DIPLOMARBEIT

Fachgebiet der Diplomarbeit:
Software Engineering

Thema der Diplomarbeit:

Ablaufoptimierung in interaktiven Storytelling-Systemen

Unternehmen, in dem die Diplomarbeit durchgeführt wurde:
Zentrum für Graphische Datenverarbeitung, e.V.
in Darmstadt

Diplomand: Philipp Walser
Referentin: Prof. Dr. Monika Lutz
Korreferent: Dr. Stefan Göbel
Betreuer ZDGV: Dipl.-Ing. Rainer Malkewitz

Fachhochschule Gießen-Friedberg, Bereich Friedberg
Fachbereiche IEM, MND, MNI
Fachrichtung Medieninformatik, Sommersemester 2006

Selbstständigkeitserklärung

Hiermit versichere ich, Philipp Walser (Matrikel-Nr. 701817), die vorliegende Arbeit allein und nur unter Verwendung der angegebenen Quellen angefertigt zu haben.

Philipp Walser

Inhaltsverzeichnis

1 Einleitung

Interactive Storytelling ist in den letzten Jahren zu einem Schlagwort geworden, das ebenso viele Synonyme wie Definitionen besitzt. Die im Rahmen dieser Diplomarbeit verwendete Definition leitet sich aus der Bedeutung der beiden Begriffe ab, aus denen Interactive Storytelling zusammengesetzt ist:

Storytelling - zu Deutsch Geschichten erzählen – ist, abstrahiert betrachtet, eine spezielle Form der Nachrichtenübertragung. Ein Sender (Erzähler) transportiert eine bestimmte Nachricht (Geschichte) über ein Medium (z.B. Sprache) an einen Empfänger (Zuhörer). Die Nachricht ist idealerweise sowohl an das Medium als auch an den Empfänger angepasst [1].

Interaktivität ist eine Eigenschaft, in diesem Falle die Eigenschaft der Nachrichtenübertragung. Bei einer interaktiven Nachrichtenübertragung gibt es keine Trennung zwischen Sender und Empfänger. Jeder Teilnehmer kann sowohl das eine als auch das andere sein. Hierfür muss das Medium, über das der Austausch stattfindet, Feedback zulassen bzw. bidirektional sein. Chris Crawford definiert Interaktivität wie folgt [1]:

> „A cyclic process between two or more active agents in which each agent alternately listens, thinks and speaks."

Da mündliches Vortragen diese Möglichkeit bietet, kann nach dieser Definition das klassische Geschichten erzählen durchaus als interaktiv bezeichnet werden. Das reine Erzählen einer Geschichte macht diese jedoch noch nicht interaktiv. Hierfür müsste der Geschichtenerzähler seine Geschichte variieren, je nachdem, ob seine Zuhörer gerade gelangweilt oder gespannt sind. Um dies feststellen zu können, benötigt der Geschichtenerzähler direktes Feedback von seinen Zuhörern. Insofern erscheint die Definition von Crawford als unzureichend, da sie nicht explizit darauf hinweist, dass sich die Kommunikationsteilnehmer gegenseitig beeinflussen. Interaktivität entsteht nur dann, wenn sich alle Teilnehmer darauf einlassen. Statt dem Begriff „speak" sollte vielmehr der Begriff „respond" verwendet werden, um zu verdeutlichen, dass es sich um eine Reaktion handelt, für die sich der Teilnehmer aufgrund des Zuhörens („listen") entschieden („think") hat.

Für Interactive Storytelling reicht es also nicht, ein Medium zu benutzen, das Interaktivität zulässt. Die Geschichte muss auf eine Art und Weise präsentiert werden, in der sie über das entsprechende Medium eine Interaktion der einzelnen Teilnehmer forciert. Interactive Storytelling beschäftigt sich, wie der Begriff „Storytelling" erahnen lässt, weniger damit, Geschichten mit Interaktivität zu versehen, als damit, Geschichten interaktiv zu präsentieren.

Schon in den frühen 80er Jahren gab es erste Versuche, Geschichten mit Hilfe von Computersystemen interaktiv zu präsentieren. Die als Hyperfiction oder auch Interactive Fiction bezeichneten Spiele bedienen sich hierbei der in ergodischer Literatur verwendeten Cybertext-Struktur, bei der eine Geschichte in

mehrere, durch Übergänge verbundene, Abschnitte aufgeteilt wird. In Kapitel 2 wird genauer auf diese und weitere Präsentationsformen eingegangen [2], [3].

Seit ihrer ersten Verwendung vor über 20 Jahren hat sich zunächst nicht viel an solchen Systemen, deren Datenmodell zumeist einer Baumstruktur entspricht, geändert. Auch heute noch wird in story-lastigen Computerspielen eine solche Struktur verwendet. Baumstrukturen sind dafür bekannt, schlecht zu skalieren, und je mehr Entscheidungsmöglichkeiten ein Autor dem Anwender gewährt, desto unübersichtlicher und unwartbarer wird die Story [1], [4], [5].

