Grundlagen der Softwarearchitektur

Vincent Langenfeld, Daniel Dietsch

8. November 2019

Agenda

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

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 Sichten:
  - Statisch, Dynamisch, Verteilung, Kontext
- Verschiedene Bausteinarten:
  - Subsysteme, Komponenten, Frameworks, Pakete, Klassen
- 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:

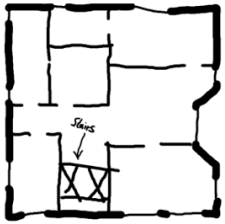
i Das Abbildungsmerkmal, d.h. es gibt eine Entität (ein Original) dessen Abbild oder Vorbild das Modell ist

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

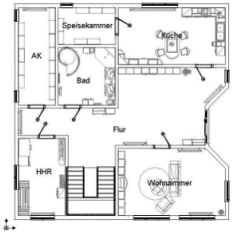
iii Die Pragmatik, d.h. das Modell wurde in einem spezifischen Kontext für einen spezifischen Zweck erstellt

[2] Glinz, 2008

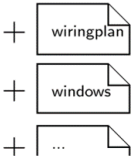
Als Skizze



Als Blaupause



Als Programmiersprache



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

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
- Metriken z.B.: Efferent/Afferent Coupling, Lack of Cohesion of Methods, Type Complexity
```

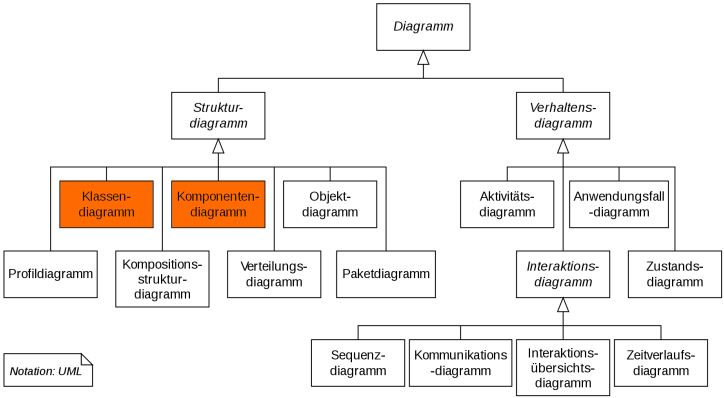
UML

UML

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

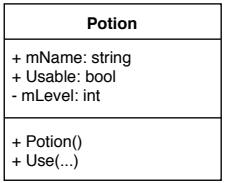
UML

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



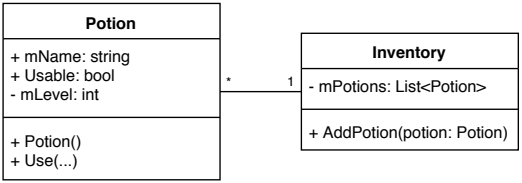
- Statische Sicht
- Klassen, Interface, Pakete und deren statische Beziehung
- Sehr nahe an der Implmenetierung

UML • Klasse




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

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

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

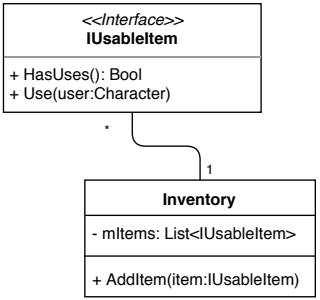
UML • Assoziation




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

        /*  ... */
}
```

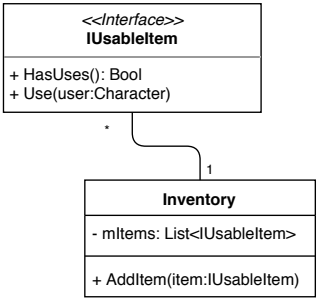
```
class Potion
{
    public string mName;
    public bool Usable
    {
        get { return mLevel > 0; }
    }
    private int mLevel = 100;

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

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

UML • Interface




