SEP 2022

– Aufgabenblatt 2 –

spätester Abgabetermin: 25.05 2022 12 Uhr

Ziel des Blattes

Auf dem letzten Aufgabenblatt wurde ein Pflichtenheft erstellt. Aufbauend auf diesen Ergebnissen wird nun die Systemanalyse bzw. der Entwurf durchgeführt.

Dieser Schritt hilft unter anderem besser die fachliche Domäne zu verstehen. Aufbauend darauf sollen Sie einen Entwurf des Systems gestalten, der als Grundlage für die Implementierung dienen wird.

Zunächst müssen Sie die Use-Cases detaillieren, indem Sie für jeden Use-Case mindestens ein Aktivitätsdiagramm erstellen.

Danach müssen Sie ein fachliches Klassendiagramm für die Systemanalyse erstellen, wobei Sie die in der fachlichen Domäne verwendeten Entitäten als Klassen darstellen. Dieses Diagramm erweitern Sie danach mit technischen Details, damit Sie die statische Struktur des Systems detailliert entwerfen.

Beim Modellieren soll auf eine starke Bindung und lose Kopplung geachtet werden und jeder Klasse eine klar definierte Aufgabe gegeben werden (Single Responsibility Principle).

Die Schnittstellen (Methoden) werden in diesem Fall durch exploratives Prototyping mit Hilfe von Aktivitätsdiagrammen bestimmt. Diese Diagramme sollen dazu dienen, die nötigen Methoden ausfindig zu machen und zu zeigen, dass diese prinzipiell in der Lage sind, die gestellten Aufgaben erfüllen zu können.

Der nächste Schritt ist der wichtigste Teil der dynamischen Abläufe des Systems (das Spiel) in einem Sequenzdiagramm darzustellen. Dadurch legen Sie die Grundlage für die Implementierung dieser Dynamiken fest und verbessern gleichzeitig das eigene Verständnis der nebenläufigen Prozesse, die Sie später implementieren müssen.

Abschließend stellen Sie die Ausnahmefälle dar und bereiten einen Gantt-Plan für die Implementierungsphase vor.

Aufgaben

1. Stellen Sie die Use Cases mithilfe von Aktivitätsdiagrammen dar. Falls Ihnen weitere Vorgänge auffallen, die Sie nicht im Pflichtenheft haben, aber die sinnvoll zu berücksichtigen sind, erstellen Sie die zusätzlichen entsprechenden Use Cases und Aktivitätsdiagramme. Achten Sie darauf, dass jeder Anwendungsfall Ihres Pflichtenheftes in mindestens einem Aktivitätsdiagramm abgebildet ist.

2. Erstellen Sie ein statisches Analysemodell mittels eines oder mehrerer Klassendiagramme. Erläutern Sie das, was nicht offensichtlich aus dem Diagramm hervorgeht. Stellen Sie sicher, dass das Diagramm alle wichtigen fachlichen Einzelheiten darstellt, die für die Umsetzung der Use-Cases wichtig sind. Hier sind Datentypen der Attribute aber auch Methoden nicht relevant.

3. Ergänzen Sie das Analysemodell um technische Aspekte (Typen, Methodensignaturen, Stereotypen, RMI, Client/Server usw.), um die statische Struktur der Software zu veranschaulichen. Das Klassendiagramm soll die wichtigen Zusammenhänge zwischen Entitäten darstellen, die für die korrekte Funktion während der Laufzeit relevant sind. Teilen Sie dabei sinnvoll alle Klassen in Pakete ein, indem Sie ein Paketdiagramm erstellen. Die GUI-Klassen, die unmittelbar für die Darstellung der Benutzeroberfläche verwendet werden brauchen Sie nicht darzustellen.

4. Erstellen Sie ein Sequenzdiagramm, das den Ablauf einer kompletten Spielrunde veranschaulicht. Verwenden Sie dabei die bereits vorhandenen Klassen. Es ist hier wichtig, das Zusammenspiel zwischen Client und Server zu verstehen und darzustellen, insbesondere die Synchronisation des Spielvorgangs unter mehreren Clients.

