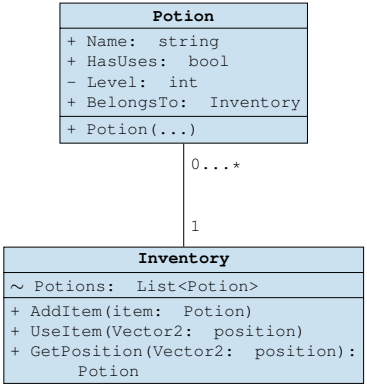

Potion
+ Name: string
+ HasUses: bool
- Level: int
- Effect: List<Effect>
+ Potion(string: name,
effects: List<Effect>)
+ Use(user: Character): bool

Klasse

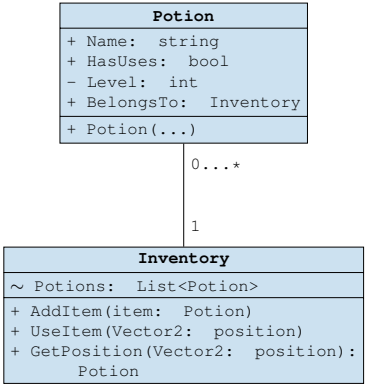



```
class Potion
{
  public string Name;
  public bool HasUses
  {
    get { return Level > 0; }
  }
  //liquid level in percent
  private int Level = 100;
  private var Effects = new List<Effect>()

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

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

Assoziation



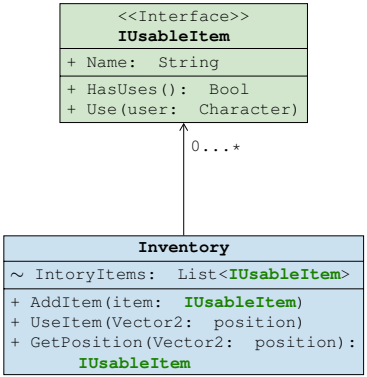
Assoziation



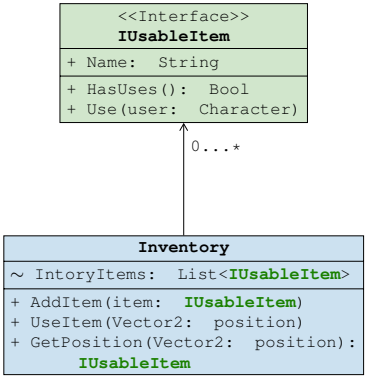

```
class Potion
{
  public string Name;
  public Inventory BelognsTo {get; set;}
  public bool Usable{ get Level > 0; }

  //liquid level in percent
  private int mLevel = 100;

  /*  ... */
}
```

```
class Inventory
{
  protected readonly List<Potion>
        Potions = new List<Potion>();

  /*  ... */
}
```

Interface


<<Interface>>
IUsableItem
+ Name: String
+ HasUses(): Bool
+ Use(user: Character)
0...*
Inventory
~ IntoryItems: List<IUsableItem>
+ AddItem(item: IUsableItem)
+ UseItem(Vector2: position)
+ GetPosition(Vector2: position): IUsableItem

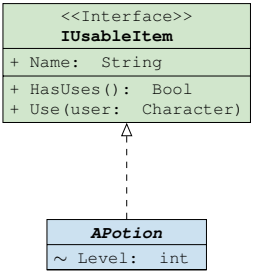
Interface



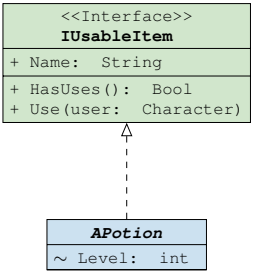

```
interface IUsableItem
{
  string Name {get; private set;}
  bool HasUses();
  void Use(Character user);
}
```

```
class Inventory
{
  protected readonly List<IUsableItem>
    InventoryItems = new
        List<IUsableItem>();

  /*  ... */
}
```

Realisierung



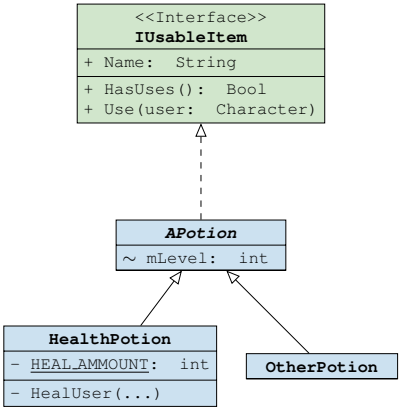
Realisierung



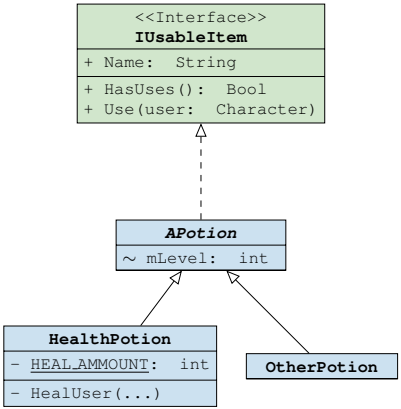