```
interface IUsableItem
{
    bool HasUses();
    void Use(/*...*/);
}
```

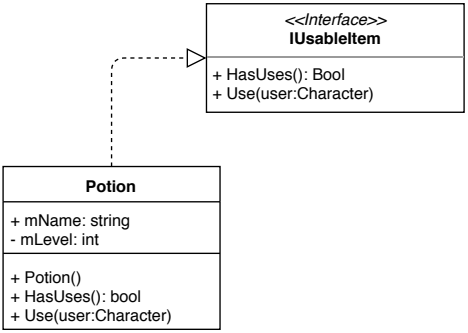
UML • Interface




```
interface IUsableItem
{
    bool HasUses();
    void Use(/*...*/);
}
```

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

    /*  ... */
}
```

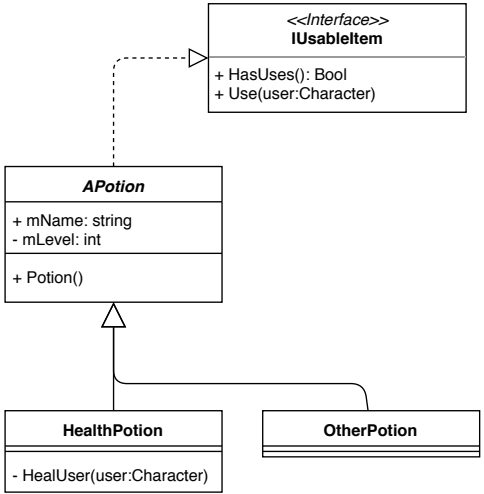
UML • Realisierung




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

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

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

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

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

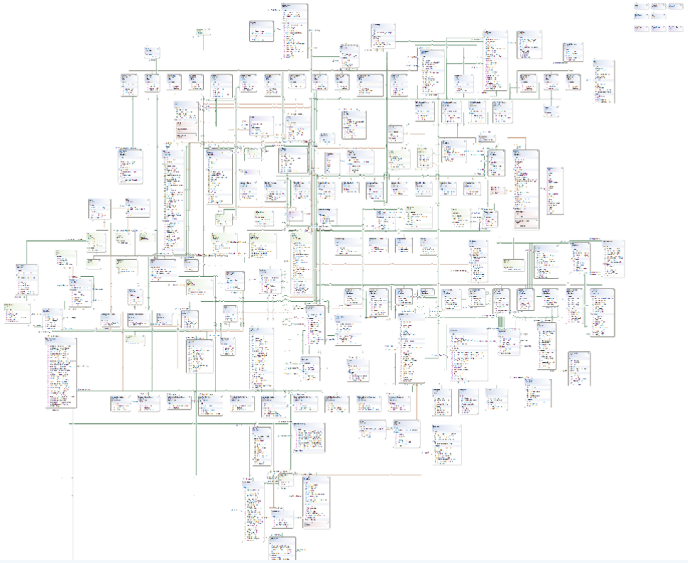
UML • Vererbung



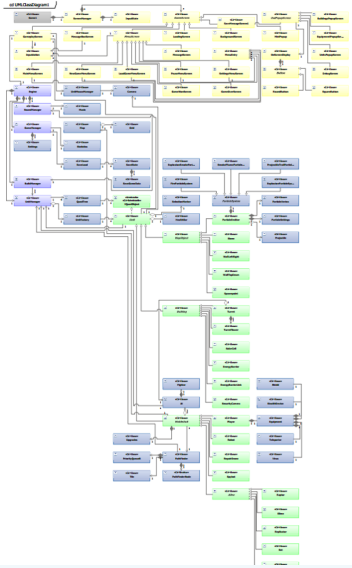

```
abstract class APotion: IUsableItem
{
    public string mName;
    protected int mLevel = 100;

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

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

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

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

UML



UML

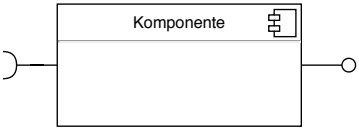


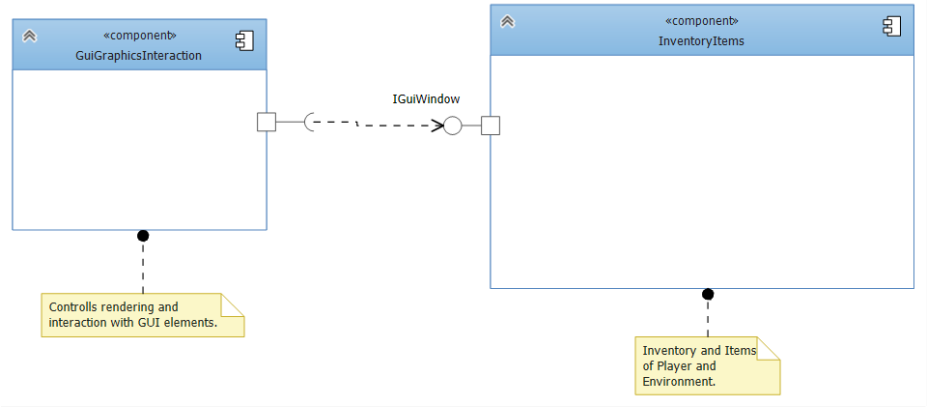
- Statische Sicht
- Komponenten, Interfaces, Parts und deren statische Beziehung als Bausteine
- Abstrakter als Klassendiagramm
  - Beschreibt die Schnittstellen zwischen den Komponenten

Komponente

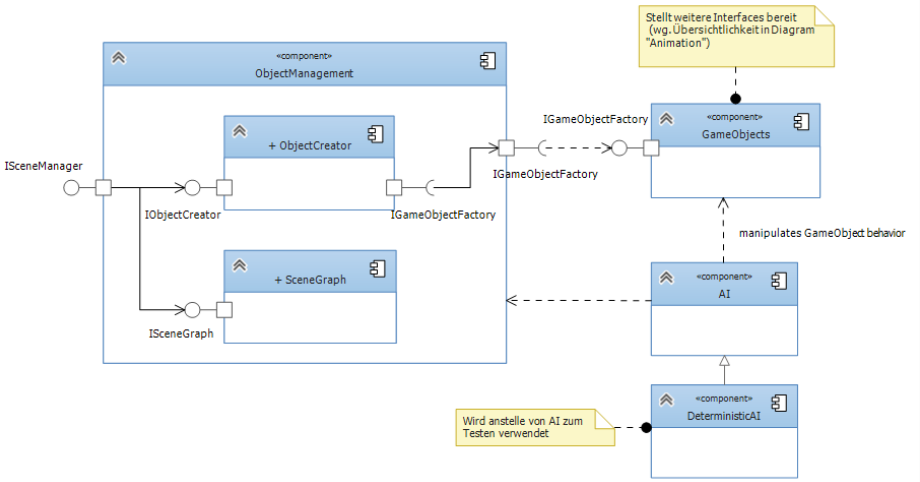