Mit der Zeit und dem technologischen Fortschritt wuchs das Bedürfnis, anspruchsvollere Systeme zu entwickeln. Neuere Systeme verwenden für die Wahl ihrer Übergangsdefinitionen von einem Story-Abschnitt in den nächsten neben der Entscheidung des Anwenders noch weitere Parameter, die den Story-Status oder Eigenschaftswerte des Anwenders berücksichtigen. Für ein optimales Anwendererlebnis ist jedoch eine möglichst umfangreiche Analyse und Bewertung des Anwenderverhaltens notwendig. Es wird eine Instanz benötigt, die das Anwenderverhalten überwacht und darauf reagiert.

Die Fähigkeiten des Computers, Berechnungen anzustellen, werden genutzt, um einen virtuellen Erzähler zu realisieren, der dynamisch mit dem Anwender interagieren kann. Solche Systeme sind unter dem Begriff „Drama Manager" bekannt. Sie haben die Aufgabe, aufgrund bestimmter Vorgaben (z.B. nach dramaturgischen Gesichtspunkten), den Story-Ablauf zu überwachen und an den jeweiligen Anwender anzupassen. Dadurch wird dem Anwender ein gewisser Grad an Entscheidungsfreiheit gegeben, um aktiv in den Story-Verlauf eingreifen zu können. Diese Entscheidungsfreiheit und somit Interaktivität zu gewährleisten, stellt für heutige Echtzeitsystemen kein technisches Problem mehr dar. Das Problem liegt vielmehr in der qualitativen und quantitativen Bestimmung des Einflusses, den der Anwender auf den Story-Verlauf haben soll.

So soll die erlebbare Geschichte während ihres Ablaufes auf das Verhalten des Anwenders reagieren und sich anpassen, doch darf dies nur innerhalb der Vorgaben des Autors geschehen. Es ist die Aufgabe des Story-Autors, diesen Einflussgrad zu bestimmen. Ein Interactive Storytelling-System muss ihm hierfür Steuerungsmechanismen zur Verfügung stellen, die eine Ablaufoptimierung veranlassen, sollte der Anwender zu einem ungewünschten Verhalten neigen, das einen befriedigenden Storyverlauf gefährdet.

Eine Umsetzung der Anforderungen, die an ein Interactive Storytelling-System gestellt werden, ist nicht trivial. Derzeit sind noch keine kommerziell erfolgreichen Systeme bekannt, die diese Technologien verwenden. Erste Erfolge wurden allerdings bereits schon von nicht-kommerziellen Projekten wie Façade erzielt [6]. Trotz des hohen Aufwands ist es erstrebenswert, solche Systeme zu entwickeln. Die für Interactive Storytelling notwendigen Funktionalitäten lassen sich in vielen verschiedenen Bereichen einsetzen und können z.B. auch in eLearning-Applikationen / Edutainment, Serious Games, Computerspielen oder interaktiven Museumsführungen eine Vertiefung und somit Verbesserung des

Erlebnisses hervorrufen [7], [8], [9].

Ziel dieser Diplomarbeit ist es, eine Software-Architektur zu erstellen, die Story-Autoren Steuerungsmechanismen zur Verfügung stellt, um eine dynamische, zeitgesteuerte Ablaufoptimierung in computergestützten Interactive Storytelling-Systemen durchzuführen. Um dies zu erreichen, werden verschiedene Konzepte zur Ablaufoptimierung erarbeitet, aus denen sich ein experimentelles Verfahren ableiten lässt, das in der zu entwickelten Software-Architektur eingesetzt werden kann. Als Basis steht hierfür die NarrationController-Komponente zur Verfügung, die im Zentrum für Graphische Datenverarbeitung (ZGDV) im Rahmen des Forschungsprojektes „Inscape" entwickelt wird. Die NarrationController-Komponente bietet grundlegende Funktionalitäten zum Abspielen einer interaktiven Story und wird im Laufe der Diplomarbeit um eine Ablaufoptimierung erweitert.

Bei der Entwicklung von Software-Architektur und Datenmodellen wird insbesondere auf deren Wiederverwendbarkeit und Erweiterbarkeit geachtet sowie notwendige Änderungen am bestehenden System vorgenommen, um die Flexibilität der Anwendung in Bezug auf die Erweiterung und Ergänzung von Steuerungsmechanismen zu erhöhen (Refactoring[1]).