```
interface IUsableItem
{
  string Name {get; private set;}
  bool HasUses();
  void Use(Character user);
}
```

```
abstract class APotion: IUsableItem
{  /* ... */
  //Behaviour shared by all potions
  public virtual bool HasUses
    { get Level > 100; }
  //Behaviour to be implemented by potion
  public abstract void
    Use(Character user);
}
```

Vererbung


<<Interface>>
IUsableItem
+ Name: String
+ HasUses(): Bool
+ Use(user: Character)
APotion
~ mLevel: int
HealthPotion
- HEAL_AMMOUNT: int
- HealUser(...)
OtherPotion

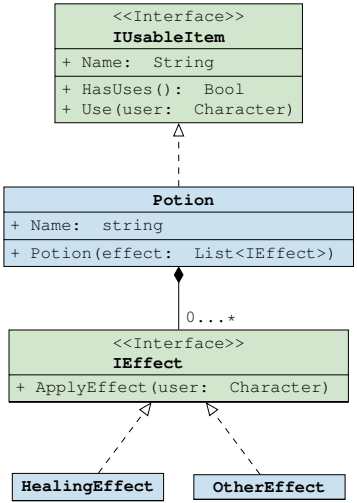
Vererbung



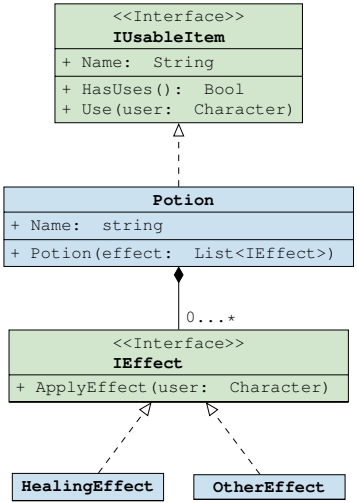

```
abstract class APotion: IUsableItem
{
  public virtual bool HasUses
    { get Level > 100; }
  public abstract void
    Use(Character user);
}
```

```
class HealthPotion : APotion
{
  public override void Use(Character user)
      { HealUser(user) }

  private void HealUser(Character user)
    {
      if (!HasUses) return;
      user.Life += HEAL_AMMOUNT;
    }
}
```

Komposition

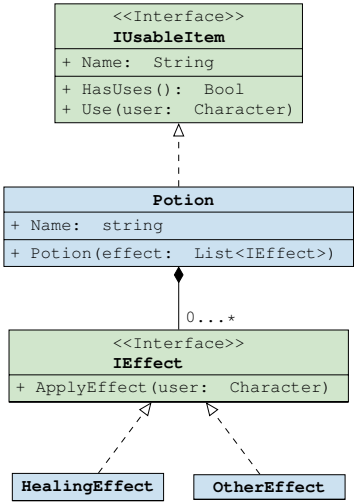


Komposition




```
class Potion
{
  private readonly List<IEffect> Effects
      //...;

  public void Use(Character user)
      {
        foreach(effect in Effects)
          effect.applyEffect(user)
      }
```

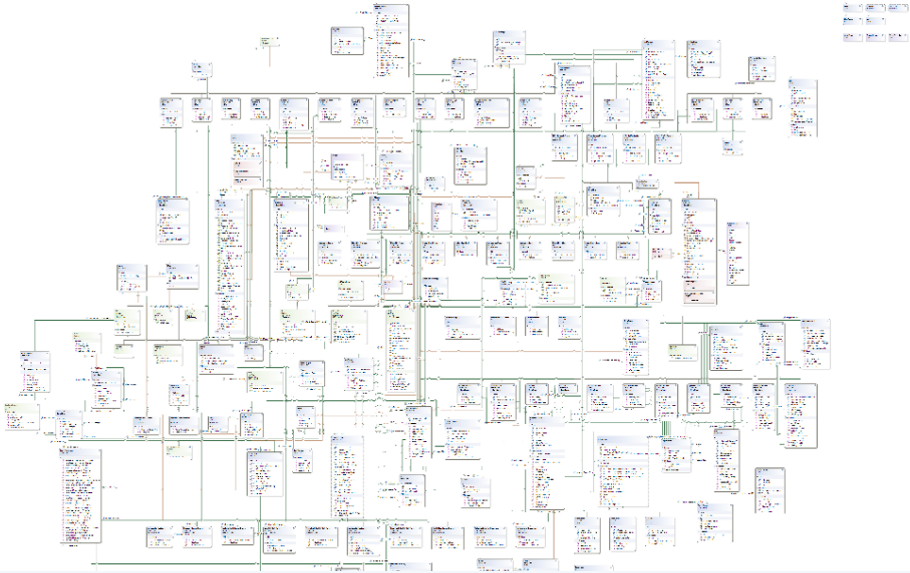
Komposition



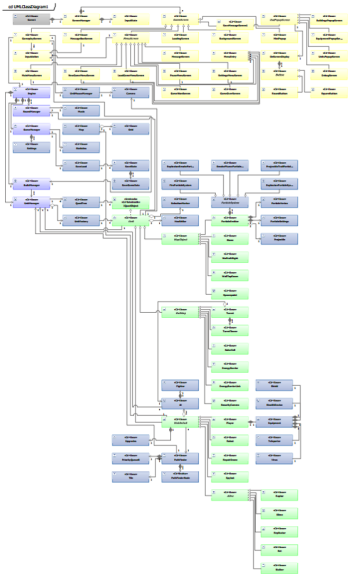