Eine Komponente ist ein Softwarebaustein, der Dritten Funktionalität über Schnittstellen zur Verfügung stellt und nur explizite Abhängigkeiten nach außen besitzt.

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

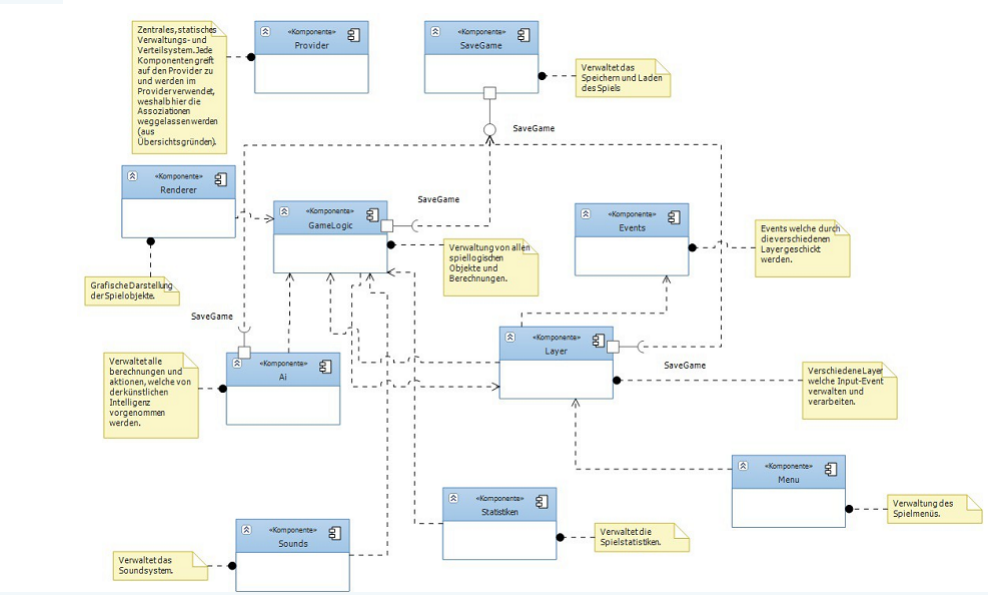



«component»
GuiGraphicsInteraction
IGuiWindow
«component»
InventoryItems
Controlls rendering and interaction with GUI elements.
Inventory and Items of Player and Environment.

UML • Komponentendiagramm



UML • Komponentendiagramm



Sie können jede Art von Werkzeug verwenden z.B.:

- Visual Studio (Design Project, siehe Wiki)
- NClass, ArgoUML, ModelMaker, StarUML
- Grafikprogramm
- Papier, Stifte, Scanner

Visual Studio

- Klassendiagramme mit zusätzlichem Plugin (siehe Wiki)
- Klassendiagramme können aus Code generiert werden

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.

Informationen über das Problem sammeln

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

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
- Vorsicht vor Optimierung
- Frühzeitig validieren
  - User Story
  - Szenarien

Szenario

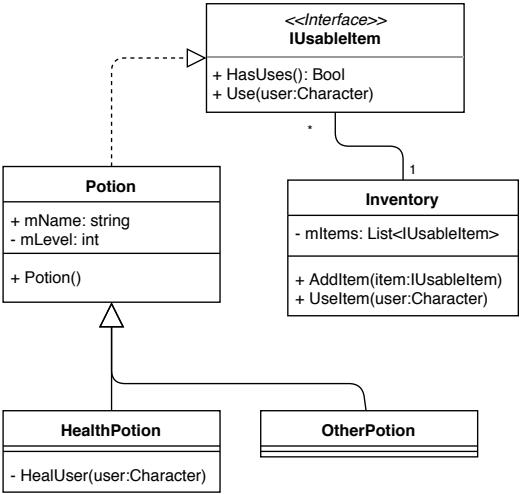
Ein Szenario ist eine Ablaufbeschreibung mit den folgenden Eigenschaften

- 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)

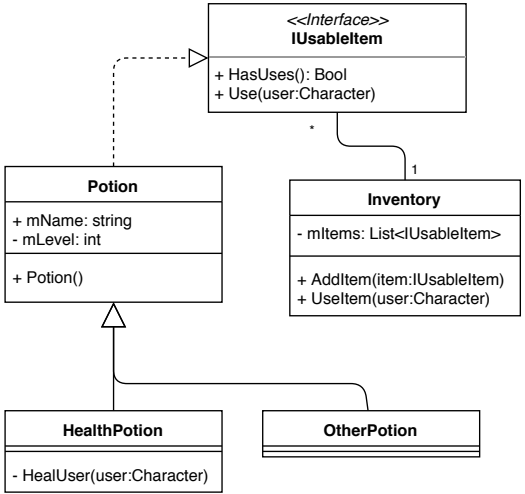
```
Die Erstellung und Einbindung eines neuen 3D-Modells durch einen Modellierer muss innerhalb von 6h abgeschlossen sein.
```

