Meilenstein 1: Kontextanalyse und Anforderungsdefinition

Abgabetermin: Mittwoch, 23.12.2020, 18:00 Uhr

Team-Meilensteine werden über einen Commit mit entsprechendem Tag im Team-Repository abgegeben.

Aufgabe 1: Kontextanalyse

In diesem Softwaregrundprojekt wollen wir das Multiplayerspiel *Marvelous Mashup* entwickeln, um spannende Abenteuer in der Welt der Comic-Superhelden zu erleben.

Zu Beginn eines Entwicklungsprojekts ist es wichtig, sich über die Motivation der Stakeholder und den Kontext des Projekts klar zu werden. Als Ergebnis dieser **Kontextanalyse** soll ein Dokument entstehen, dass folgende Aspekte behandelt:

- Einleitung
  - Worum geht es überhaupt?
  - Was ist das Ziel des Projekts?
- Motivation
  - Warum wird dieses Projekt durchgeführt?
  - Welchen Mehrwert hat der Auftraggeber dadurch?
- Vision
  - Wie soll das fertige System ungefähr aussehen?
  - Welche Features soll es haben?
  - In welchen Szenarien soll es eingesetzt werden?
- Projektkontext
  - Wie passt das Projekt in die durchführende Organisation?
  - Welche Stakeholder sind daran beteiligt?
  - Welche möglichen Weiterentwicklungen oder Folgeprojekte sind denkbar?

Fertigt eine solche Kontextanalyse an. Das kann ein Dokument im Fließtext mit sinnvoller Gliederung sein, oder eine tabellarische Form haben.

Aufgabe 2: Fachwissen

Missverständnisse in der Kommunikation zwischen dem Kunden als Auftraggeber und den Entwicklern als Auftragnehmer können sehr problematisch für ein Projekt werden. Um das zu vermeiden, muss man sich auf eine gemeinsame "Weltsicht" und eine gemeinsame Sprache einigen.

- Identifiziert und definiert die **Fachsprache der Anwendungsdomäne.** Beschreibt dazu alle wichtigen sprachlichen **Konzepte und Entitäten** der Anwendungsdomäne in einem **Glossar.** Achtet dabei auch auf Synonyme (verschiede Begriffe mit gleicher Bedeutung) und Homonyme (ein Begriff mit mehreren Bedeutungen). Idealerweise sollte jedes Konzept mit einem eindeutigen Begriff bezeichnet werden, und ein Begriff sollte im Kontext des Projekts für genau ein Konzept stehen. Begriffe sollten wenn nötig voneinander abgegrenzt werden, um Missverständnissen vorzubeugen.

  Beispiel: Die Person, die am Computer sitzt, und *Marvelous Mashup* spielt, könnte man als "Spieler" oder "Benutzer" bezeichnen. Als "Benutzer" könnte man aber auch einen zuschauenden Client bezeichnen, der das Spiel nur passiv beobachtet. Ist mit "Team" ein Sopra-Team gemeint, oder die Gesamtheit der Charaktere, die ein Spieler steuert? Es kann also schnell verwirrend werden, wenn man Begriffe nicht konsistent verwendet.

- Dokumentiert für Konzepte auch deren **Aspekte, Rollen-** und **Spezialisierungsbeziehungen.**

Beispiel:

Begriff	Charakter
Beschreibung	Eine Figur aus allseits beliebten Comic-Heften, die durch eine Einheit auf dem Raster der Spielfelder repräsentiert wird.
Ist-ein	bewegliche Einheit
Kann-sein	Player Character (PC), Non-Player Character (NPC)
Aspekt	Name, zugehöriger Spieler oder NPC, Eigenschaften, Health Points, Inventar, Position auf dem Spielfeld, ...
Bemerkung	Der Begriff "Charakter" bezeichnet einen Superhelden, der auf dem Spielfeld herumläuft.
Beispiele	Tony Stark a.k.a. The Invincible Iron Man, gehört zur Heldengruppe von Benutzer Alice, Healthpoints: 100 Movement Points: 3 Action Points: 2 Inventar: Space Stone Position: {x: 7, y: 18} etc.

Aufgabe 3: Domänenmodell

Das Domänenmodell ergibt sich aus den Konzepten und Entitäten der Anwendungsdomäne, die für das System eine Rolle spielen. Es wird in Form eines UML2-Klassendiagramms dargestellt. Wichtig ist dabei, dass hier noch *keine Implementierungsklassen (!)* modelliert werden, sondern eben nur **Konzepte der Anwendungsdomäne** sowie ihre **Attribute und Relationen** untereinander (Assoziationen und Aggregationen mit Kardinalitäten, Generalisierungen bzw. Spezialisierungen).

- Stellt ein **Domänenmodell** für *Marvelous Mashup* auf.
- Achtet auf den Unterschied zwischen **Attributen** ("einfache Werte, stecken im Kästchen drin") und **objektwertigen Aspekten** ("Beziehungen zu anderen Objekten"). Letztere sollten über Relationen ausgedrückt werden ("Pfeile/Linien zu anderen Kästchen").
- Achtet bei **Relationen** auf deren Typ und wählt die entsprechende graphische Darstellung. Gebt auch (sofern relevant) die Kardinalitäten einer Relation an.
- Viele Regeln und Randbedingungen können direkt über Relationen mit ihren Typen und Kardinalitäten ausgedrückt werden. Wenn bspw. ein Charakter 0 bis 6 Infinity Stones im Inventar haben kann, so modelliert man eben von der Domänenklasse `Character` eine Containment Relation `inventory` mit Kardinalität 0...6 zur Domänenklasse `InfinityStone`. Manche **Invarianten** können auf diese Weise aber nicht oder nur sehr umständlich ausgedrückt werden. Wenn man bspw. festlegen möchte, dass das `healthPoints` Attribut vom Typ Integer eines Felsen immer Werte zwischen 0 und 100 hat, sollte das Modell entsprechend annotiert werden (mit Annotationselementen, oder bei größerem Umfang in begleitendem Text).