```
class Potion
{
  private readonly List<IEffect> Effects
      //...;

  public void Use(Character user)
      {
        foreach(effect in Effects)
          effect.applyEffect(user)
      }
```

```
// Builder code for an energy drink
public static Potion EnergyDrink(){
  return new Potion(
    new List<IEffect>()
      {new HealEffect(),
       new FlyingEffect()}
  )
}
```

UML



UML



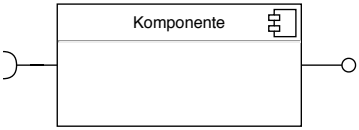
Komponentendiagramm

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

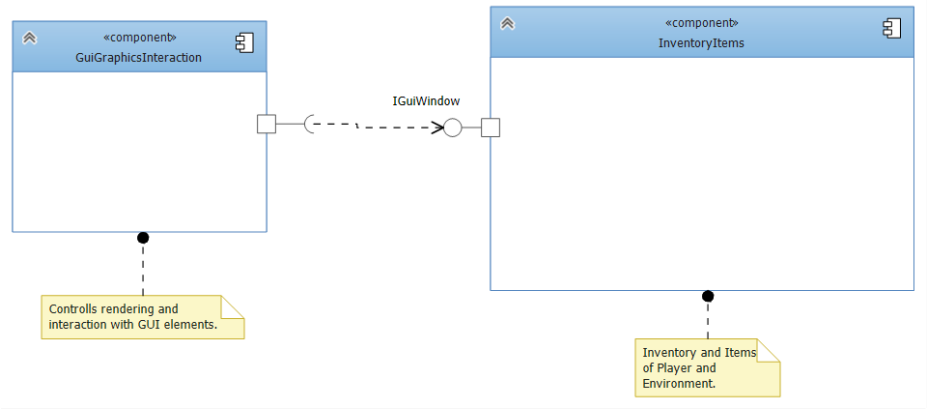
Komponente

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

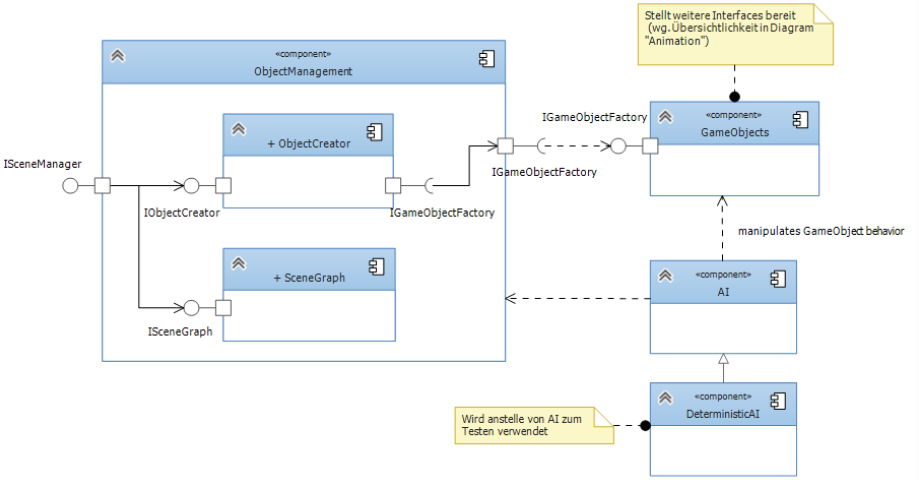
- Eine Komponente enthält z.B.: Klassen, Interfaces, Enumerations, ...
- Kann eine Schnittstelle anbieten
- Kann Schnittstellen fordern



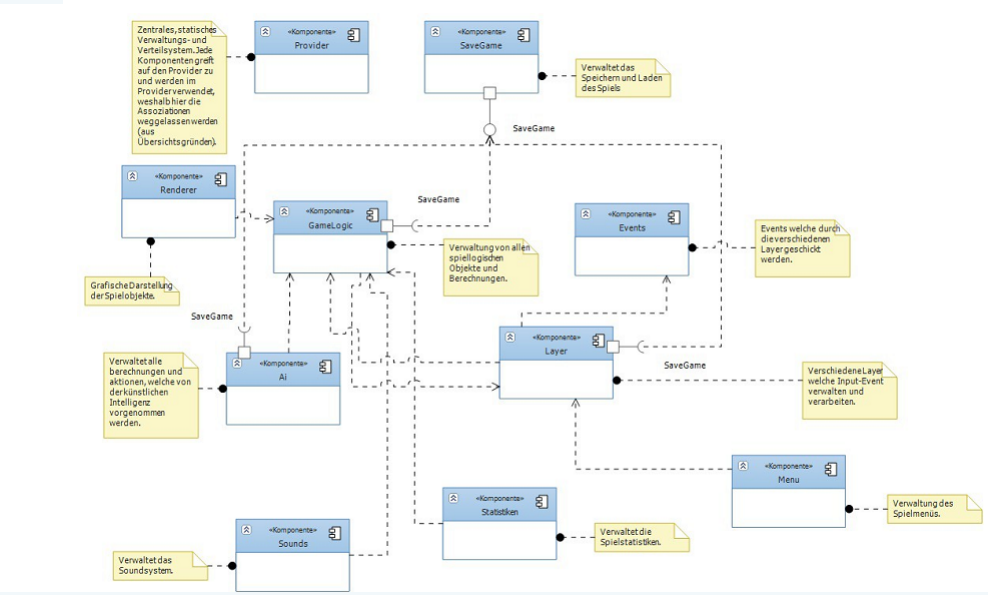
Komponentendiagramm



Komponentendiagramm



Komponentendiagramm



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.

Planung

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

Vorgehen

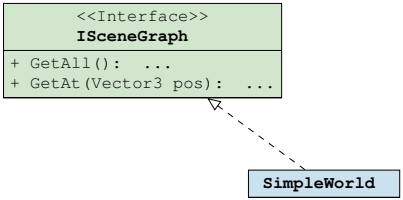
Informationen über die Lösung sammeln

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

Vorgehen

Arbeiten Sie iterativ und inkrementell

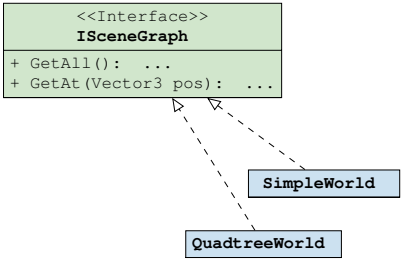
- Einfache Lösungen bauen und weiterentwickeln
- Frühzeitig validieren und dann erweitern
  - User Stories
  - Szenarien