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

Planung • Beispiel

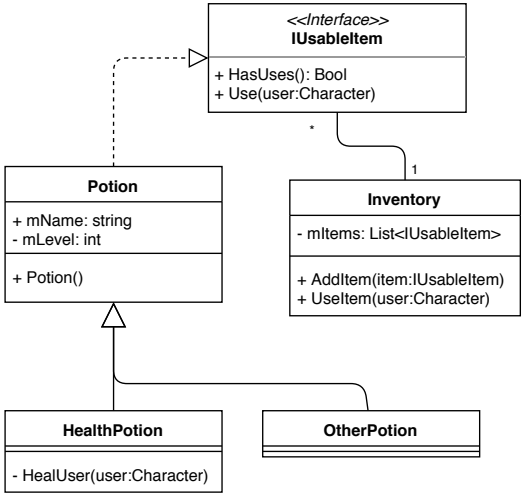


Planung • Beispiel



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.

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


Astonishment
Prinzip
Code
yourself
Teilprobleme
einfach
simple
KISS
stupid
DRY
Conventions
Responsibility
repeat
Coding
Least
least
Keep
Don't
Clean
SoC
SRP
YAGNI
Concerns
Information
Hiding
Schnittstellen
Kommentare
dokumentiert
Lesbarkeit
Inkonsistenzen
Namen
Richtlinien
Schnittstelle
Überblick
Optimierung
Sprints
gonna
ain't
Fehlern
Stelle
geändert
spart
anderer
gemeint
benötigt
Kopie
Dinge
nötig
danach
sorgt
allem
wichtigen
unnötig
Lösung
sodass
gesamte
Gruppe
klare
kopieren
darauf
Zeitaufwand
Wartbarkeit
Zusammenarbeit
Relevanz
Copy&Paste
entsprechend
könnten
doppelt
allerdings
überlegen
gehalten
Kopiere
Programmebene
wichtiges
Komponenten
notwendig
konzentrieren
Demeter
Variablen
Verwunderung
komplizierter
Optimizations
versucht
erschließt
einfachere
Erachtens
Codeblock
fremdem
Kommunikation
verlangt
Gruppenarbeit
Werte
geschrieben
folgen
bestimmten
Programme
feature
programmiert
gemeinsamen
erreichen
Developer
finden
größeren
verbessern
vermieden
nachvollziehen
entwickeln
übersichtlicher
neben
CleanCode
Dokumentation
unterscheiden
Repeat
meisten
testen
verständlich
entstehen
sinnvoll
gesamten
Anderen
nachzuvollziehen
implementieren
Verständnis
deutlich
verändert
verändern