An die Umsetzung dieses Vorhabens wird folgendermaßen herangegangen: Zunächst werden Motivation und Sinn hinter einer Ablaufoptimierung in Interactive Storytelling-Systemen beleuchtet und was bei einer Umsetzung dieser in computergestützten Anwendungen zu beachten ist.

In Kapitel 3 werden die im Rahmen dieser Diplomarbeit verwendeten Komponenten beschrieben sowie die Funktionalität, die sie zum Beginn der Diplomarbeit aufweisen. Daraufhin wird in Kapitel 4 eine umfassende Analyse angestellt, um die für eine Ablaufoptimierung notwendigen Erweiterungen zu ermitteln. Aus verschiedenen Konzepten wird ein Verfahren hergeleitet, welches es ermöglichen soll, diese in einer Interactive Storyteling Anwendung zu zu realisieren und im System abzubilden.

Aus dem Verfahren lassen sich in Kapitel 5 die Anforderungen an die Systemarchitektur ableiten, worauf in Kapitel 6 deren Umsetzung beschrieben wird. Neben der Beschreibung der neuen Funktionalitäten wird hierbei auch auf das Refactoring der bestehenden Komponente eingegangen.

Kapitel 7 geht auf einige Implementierungsdetails ein, die zum besseren Verständnis der neu entwickelten Software-Architektur beitragen sollen, so dass in Kapitel 8 einige Konzepte anhand von Beispielen umgesetzt werden können. Die daraus resultierenden Ergebnisse werden in einem Fazit zusammengefasst.

1 Siehe Glossar

2 Gründe für eine Ablaufoptimierung

2.1 Einleitung

Die Notwendigkeit einer Ablaufoptimierung in Interactive Storytelling-Systemen ergibt sich aus der Wahl der Präsentationsform und der Problematik, die diese für die jeweilige Anwendung mit sich bringt. Diese Problematik muss zunächst analysiert und die Gründe für eine Ablaufoptimierung herausgearbeitet werden. Hierbei müssen Besonderheiten, die der Einsatz in Computersystemen mit sich bringt, erkannt und berücksichtigt werden.

2.2 Interactive Storytelling - Präsentationsformen

Interactive Storytelling soll seinen Anwendern die Möglichkeit bieten, aktiv in den Story-Verlauf eingreifen zu können, um ihm ein intensiveres Erlebnis zu bieten, als es eine vorgefertigte Story, die nicht auf die Bedürfnisse des Anwenders angepasst ist, könnte. Wie schon erwähnt, ist hierbei die Art der Präsentation wichtiger als das Medium, über das Interactive Storytelling betrieben wird. Tatsächlich ist es durch die Wahl einer geeigneten Präsentationsform möglich, über unidirektionale Medien Interactive Storytelling zu simulieren. Bücher z.B. haben eigentlich nicht die Möglichkeit, Feedback zu geben. Trotzdem gibt es interaktive Abenteuer-Romane, auch als Cybertext oder ergodische Literatur bekannt, in denen Entscheidungen des Lesers den Verlauf der Geschichte beeinflussen [2].

Die Geschichte wird hierfür in viele kleinere Abschnitte eingeteilt. Jeder Abschnitt verfügt über eine eindeutige Identifizierung. Am Ende eines Abschnittes stehen dem Leser verschiedene Entscheidungsmöglichkeiten zur Auswahl. Jede Entscheidung ist an einen Folge-Abschnitt geknüpft. Damit ist es möglich, das für Interaktivität notwendige „Zuhören", „Denken", und „Reagieren", also die Rolle des Erzählers, teilweise zu simulieren. Hierbei muss darauf geachtet werden, dass durch die Abschnittssprünge keine logischen Fehler in der Story auftauchen, was eine Herausforderung an den Autor darstellt. Ansonsten kann es passieren, dass die Entscheidungen des Anwenders ihn in logische Sackgassen führen.

Durch diese Präsentationsform ist es möglich, Entscheidungen des Anwenders zu berücksichtigen. Doch durch das unidirektionale Medium ist kein weiteres Eingreifen in den Ablauf der Geschichte möglich, sollte es zu unvorhergesehenen Ereignissen kommen. Man könnte in diesem Fall von einem statischen Interactive Storytelling sprechen. Erzähler und Anwender sind klar getrennt als Sender und Empfänger. Dem gegenüber steht das dynamische Interactive Storytelling. Hier existiert keine explizite Einteilung in Sender und Empfänger. Trotzdem muss einer der Teilnehmer die Rolle des Erzählers einnehmen, der für die interaktive Präsentation der Story verantwortlich ist.