  - Techdemo/Developer Island
- Vorsicht vor vorzeitiger Optimierung



Vorgehen

Arbeiten Sie iterativ und inkrementell

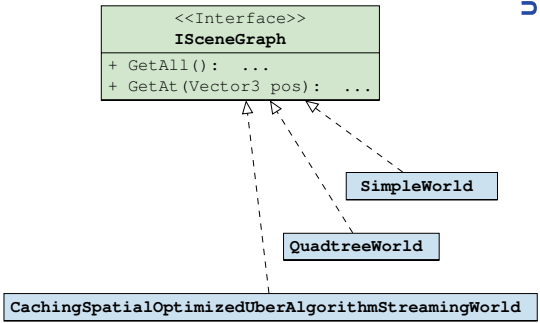
- Einfache Lösungen bauen und weiterentwickeln
- Frühzeitig validieren und dann erweitern
  - User Stories
  - Szenarien
  - Techdemo/Developer Island
- Vorsicht vor vorzeitiger Optimierung



Vorgehen

Arbeiten Sie iterativ und inkrementell

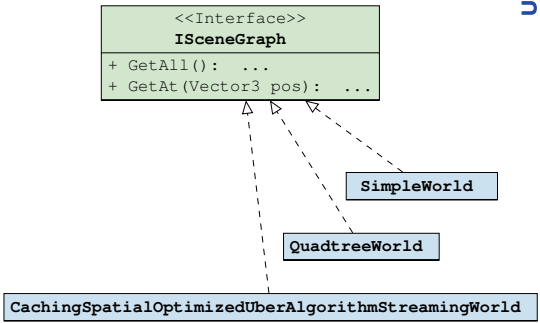
- Einfache Lösungen bauen und weiterentwickeln
- Frühzeitig validieren und dann erweitern
  - User Stories
  - Szenarien
  - Techdemo/Developer Island
- Vorsicht vor vorzeitiger Optimierung



Vorgehen

Arbeiten Sie iterativ und inkrementell

- Einfache Lösungen bauen und weiterentwickeln
- Frühzeitig validieren und dann erweitern
  - User Stories
  - Szenarien
  - Techdemo/Developer Island
- Vorsicht vor vorzeitiger Optimierung



> Make decisions as late as you responsibly can,
> because that's when you have the most information[a].
>
> [a] Mary Poppendieck, Tom Poppendieck - Lean Software Development: An Agile Toolkit

Clean Code als Architekturrichtline

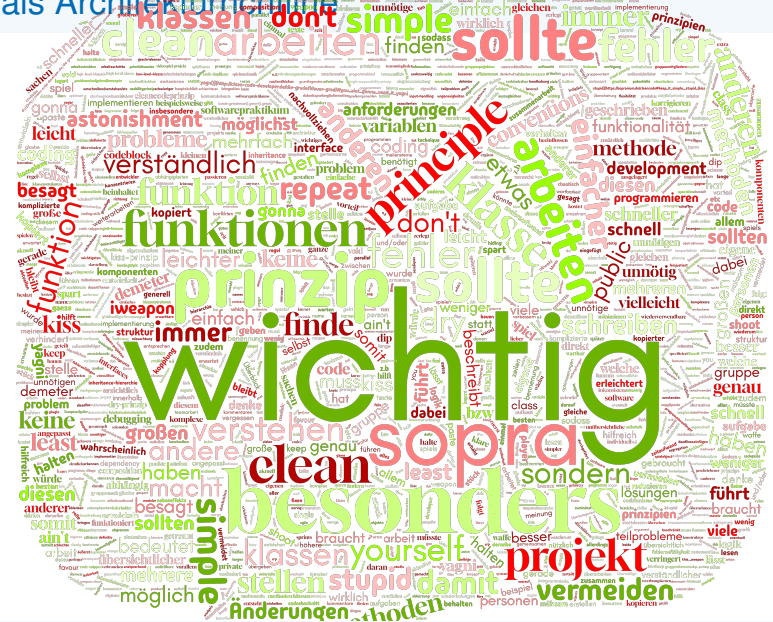
Clean Code gibt Hinweise

- 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

Clean Code als Architekturrichtlinie



Clean Code als Architekturrichtline

Single Responsibility Principle [2]

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