Zweifel
wohl
indem
Separation
komplizierten
gemeinsame
hindert
gleichzeitig
nachträgliche
Zeiterfassung
Damit
conventions
ersten
Inkonsistenz
Verständlichkeit
gegebenen
Prinzipien
sparen
Programms
zerlegbar
ändern
Variabeln
auszulagern
außen
unserer
Dabei
verwendeten
bearbeiten
einzelne
geschriebenen
Leuten
jeweils
Quelltext
Problemen
bevorzugen
YAGNI-Prinzip
stehen
verschiedene
Gerade
Zeitaufwand
Person
Somit
Fall
Stil
erst
will
etwas
später
acht
bessere
tun
fuer
diesen
Beginn
kleinere
denn
nie
gilt
kostet
hilfreich
zwischen
nichts
lassen
allen
Projekts
zwei
Leute
Ort
Wiki
Kunden
gewährleistet
zusätzlich
Studierenden
können
Don't
wissen
Implementierung
reduziert
programmieren
macht
gelöst
bringt
Semesters
Änderung
achten
gerade
deren
meines
erste
keine
Low
Ziel
Blick
ihm
benötigte
Yourself
letzten
Person
evtl

Single Responsibility Principle [2]

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

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

Single Responsibility Principle [2]

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 `IUsableItem` ändert)

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

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

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

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

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

    public void SaveToFile(/*...*/)
        {/*...*/}
}
```