Die stärkste Ausprägung von dynamischem Interactive Storytelling wird wahrscheinlich in Rollenspielen betrieben, allen voran in Pen&Paper-Rollenspielsystemen. Hierbei gibt es mehrere Spieler und einen Spielleiter. Die Spieler nehmen die Rollen unterschiedlicher Charaktere ein, während der Spielleiter die Aufgabe hat, diese in eine Geschichte einzubinden. In einigen Rollenspielsystemen wird der Spielleiter daher auch tatsächlich als „Storyteller" bezeichnet. Die Spieler können nun durch ihre Charaktere Einfluss auf die Geschichte nehmen. Der Spielleiter muss dabei eine Balance herstellen, zwischen dem, was die Spieler erleben wollen und dem, was die Geschichte sie erleben lässt. Im Idealfall haben die Spieler das Gefühl, dass alle Ereignisse nur deswegen in der erlebten Form stattgefunden haben, weil sie sich durch ihre Aktionen so dafür entschieden haben [10].

Diese Balance ist ein wichtiges Kriterium, um Teilnehmer der Interactive Storytelling-Anwendung zu motivieren, sich an der Geschichte zu beteiligen und tatsächliche Interaktion zu betreiben. Ein solches immersives[2] Erlebnis, bei dem der Anwender in seiner Tätigkeit aufgeht, wird in der Psychologie als Flow-Erlebnis bezeichnet. Dieser Ausdruck wurde vom Psychologen Mihaly Csikszentmihalyi geschaffen und wird von ihm wie folgt beschrieben [11]:

> „...the state in which people are so involved in an activity that nothing else seems to matter; the experience itself is so enjoyable that people will do it even at great cost, for the sheer sake of doing it"

Das Ziel, eine Tätigkeit der Tätigkeit wegen zu vollführen, bezeichnet Csikszentmihalyi als Autotelie [11].

> „The term 'autotelic' derives from two Greek words, auto meaning self, and telos meaning goal. It refers to a self-contained activity, one that is done not with the expectation of some future benefit, but simply because doing itself is the reward."

Aufbauend auf den Beschreibungen von Csikszentmihalyi wurde Flow in der englischen Wikipedia wie folgt definiert [12]:

> „Flow is a mental state of operation in which the person is fully immersed in what he or she is doing, characterized by a feeling of energized focus, full involvement, and success in the process of the activity."

Im Weiteren wird von dieser Definition ausgegangen, da sie die zu erreichen gewünschte Immersion am besten beschreibt. Um Immersion zu erreichen, sind laut Csikszentmihalyi eine oder mehrere aber nicht zwingend alle der folgenden Komponenten von Nöten [11], [13]:

2 Siehe Glossar

- Wir sind der Aktivität gewachsen.
- Wir sind fähig, uns auf unser Tun zu konzentrieren.
- Die Aktivität hat deutliche Ziele.
- Die Aktivität hat unmittelbare Rückmeldung.
- Wir haben das Gefühl von Kontrolle über unsere Aktivität.
- Unsere Sorgen um uns selbst verschwinden.
- Unser Gefühl für Zeitabläufe ist verändert.
- Die Tätigkeit hat ihre Zielsetzung bei sich selbst (sie ist autotelisch)

Flow als „optimales Erlebnis" des Interactive Storytelling hat mehrere Vorteile, unter anderem die bereits erwähnte Teilnehmermotivation. Diese sorgt für eine erhöhte Aufmerksamkeit und somit Aufnahmefähigkeit. Ein Flow-Zustand hat also nicht nur den Sinn des reinen Vergnügens, sondern ist auch hilfreich, um an der vollführten Tätigkeit zu wachsen und neues dazuzulernen. Die Botschaft, die ein Autor mit seiner Story übermitteln will, wird also mit großer Wahrscheinlichkeit am ehesten in einem Zustand des Flows von einem Anwender aufgenommen. Dies ist insbesondere für die Bereiche Serious Games und eLearning interessant. Hierbei ist jedoch darauf zu achten, dass die Botschaft des Story-Autors den Anwender nicht überfordert. In eLearning-Anwendungen kann man sich dies als eine zu schwere Aufgabe vorstellen, in anderen Anwendungen, die ihren Schwerpunkt im reinen Storytelling haben, in unerwarteten Ereignissen oder nicht benutzerfreundlichen Interaktionsmöglichkeiten. Auch eine Unterforderung, die zu Langeweile führen kann, ist zu vermeiden. Csikszentmihalyi stellt dies in einem Diagramm dar (Abbilldung 1) [11]:

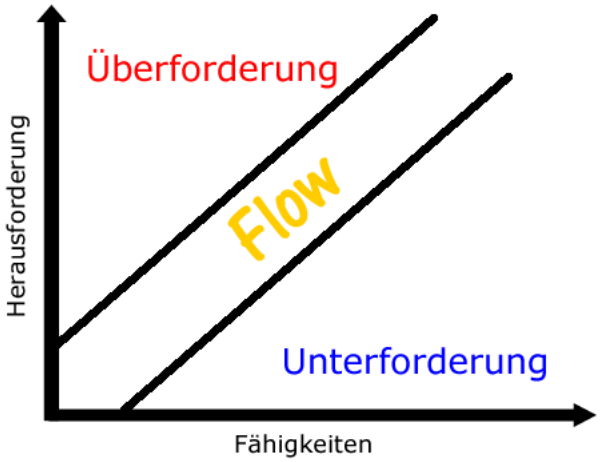

Abbildung 1: Balance zwischen Fähigkeiten und Herausforderungen

Die Balance zwischen den Fähigkeiten des Anwenders und den Herausforderungen, die an ihn gestellt werden, ist gleichbedeutend mit den Bedürfnissen des Anwenders und der Botschaft der Story. Neben den Bedürfnissen muss auch die Story selbst vom Anwender verstanden und als solche akzeptiert werden, um in einen Zustand des Flows zu gelangen. Es reicht nicht, dem

Anwender eine gewisse Anzahl an Interaktionsmöglichkeiten zu bieten. Auch allgemeine, dem Anwender bekannte und für ihn verständliche Story-Strukturen sollten Anwendung finden. Es gibt viele Nachforschungen darüber, wann ein Mensch eine Story als solche erkennt, woraus sich verschiedene Story-Modelle und Richtlinien für Story-Autoren ergeben haben. Am bekanntesten ist hierbei die Aufteilung einer Story in drei Akte: Einleitung, Klimax und Ende. Doch es gibt auch Untersuchungen, die eine Story in noch kleinere Komponenten zerlegen. So hat Wladimir Propp eine Morphologie über Russische Märchen erstellt, in der er die einzelnen Rollen, die in Märchen vorkommen, verallgemeinert und abstrahiert hat. Grundsätzlich sollte die Gestaltung der Story also einem Experten überlassen werden, damit dieser eine erlebenswerte Story für den Anwender kreiert, dem Anwender dabei aber auch die Möglichkeit gibt, seinen Bedürfnissen Ausdruck zu verleihen [5], [14].

2.3 Probleme des dynamischen Interactive Storytelling

Unterschiedliche Anwender haben unterschiedliche Bedürfnisse, um einen Zustand des Flows zu erreichen. Der Autor einer Geschichte kann diese Bedürfnisse zum Zeitpunkt der Erstellung nicht vorhersehen. Die Geschichten werden von ihm entworfen, ohne dass er weiß, wie die Anwender darauf reagieren werden. Es wäre also hilfreich, diese Bedürfnisse des Anwenders während des Erlebnisses zu erkennen, um die Story so anzupassen, dass bei dem Anwender ein Flow-Erlebnis entstehen kann.

Hierbei stößt ein Autor jedoch auf das Problem der Unvereinbarkeit von Plot und Interaktivität. Als Plot wird der konkrete Verlauf einer Story bezeichnet. Wenn dieser aufgrund eines bestimmten Anwenderverhaltens angepasst werden soll, kann es keinen vollständig vordefinierten Plot geben. Normalerweise möchte ein Autor mit seiner Geschichte aber auch eine gewisse Botschaft übermitteln oder hat sonstige künstlerische Ansprüche an den Verlauf der Story, die er vorgeben möchte. Weitere Gründe, die eher für eine vordefinierte Story sprechen, sind Ressourcen-Beschränkung (der Story-Autor kann nicht jede mögliche Entscheidung abdecken oder möchte bestimmte Entscheidungen nicht dem Anwender überlassen) oder externe Einschränkungen wie ein Zeitlimit (z.B. Museumsführung) [1], [15].

Um diesen Interessenkonflikt aufzulösen, muss der Story-Autor umdenken. Er darf keinen Plot vorgeben, da der Plot der Story individuell je nach Anwenderverhalten entstehen soll. In Pen&Paper-Rollenspielen wird dies gelöst indem, neben den möglichen Ereignissen in der Story und einem ungefähren Ablauf, vor allem Empfehlungen und Richtlinien an den Spielleiter weitergegeben werden, die bei der Anpassung der Story an die Spielrunde helfen sollen. Hierbei wird der Spielleiter idealerweise schon auf mögliche Verhaltensweisen der Spieler vorbereitet und erhält eine Auswahl verschiedener Reaktionsmöglichkeiten, die trotz der Beeinflussung durch die Spieler den gewünschten Story-Verlauf nicht gefährden. Chris Crawford beschreibt diese Aufgabe des Autors als die Definition eines Metaplots. Der Autor einer Geschichte wird dadurch

zum Autor einer vollständigen Story-Welt, in der es bestimmte zu definierende Regeln gibt, aus denen sich viele verschiedene Plotmöglichkeiten und somit für den Anwender individuelle Story-Verläufe ergeben können. Andrew Glassner stellt mit der Definition eines Story Environments eine ähnliche These auf [1], [5].