```
class Potion: IUsableItem
{
  public string Name;
  private int LiquidLevel = 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

Clean Code als Architekturrichtline

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 das Inventar gezeichnet wird
- Sich das Speichern/Laden ändert
- Sich Dateisystemdetails ändern
- Sich die Inventarsortierung ändert

```
class Potion: IUsableItem
{
  public string Name;
  private int LiquidLevel = 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

Validieren mit Szenarien

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, 45 FPS)

Validieren mit Szenarien

Szenario

Ein Szenario ist eine Ablaufbeschreibung mit

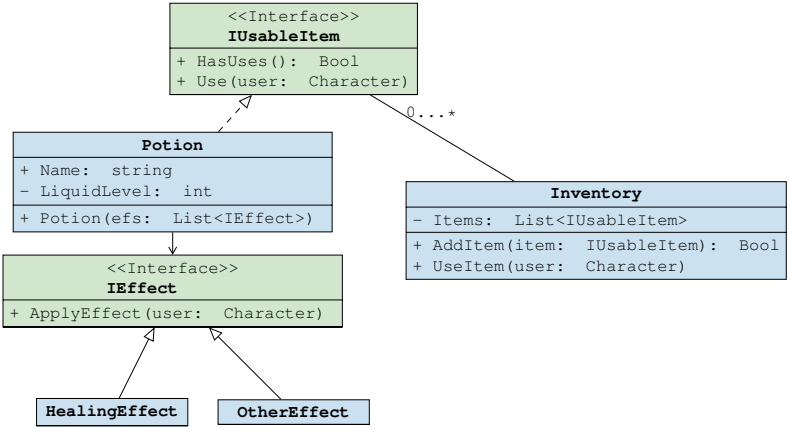
- Quelle
  (z.B. Auftraggeber)
- Auslöser
  (z.B. Modellierer, Spieler, Angreifer)
- Umgebung
  (z.B. im Endlosspielmodus, im Spielverzeichnis)
- Beschreibung
- Antwortmetrik
  (z.B. Modell erfolgreich eingebunden, gleichzeitige Darstellung, 45 FPS)

- Quelle: Auftraggeber
- Auslöser: Spieler

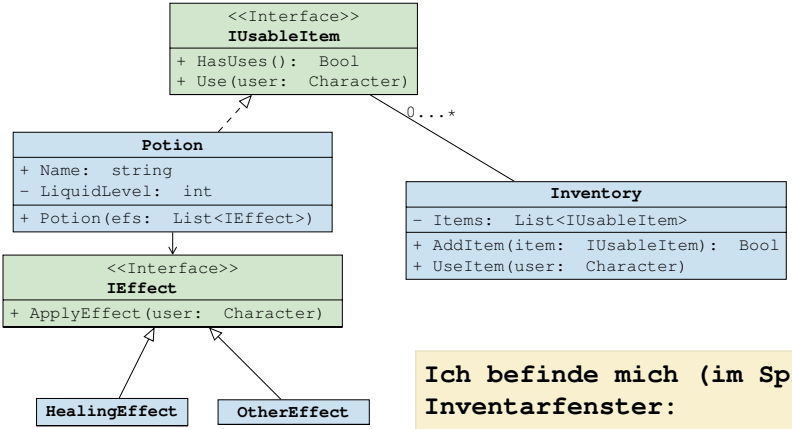
Ich starte das Spiel aus dem Hauptmenü:
Die Spielwelt, die Startspielobjekte, das HUD und die Minimap werden angezeigt.