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

Entwurfsmuster

Entwurfsmuster

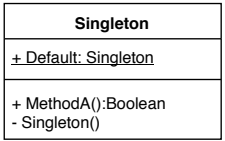
Nächstes Jahr in der Softwaretechnikvorlesung Design Patterns

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

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?




```
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

Metriken

Metrik

A quantitative measure of the deree to which a system, componente, or process possesses a given attribute [3]

- 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

[3] 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

```
Jede Methode sollte
vollständig auf einen
Bildschirm passen
```

Cyclomatic Complexity (CC)

Die Cyclomatic Complexity ist die Anzahl der Ausführungspfade innerhalb einer Methode

- Klassen: je nach Tool
  - Durchschnitt aller Methoden
  - Summe aller Methoden
- $15 < CC < 30$: kompliziert aber ok

```
class Class1
{
    private const SOMECNOST = 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();
        }
}
```

Metriken

Coupling (Afferent)

Zahl der Klassen, die von dieser Klasse abhänig sind

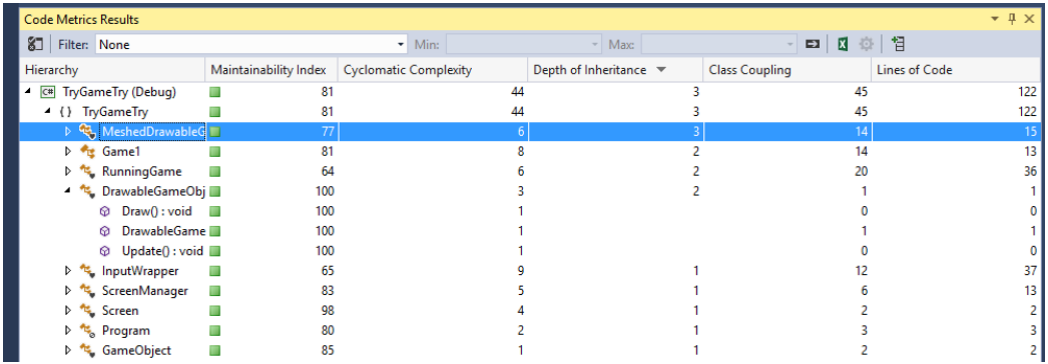
- Nicht schlimm wenn die Klasse stabil ist (z.B.: standard Libraries)
- Hohes Coupling: Änderungen an dieser Klasse verursachen Änderungen an viele anderen Stellen

Coupling (Efferent)

Zahl der Klassen von denen die Klasse abhängig ist

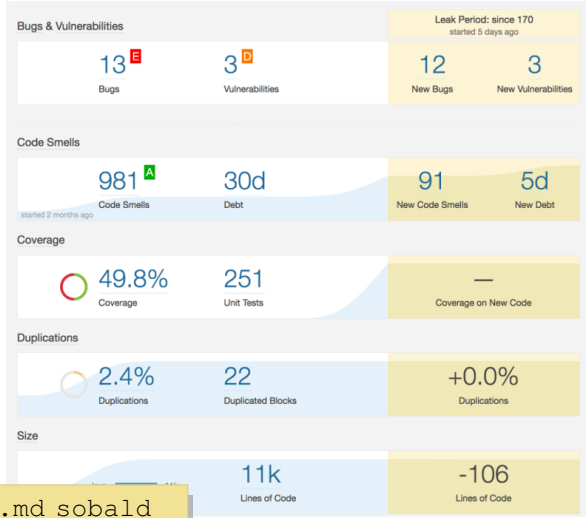
- Hinweise auf eine Gottklasse (macht alles und muss dafür jeden kennen)

Metriken • Visual Studio



`Analyze -> Calculate Metrics`

Metriken • SONAR



Link in der Readme.md sobald der Buildserver läuft

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 (Woche 3)

- Code auf Clean Code Richtlinien prüfen
- Code Reviews machen und Ergebnisse im Gruppentreffen präsentieren
- ReSharper Konformität herstellen

Vorgehen:

- Problematische Dateien wählen
- Datei auf Probleme untersuchen
  - Eindeutiges sofort verbessern
  - sonst gemeinsam lösen
- Ergebnisse dokumentieren
- Ergebnisse präsentieren
  (als Teil von Sprint Retrospektive)

```
- Ich wusste nicht wie ich Z verwenden soll
- Was macht DoComputation noch mal?
- Keine Ahnung wie ich das schöner machen
soll
- Klasse X ist zu komplex laut Sonar/VS
```

Fragen?