Aufgabe 4: Anforderungsdefinitionen

Um ein Projekt effizient durchführen, und ein den Kundenwünschen entsprechendes Produkt entwickeln zu können, müssen am Anfang die Anforderungen möglichst präzise, korrekt, konsistent und vollständig erfasst werden.

- Listet zunächst alle **Akteure** auf, die am System beteiligt sind oder mit ihm interagieren, und gebt deren **Rollen** und **Aufgaben** an. Akteure können Menschen sein, aber auch technische Drittsysteme.
- Formuliert ausgehend vom Lastenheft die **Funktionalen Anforderungen** an das zu entwickelnde System. Beachtet dabei auch die im Domänenmodell und im Glossar etablierten Konzepte und Begriffe. Überlegt euch hierzu eine sinnvolle Hierarchie und Strukturierung der Anforderungen. Jede Anforderung sollte einen Titel und eindeutigen Bezeichner (ID) haben, präzise beschrieben sein, und eine Begründung enthalten (warum überhaupt, oder warum gerade so und nicht anders). Abhängigkeiten zwischen Anforderungen sollten explizit gemacht werden. Es sollte auch klar werden, welcher Akteur in welcher Rolle die Anforderung an das System stellt, und welche Priorität sie hat.
- Formuliert außerdem die **Nichtfunktionalen Anforderungen** an das System. Dafür gelten im Wesentlichen dieselben Kriterien wie für die funktionalen Anforderungen. Achtet dabei aber besonders auf das Merkmal der Überprüfbarkeit: NFAs werden gerne eher wolkig formuliert. Das macht es einerseits für die Entwickler schwer, zu entscheiden, wie man sie konkret implementieren soll, und andererseits für den Kunden schwierig, die korrekte Umsetzung zu überprüfen. (Bspw. "Stabilität: Das System soll stabil sein." vs. "Stabilität: Bei der Abarbeitung des Anwendungsfalls Nr. 42 darf höchstens mit einer Wahrscheinlichkeit von 0.1% eine nicht abgefangene Ausnahmebehandlung eintreten, die zur vorzeitigen Beendigung des Programms führt." Bei der zweiten Variante kann man nach einiger Zeit Statistiken anschauen, und sehen, ob das Spiel das geforderte Maß an Stabilität (das ja genau definiert ist) erreicht oder eben nicht. Im ersten Fall ist nicht so klar was "stabil" bedeutet. Gar keine Ausfälle? Oder nur, dass es "gut genug funktioniert"? Und wie gut muss es dann sein für "gut genug"?)

Erfasst die funktionalen und nichtfunktionalen Anforderungen für alle Komponenten des verteilten Systems. Am besten wählt ihr dafür eine tabellarische Darstellung für Anforderungen analog zur Anforderungsanalyse für das Spiel *Gomoku* in Einzelübungsblatt 1.

Hinweise

- Wenn ihr euch nicht sicher sind, ob ihr die richtige Form für die zu erstellenden Dokumente gewählt habt, oder wieviel Umfang/Detailgrad angemessen ist, sprecht vor der Abgabe mit eurem Tutor! Er ist der Vertreter des Auftraggebers im Entwicklungsprojekt, und kann solche Fragen entscheiden. Die Tutoren geben gerne auch Feedback zu Entwürfen oder Zwischenergenissen! So kann man die Arbeit früh in die richtige Bahn lenken, und ihr lernt etwas dabei und spart Zeit.
- Die Einzelübungen dienen als Vorbereitung für die Team-Meilensteine. Die kommenden Meilensteine beinhalten deshalb ähnliche Aufgabenstellungen wie die vorangegangenen Einzelübungsblätter. Schaut euch deshalb vor der Bearbeitung der Meilensteine nochmals die Korrekturen und das Feedback eures Tutors zu den entsprechenden Aufgaben an. Versucht dabei, in den Übungen gemachte Fehler bei den Meilensteinen zu vermeiden und die Verbesserungstipps des Tutors umzusetzen.
- Ein trotz aller Hinweise immer wieder und oft gemachter Fehler ist, im Domänenmodell außer oder statt den Entitäten der Anwendungsdomäne auch Implementierungsklassen darzustellen. Ins Domänenmodell gehören keine Klassen für Komponenten wie Clients, Widgets der graphischen Oberfläche, Netzwerkcontroller, JSON-Parser oder sonstige "technische Sachen", sondern bspw. im Fall von *Marvelous Mashup* die Konzepte des eigentlichen Spiels wie u.a. Benutzer, Charaktere, Infinity Stones, das Spielfeld, Goose, Stan Lee etc.

  Das Domänenmodell beschreibt um *was* es in einer Anwendung geht, nicht *wie* es konkret technisch umgesetzt ist.

  Später fertigt man dann einen Implementierungsentwurf an, in dem (drum heißt er so) dann auch Implementierungsklassen enthalten sind.
- Im Lauf der nächsten Meilensteine werden etliche Dokumente erzeugt. Diese müssen digital erstellt werden. Insbesondere für Diagramme gibt es eine Vielzahl von guten graphischen Editoren (z.B. Draw.io), oder Tools die Diagramme aus textueller Syntax generieren (z.B. PlantUML, Mermaid). Die Dokumente sollten ordentlich und übersichtlich aussehen, und vorzugsweise innerhalb des Teams einem einheitlichen Stil folgen.
- Und noch ein letzter Hinweis: im Domänenmodell sollten keine Implementierungsklassen modelliert werden ;)