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

Validieren mit Szenarien

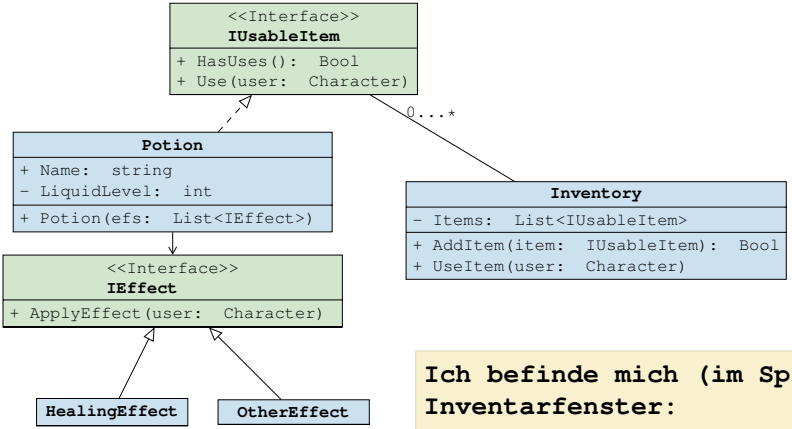


Validieren mit Szenarien



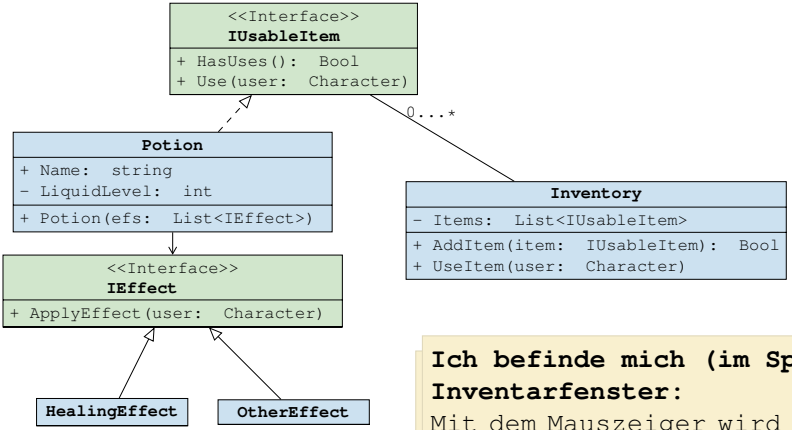
Ich befinde mich (im Spiel) im Inventarfenster:
Die Namen aller Items werden in sortierter Reihenfolge dargestellt.

Validieren mit Szenarien



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

Validieren mit Szenarien

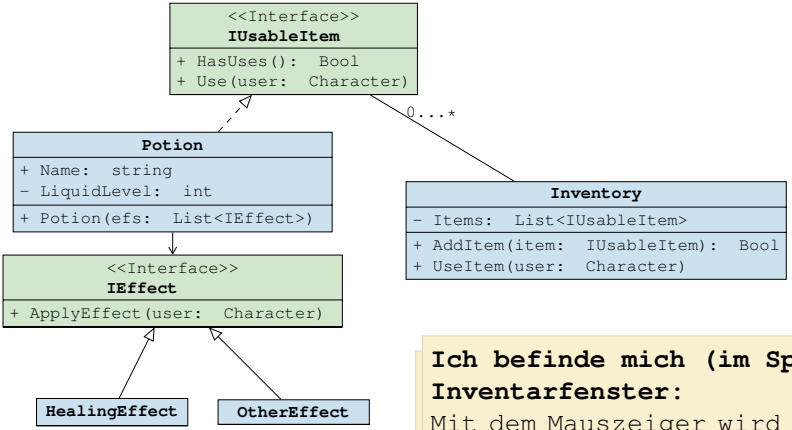


Ich befinde mich (im Spiel) im Inventarfenster:
Mit dem Mauszeiger wird auf einen Heiltrank geklickt, die Spielfigur wird geheilt.

r

✗

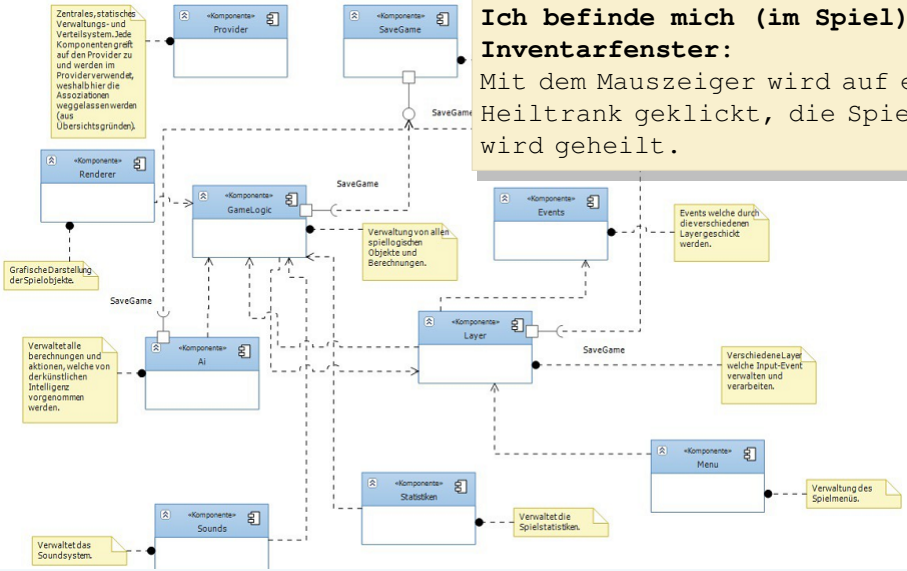
Validieren mit Szenarien



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

r ✗

Validieren mit Szenarien



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

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

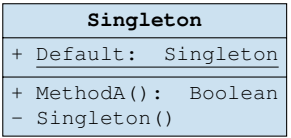
- Nützlich für Spiele:
  Composite, (Abstract) Factory, Builder, Flyweight, Observer, Visitor, Iterator
- Vielleicht nützlich:
  Object Pool, Proxy, Prototype, Decorator, Command, Strategy, ...

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 · Antipattern: Singleton