5. Überlegen Sie sich, welche Exceptions Ihre Klassen werfen werden. Definieren Sie eine Hierarchie fachlicher Exceptions und veranschaulichen Sie diese in einem Klassendiagramm.

6. Beschreiben Sie für einen Computerspieler (Bot) für alle Entscheidungen Handlungsvorgaben. Es kann hierbei gegebenenfalls hilfreich sein einen Zustandsautomaten zu modellieren.

7. Erstellen Sie ein Gantt-Diagramm für die Implementierungsphase. Achten Sie hierbei darauf, dass aus Ihrem Diagramm hervorgeht wer für welchen Teil verantwortlich ist. Diese Informationen sollen auch in Ihrem Diagramm ersichtlich sein.
   Implementierung und Optimierung erstrecken sich über zwei Phasen:
   a) Grundgerüst, Dokumentation, Tests und Implementierung von Chat-, Login- und GUI-bezogene Funktionen
      (Späteste Abgabe: 22.06.)
      i. In dieser Phase sollen die Interfaces/Klassen jeweils mit ihren Methodensignaturen und die Beziehungen der Klassen umgesetzt werden.
      ii. Ferner sind Unit-Tests und Java-Doc für alle öffentlichen Methoden zu schreiben.
      iii. Schließlich müssen Chat-, Login- und GUI-bezogene Funktionen implementiert werden.
   b) Erweiterung, Validierung und Optimierung
      (Späteste Abgabe: 13.07.)

i. In dieser Phase soll die Implementierung abgeschlossen werden, indem alle verbleibenden Klassen/Methoden implementiert werden, insbesondere die Spiellogik und Synchronisation des Spielvorgang unter mehreren Clients.

ii. Hier soll dann auch der auf diesem Blatt entworfene Bot umgesetzt werden.

Hinweise

1. Wenn Ihr Modellierungstool Einschränkungen hinsichtlich der Gestaltung der Diagramme aufweist, sodass Sie von dem Abweichen müssen, was in anderen Vorlesungen gelehrt wurde, schreiben Sie das bitte dazu.

2. Wenn Sie absichtlich von dem abweichen, was in Modellierung von Software-Systemen (bzw. SE2) gelehrt wurde, ist das auch kein Problem, solange Sie das sinnvoll begründen.

3. Vermeiden Sie es, sofort das Entwurfsklassendiagramm anzufangen. Sie sollen zunächst die wichtigen Zusammenhänge zwischen den fachlichen Entitäten verstehen.

4. Benutzen Sie `«use»`-Beziehungen um zu zeigen welche Pakete auf welche anderen Pakete zugreifen dürfen.

5. Mittels Stereotypen lassen sich bestimmte Informationen leicht an Klassen „anheften“. So ist es beispielsweise möglich, Remote-Interfaces durch einen Stereotyp `«remote»` zu kennzeichnen, anstatt jedes Mal einen Generalisierungspfeil zu `java.rmi.Remote` zu zeichnen. Auf diese Weise können Sie Ihre Diagramme übersichtlicher gestalten.

6. Bedenken Sie, falls Sie RMI verwenden, dass bei RMI alle Daten, die zwischen Client und Server ausgetauscht werden, entweder `Remote` oder `Serializable` sein müssen.

7. Sehen Sie sich die *Java Collection API*[1] an. Sie werden einige dieser Klassen bzw. Interfaces brauchen.

8. Es ist nicht nötig alle GUI-Klassen von Client zu modellieren. Es reicht die Klassen zu modellieren, die unmittelbar in der Kommunikation mit dem Server stehen und so die Schnittstellen zwischen eurem System und dem GUI-Framework darstellen.

9. Machen Sie im Klassendiagramm deutlich, in welchen Paketen sich die einzelnen Klassen befinden. Das kann dadurch geschehen, indem Sie die Klassen in das Paket „schachteln“.

[1] http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

10. Es ist nicht möglich, das Sequenzdiagramm ohne Verwendung von Combined Fragments sinnvoll zu modellieren[2]. Mit Sicherheit wird die Abgabe nicht akzeptiert, wenn sie keine solche Fragmente enthält.