2.4 Besonderheiten der Ablaufoptimierung in Computersystemen

Computer eignen sich als bidirektionales Medium hervorragend für Interaktion. Über Eingabegeräte (Tastatur/Maus) können sie dem Anwender zuhören, Berechnungen anstellen, um über die Eingaben nachzudenken und über Ausgabegeräte (Monitor/Lautsprecher) auf den Anwender reagieren. Es liegt also nahe, Interactive Storytelling mit Hilfe von Computersystemen zu realisieren. Im Fall von Interactive Storytelling geht es aber, wie gesagt, nicht nur darum, Interaktion zu ermöglichen, sondern innerhalb eines gewissen Kontextes bestimmte Interaktionsarten/-formen anzubieten.

Zum besseren Verständnis sei hier ein Vergleich zu Computerspielen aufgeführt: Ein Großteil heutiger Computerspiele ist hochgradig interaktiv und simuliert für seine Anwender eine vollständig virtuelle Welt, in der sie unterschiedliche Objekte beeinflussen können. Doch in den meisten Fällen sorgt man mit dieser Interaktivität nur dafür, dass die Story, sofern vorhanden, fortschreitet. Die Story selbst ist meistens jedoch linear und lässt sich nicht durch unterschiedliche Verhaltensweisen der Anwender ändern [16], [17].

Die Interaktionsmöglichkeiten, die ein Spiel bietet, werden „Gameplay" genannt [17], [18]. Die Art und Weise in der einzelne Interaktionsformen miteinander funktionieren und wie der Spieler diese nutzt, wird von Craig A. Lindley als „Gameplay Gestalt" bezeichnet, wobei „Gestalt" wie folgt definiert ist [16]:

> „A gestalt may be understood as a configuration or pattern of elements so unified as a whole that it cannot be described merely as a sum of its parts."

Diese Gestalt muss üblicherweise zuvor erlernt werden, um das Spiel erfolgreich anzuwenden. Hierbei ist jedoch darauf zu achten, dass die „Gameplay Gestalt" nicht alle Interaktionsmöglichkeiten, die einem ein Spiel bietet abdeckt, sondern nur diejenigen die vom Spieler letztendlich angenommen werden. Trotzdem kann durch ein gewisses Angebot an Interaktionsmöglichkeiten eine bestimmte Art von „Gameplay Gestalt" im Spieler forciert werden. Mirjam Eladhari sagt aus ihrer eigenen Erfahrung, dass dieses „Erlernen der Gameplay Gestalt" zu einem Automatismus geworden ist, unter dem die Immersion des Anwenders leiden kann. Statt in der Story-Welt aufzugehen, versucht er herauszufinden, wie diese funktioniert, wobei nach dem Erlernen der „Gameplay Gestalt" die eigentlichen Story in den Hintergrund gerät [19].

Der „Gameplay Gestalt" gegenüber steht die „Narrative Gestalt". Auch diese Gestalt umfasst die Art und Weise, wie ein Spieler Interaktionsmöglichkeiten nutzt, in diesem Falle aber die Interaktionsmöglichkeiten mit der Story. Statt denselben sich wiederholenden Verhaltensweisen, die nach dem Erlernen der „Gameplay Gestalt" Anwendung finden, weist die „Narrative Gestalt" ein kontextsensitives Verhalten auf. Je nachdem, wo man sich in der Story befindet, stehen andere Interaktionsmöglichkeiten zu Verfügung. Dies sollte nicht als Einschränkung angesehen werden, sondern so, dass je nach dem wie sich der Spieler verhält, ein anderer Kontext entsteht und ihm somit andere Aktionen zur Verfügung stehen, die an sein Verhalten und somit an seine Gestalt angepasst sind.

Lindley stellt die beiden Gestalten gegenüber, doch glaubt sie nicht an eine Unvereinbarkeit der beiden, wie man es von Plot und Interaktivität kennt. Vielmehr legt sie den Schwerpunkt der Gegenüberstellung auf ein anderes Problem [20]:

> „...the problem is really a matter of fundamental but unacknowledged differences in the kind of experiences that different players are seeking."

Um letztendlich ein befriedigendes Erlebnis für den Anwender zu ermöglichen, stellt Lindley die folgende Hypothese auf [16]:

> „...we hypothesise that the complexity and performative demands of a gestalt must lie within a particular range for a specific person in order for a game to be engaging and immersive."