```
// This antipattern and should not be used
public class Singleton
{
  private static Singleton instance;
  public static Singleton Default
  {
    get
    {
      if (instance == null)
      {
        instance = new Singleton();
      }
      return instance;
    }
  }

  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

Metriken

Metrik[3]

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

- Gemessen in Einheiten wie z.B. cm, km, kg, km/h, ...
- Softwaremetriken sind Funktionen, die eine Eigenschaft einer Software auf eine Zahl abbildet
- Vergleichen, Bewerten
- Verschiedene Tools: Sonar, NDepend, Visual Studio

[3] IEEE Std. 610.20 (1990)

Lines of Code (LoC)

Anzahl an Zeilen einer Klasse bzw. des ganzen Projektes

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

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

```
/*
add four and five
*/
//TODO: remove cryptic comment
var x
=
4
+
5;
```

Lines of Code (LoC)

Anzahl an Zeilen einer Klasse bzw. des ganzen Projektes

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

Grundsätzlich: Jede Methode sollte vollständig auf einen Bildschirm passen

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

```
/*
add four and five
*/
//TODO: remove cryptic comment
var x
=
4
+
5;
```

Lines of Code (LoC)

Anzahl an Zeilen einer Klasse bzw. des ganzen Projektes

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

Grundsätzlich: Jede Methode sollte vollständig auf einen Bildschirm passen nur eine Sache tun und wenige Verschachtelungsebenen haben.