11. Klassen, die auf `-Manager` enden oder auf ähnliche Weise mehr einen technischen als einen fachdomänenspezifischen Aspekt darstellen, gehören i. d. R. nicht in das Analysemodell. Technische Aspekte gehören in den den Entwurf. Analyseklassen sollten vorrangig „etwas sein“ und nicht „etwas tun“.

12. java.util.concurrent kann eine Hilfe darstellen, sowohl zum Verständnis von Nebenläufigkeit als auch um Deadlocks zu vermeiden[3] (insbesondere lohnt es sich die Klassen CountDownLatch, CyclicBarrier, and Semaphore anzuschauen).

13. Wählen Sie sinnvolle Abstraktionsebenen für jedes Diagramm: Diagramme sollen wichtige Informationen veranschaulichen, aber zu detailliert sollen sie nicht sein.

14. Die Verwendung von Design Patterns können wesentlich hilfreich sein. Insbesondere können die folgenden Patterns relevant sein: Observer (Publish/Subscribe), Facade, Iterator und Strategy.

15. Versuchen Sie mit den Aktivitätsdiagrammen neben dem groben Spielablauf auch die Spielregeln zu verdeutlichen.

16. Bei Aktivitätsdiagrammen sind die Knoten Aktivitäten und die Kanten Kontrollflüsse.

Checkliste zur Vermeidung typischer Fehler

- ☐ Die Lösungen beziehen sich unmittelbar auf das Pflichtenheft: Es werden die Begriffe, die Datentypen sowie weitere Entitäten verwendet, die früher bereits ihre Namen bekommen haben, damit man nicht das gleiche unter unterschiedlichen Namen hat.
- ☐ Die dynamischen Diagramme haben eine leicht verständliche Verbindung zur statischen Struktur.
- ☐ Diagramme werden nach UML-Syntax und Semantik erstellt und zwar nicht mit “Wissen” aus Wikipedia (insbesondere die Aktivitätsdiagramme). Wikipedia stellt einige Dinge von UML falsch dar.
- ☐ Benutzung von RMI wird bei der Modellierung berücksichtigt.
- ☐ Modellierung wird unter Beachtung von Java-spezifischen Einzelheiten erledigt.
- ☐ UML-Syntax wird beachtet (insbesondere von Sequenzdiagrammen[4]).

[2] https://www.uml-diagrams.org/sequence-diagrams-combined-fragment.html
[3] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
[4] https://www.uml-diagrams.org/sequence-diagrams.html

☐ Der Teil des Systems welches auf dem Client läuft verwendet das GUI-Framework aber ist nicht dem Framework gleich.

☐ Die Aktivitäten in den Aktivitätsdiagrammen sind ausführbar (s. Sie bitte dazu die Anforderung an Aktivitätsdiagramme Nummer 2)

Anforderungen an die Aktivitätsdiagramme

Da es oft zu Problemen beim Thema Aktivitätsdiagramme kommt, geben wir Ihnen diese kleine Auszüge aus dem OMG UML 2.5 Standard, damit ihr nicht das Dokument bearbeiten müsst[5][6]

1. Alle Startknoten bekommen zu Beginn der Aktivität ein Token. Dieses Token geht dann durch die Kontrollflows/Aktionen und solange keine Joins kommen, läuft alles unsynchronisiert, d.h. sobald ein Token sich weiter bewegen kann, macht es das auch[7].

2. Executables sind die eigentlichen Aktionen, dargestellt als Rechteck mit abgerundeten Ecken. Wichtig ist aber, dass 'An ExecutableNode shall not execute until **all** incoming ControlFlows (if any) are offering tokens.'. Das heißt, Aktivitäten mit mehreren eingehenden Kanten warten bis auf jeder Kante ein Token bzw. Kontrollfluss bereit ist. Ein paar nützliche Beispiele finden Sie in der Abbildung 1.

3. Fork und Join werden beide als einfache Balken dargestellt, die auch zu einem Element kombiniert werden können, wenn sie hintereinander stehen: "A ForkNode is a ControlNode that splits a flow into multiple **concurrent** flows[8]. A ForkNode shall have **exactly one** incoming ActivityEdge, though it may have multiple outgoing ActivityEdges."