Dies deckt sich mit der Definition von Flow-Erlebnissen. Eine Anpassung der „Narrative Gestalt" auf die „Gameplay Gestalt" und somit auf die bevorzugte Interaktionsart des Anwenders, wäre also ideal. Für die Erkennung der „Gameplay Gestalt" und die Wahl der jeweiligen Interaktionsarten müssen jedoch Berechnungen angestellt und Entscheidungen getroffen werden, die selbst mit heutigen Technologien noch immer ein Problem für Computersysteme darstellen. Abhilfe schaffen hier von Lindley vorgenommenen Kategorisierungen, die bestimmte Verhaltensweisen und Anwenderbedürfnisse in verschiedene Gestalten unterteilt und somit greifbar macht. Um festzustellen, welche „Gameplay Gestalt" das intensivste Flow-Erlebnis in Anwendern hervorruft, können Flow-Messungen durchgeführt werden. Die Ähnlichkeit zwischen Computerspielen und Interactive Storytelling-Anwendungen sollte es ermöglichen, von den Erfahrungen, die bei Flow-Messung in Computerspielen gemacht wurden, zu profitieren [17], [20], [21], [22].

Derartige Flow-Messungen und darauf folgende Auswertungen der Flow-Erlebnisse werden üblicherweise in Test-/Usability-Laboren durchgeführt. Die Ergebnisse daraus können verwendet werden, um die Story-Definition zu verbessern oder aber, um komplexe Verhaltensmuster-Erkennungen durchzuführen, die bestimmte Anwenderbedürfnisse schon während der Laufzeit erkennen können

und somit eine Anpassung an der Story, also der „Narrative Gestalt", vornehmen [20], [21].

Hierbei muss jedoch auch darauf geachtet werden, dass gerade im Computerbereich eine Ablaufoptimierung ihre Grenzen hat. Design und Entwicklung einer solchen Optimierung sind im Vergleich zu Pen&Paper-Rollenspielen mit höherem Aufwand verbunden. Statt einer linearen Story müssen nun multiple Handlungsstränge erstellt werden, die aufgrund zu definierender Anwenderbedürfnisse bei der eigentlichen Ausführung einen zusammengehörigen Plot ergeben. Die Erkennung von „Flow" und „Gestalt" lässt sich hierbei auf unterschiedliche Weisen realisieren, und für die durch eine Erkennung zu treffenden Entscheidungen gibt es ebenso eine Vielzahl von Konzepten, die, je nach Vorlieben und somit Vorgaben des Story-Autors, variieren können. Das Treffen von Entscheidungen muss also weiterhin in der Hand des Story-Autors bleiben und kann nicht durch Berechnungen eines Computers ersetzt werden.

2.5 Zusammenfassung

Das Ziel einer Ablaufoptimierung ist es, den Anwender während des Erlebens einer Story in einem Flow-Zustand zu versetzen, diesen aufrecht zu erhalten und nicht zu zerstören. Hierfür muss die Story während des Ablaufs an die Bedürfnisse des Anwenders angepasst werden. Diese Anpassung des Ablaufs sorgt für einen Konflikt mit der Botschaft, die der Story innewohnt. Die Konsequenz daraus ist, dass Änderungen an der Story nur in einem bestimmten Rahmen stattfinden dürfen, der vom Story-Autor vorgegeben werden muss. Ohne diesen Rahmen besteht die Gefahr, dass die Botschaft der Story nicht mehr vermittelt werden kann und das Erlebnis vom Anwender nicht mehr als Story empfunden wird, was auch zu einer Störung des Flows führt. Um diesen aufrecht zu erhalten, muss eine Ablaufoptimierung sowohl für die Einhaltung des durch den Autor gesetzten Rahmens, als auch für das Eingehen auf die Bedürfnisse des Anwenders sorgen.

3 NarrationController und ICML

3.1 Einleitung

Die Realisierung einer Ablaufoptimierung wird als Erweiterung eines bestehenden Interactive Storytelling-Systems durchgeführt. Im Folgenden werden die schon vorhandenen und verwendeten Komponenten und das Projekt, in dessen Rahmen die Diplomarbeit angefertigt wird, vorgestellt.

3.2 Das Inscape-Projekt

Inscape ist ein Projekt der Europäischen Kommission, dessen Ziel es ist, eine vollständige Entwicklungs- und Ausführungsumgebung für interaktive, digitale Medien zu erstellen. Das Projekt startete im September 2004 und hat eine Laufzeit von 4 Jahren. In diesem Zeitraum sollen Methoden, Konzepte und Werkzeuge für Interactive Storytelling entworfen und entwickelt werden.

Die projektbeteiligten Unternehmen bekamen hierfür unterschiedliche Aufgaben aus den Themenbereichen „Forschung", „Entwicklung" oder „Anwendung" zugeteilt. Die Abteilung „Digital Storytelling" des ZGDV ist verantwortlich für die Forschung im Bereich von Story-Modellen und Narration Engine, die die Repräsentation, Ausführung und Überwachung von Stories übernehmen sollen.