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

```
/*
add four and five
*/
//TODO: remove cryptic comment
var x
=
4
+
5;
```

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 int mFieldA;
    private const int SOMECONST = 5;

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

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 int mFieldA;
    private const int SOMECONST = 5;

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

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 fieldA;
  private int fieldB;

  void MethodA()
  {
    fieldA = 1;
  }

  void MethodB()
  {
    fieldB = 2;

  }
}
```

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 fieldA;
  private int fieldB;

  void MethodA()
  {
    fieldA = 1;
  }

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

Coupling (Afferent)

Zahl der Klassen, die von dieser Klasse abhängig sind

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

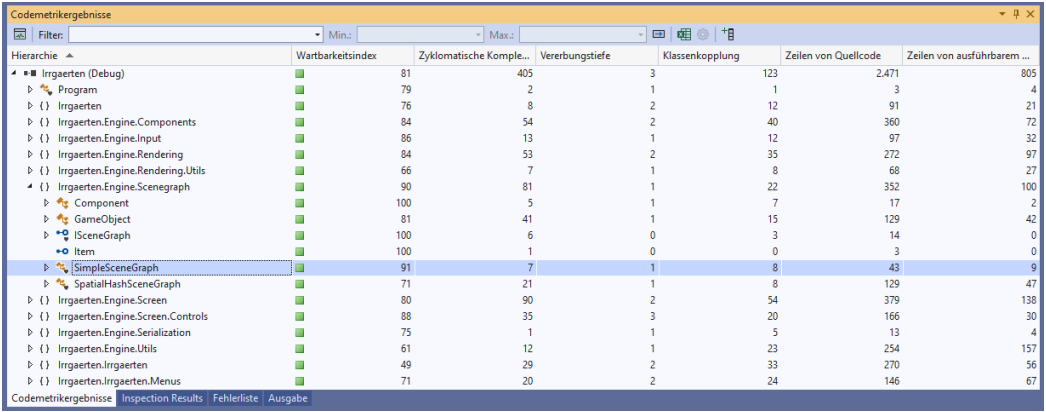
Coupling (Efferent)

Zahl der Klassen von denen diese Klasse abhängig ist

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

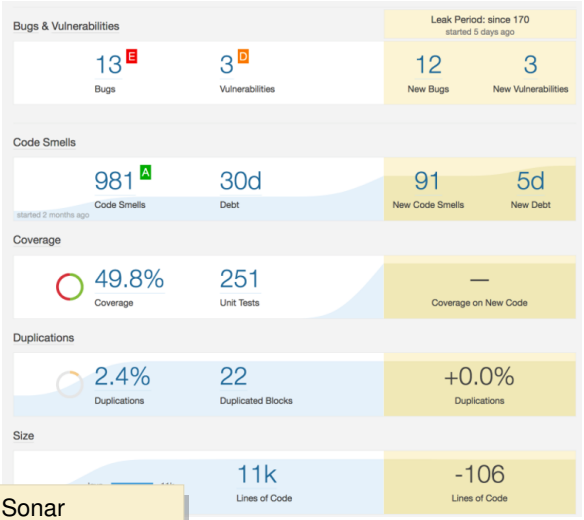


Metriken in Visual Studio



VS → Analyze → Calculate Metrics

Metriken in 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

Qualitätssicherung (ab Woche 3 je zwei Personen)

Code Reviews machen, um die Codequalität zu verbessern, mit dem Ziel, den Code verständlich und einfach erweiterbar zu machen.

Vorgehen:

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

Qualitätssicherung (ab Woche 3 je zwei Personen)

Code Reviews machen, um die Codequalität zu verbessern, mit dem Ziel, den Code verständlich und einfach erweiterbar zu machen.

Vorgehen:

- Ein interessantes Stück Code suchen
- Code lesen und verstehen
- Verständnisprobleme und gefundene Fehler festhalten
- Wenn möglich Verbesserungen vornehmen
- 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 je zwei Personen)

Code Reviews machen, um die Codequalität zu verbessern, mit dem Ziel, den Code verständlich und einfach erweiterbar zu machen.

Vorgehen:

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

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*, . . .)

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

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

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

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

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

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

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

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

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

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

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming

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

Besser:

```
private int currentItemIndex;
private int temperatureMaxInside;
// Number of frames used for moving
    default window of fps computation
private const int
  DEFAULT_WINDOW_LENGTH = 42;
protected void Remove(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) /*...*/

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

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;
  }
}
```

```
// update all entries that are already
    available
for (x in someList){
  if (!x.Initalised) return;
    updateEntry(x);
}
```

Qualitätssicherung

Checkliste:

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

Qualitätssicherung

Checkliste:

- Commitmessages, Kommentare und Naming
- Codestyle und Struktur
- Funktionalität
- Clean Code/Architektur Prinzipien

Clean Code
Habe `GameWorld.cs` angeschaut. @Tinat, schau dir bitte `957baf47` an, probleme waren .... Der Rest war fast nur Codestyle. Bei einer Sache bin ich mir unsicher . . .

Was nun?

- GDD Abgabe diesen Samstag (23:59)
- Fangen Sie mit Programmieren an