4. "A JoinNode is a ControlNode that synchronizes multiple flows. A JoinNode shall have **exactly one** outgoing ActivityEdge but may have multiple incoming ActivityEdges."[9]

5. "If a JoinNode does not have a joinSpec[10], then this is equivalent to a joinSpec Expression with the Boolean operator 'and'."

6. Merge und Decision werden beide als Rauten dargestellt, die auch zu einem Element kombiniert werden können, wenn sie hintereinander stehen (unübersichtlich).

[5] Das solltet Ihr aber in Zweifelsfällen tun (oder euren HiWi fragen).
[6] Nichtbeachtung dieser Hinweise ist nicht empfehlenswert.
[7] Wie das funktioniert, können Sie hier anschauen (1,5 Minuten): https://www.youtube.com/watch?v=qeWrvNFX-cY (Achtung! Fehler im Video bei durchgang vom ersten DecisionNode, es gibt keine Multiplizierung der Tokens beim DecisionNode.) und hier: (c.a. 4 Minuten): https://www.youtube.com/watch?v=dkd1Bw_B6jA
[8] D.h. jede ausgehende Kante bekommt einen Token.
[9] D.h. mehrere Tokens gehen rein, nur Einer geht raus.
[10] Die Bedingung dafür, dass die ausgehende Kante einen Token ausgibt == Kontrollflow weiter geht.

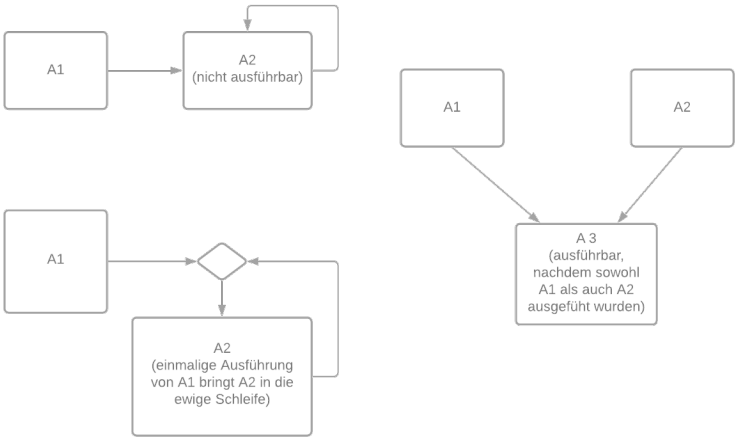

Abbildung 1: Aktivitätsdiagramme und eingehende Kontrollflüsse. Die Einbau des MergeNodes (unten links) löse oft einige Probleme (oben links).

7. "A MergeNode is a control node that brings together multiple flows **without synchronization**[11]. A MergeNode shall have **exactly one** outgoing ActivityEdge but may have multiple incoming ActivityEdges."

8. "All tokens offered on the incoming edges of a MergeNode are offered to the outgoing edge. There is **no synchronization of flows or joining of tokens**."

9. "A DecisionNode is a ControlNode that chooses between outgoing flows. A DecisionNode shall have at least one and at most two incoming ActivityEdges, and at least one outgoing ActivityEdge."

10. Bei einer Entscheidung müssen die ausgehenden Kanten eindeutig unterscheidbar sein, da es keine Auswertungsreihenfolge gibt und somit das Verhalten sonst nichtdeterministisch wird: "A DecisionNode accepts tokens on its primary incoming edge and offers them to all its outgoing edges. However, each token offered on the primary incoming edge shall traverse **at most one** outgoing edge. **Tokens are not duplicated**."

Hilfreiche Links

- OMG UML 2.5: http://www.omg.org/spec/UML/2.5/PDF
- UML-Referenz (kompakter, weniger Infos): http://uml-diagrams.org/

[11] D.h. keine Tokens werden vermehrt oder gelöscht. Das ist auch so beim DecisionNode.

Kontakt

OLAT	https://olat.vcrp.de/url/RepositoryEntry/3654517002
Leitung	apl. Prof. Dr. Achim Ebert ebert@cs.uni-kl.de
Organisation	Dr. Taimur Khan tkhan@cs.uni-kl.de
Hiwis	Andreas Tomasini
	Hasan Albakkour
	Marvin Häuser
	Mail an alle sep-support@cs.uni-kl.de