Andere Unternehmen realisieren z.B. einen Story-Editor, zwei Abspielkomponenten (2D/3D) und einen Multimodal Device Controller, der für den Input in jeglicher Form (Joystick, Tastatur, Maus, Sprache u.s.w.) zuständig ist. Die einzelnen Komponenten ergeben zusammen das Inscape-Tool [23].

3.3 ICML

ICML steht für Inscape Communication Markup Language. Der XML-Dialekt ist das Bindeglied zwischen den einzelnen Komponenten des Inscape-Frameworks und enthält die vollständige Ablauflogik einer Story. Neben den inhaltlichen Elementen werden hier also auch die Interaktionsmöglichkeiten des Anwenders und die dafür zu erfüllenden Bedingungen angegeben. Dadurch ist auch definiert, in welcher Weise der NarrationController für die jeweilige Story mit den anderen Komponenten des Systems kommuniziert. ICML wird zur Laufzeit interpretiert, so dass mit ihr unterschiedliche Konfigurationen und Story-Definitionen erstellt werden können, ohne dass der NarrationController oder das Inscape-Framework neu kompiliert werden müssten. Nur bei einer Änderung des XML Schemas, und somit eine Veränderung der Struktur von ICML, würde dies nötig werden, um nach wie vor eine korrekte Interpretationder Sprache durchzuführen.

ICML gruppiert die Zuständigkeit der Komponenten in mehrere Oberelemente. „ICML_story" beinhaltet die Story-Definition, während in „ICML_player" die Ak-

tionsmöglichkeiten, die dem Anwender durch die jeweilige Abspiel-Komponente gegeben sind, hinterlegt werden. „ICML_contentElements" wiederum enthält Referenzen auf die möglichen Objekte in der Story-Welt (z.B. 2D-Grafiken, 3D-Models), die aus einer Datenbank geladen werden. Um unnötige Redundanz in den Definitionen zu vermeiden, werden innerhalb der Gruppen Referenzelemente verwendet, um auf Elemente anderer Hierarchieebenen zuzugreifen (siehe unten).

ICML befindet sich in kontinuierlicher Entwicklung und soll später einmal ein Standard für die Definition von Interactive Storytelling Erlebnissen werden. Sobald neue Komponenten oder in bestimmte Zuständigkeiten separierbare Strukturen benötigt werden, wird ICML um ein entsprechendes Oberelement erweitert. Im Rahmen der Diplomarbeit wird ICML-Version 0.95 verwendet und gegebenenfalls erweitert. Die erlaubten Elemente von ICML sind in einer XML-Schema-Definition hinterlegt und dokumentiert. Sie dient vor allem zur Validierung von ICML-Dokumenten, lässt sich jedoch zur besseren Verständlichkeit auch graphisch darstellen (Abbildungen 2 und 4).

Das Story-Model innerhalb des „ICML_story"-Elements unterstützt eine szenenorientierte Narration. Auf eine automatische Erkennung dramaturgischer Ereignisse wird verzichtet, stattdessen wird die dramaturgische Gestaltung in die Hand der Autoren gelegt. Wie in den meisten interaktiven Storytelling-Systemen wird die Story-Welt in einzelne Szenen unterteilt, die sich je nach Status in der Story-Welt unterschiedlich präsentieren können. Der Status wird hierbei durch Variablen ausgedrückt, deren Gültigkeitsbereich sich über alle Szenen erstreckt. Zwischen den Szenen können Übergänge (Transitions) erstellt werden, die unter definierbaren Bedingungen ausgelöst werden. Diese Übergänge werden in einem Szenario-Kontext definiert, so dass man unter Verwendung von mehreren Szenarien mit unterschiedlichen Übergangsdefinitionen einen anderen Blickwinkel auf dieselben Szenen erzeugen kann (siehe Listing 1).

Abbildung 3 stellt einen möglichen Story-Graphen dar, wobei die farbigen Boxen den einzelnen Szenen entsprechen, die durch Linien miteinander verbunden sind. Die Pfeile stellen die Übergangsrichtung dar. Diese graphische Repräsentation des Story-Graphen entspricht der in Listing 1 aufgeführten ICML-Definition. Wie dort zu sehen ist, besitzt eine Szene eine Liste von Aktionen (ActionSet), die während dem Ablaufen der Szene ausgeführt werden. Da Aktionsmöglichkeiten in der Hierarchie des „ICML_player"-Elementes (siehe Abbildung 4) definiert werden, wird hier ein Referenzelement benötigt (actionSetRef), um auf das entsprechende ActionSet zuzugreifen. Der Ablauf von Aktionen kann sowohl durch Bedingungen innerhalb des ActionSets beeinflusst werden als auch durch Stimuli. Ein Stimulus stellt hierbei allgemein eine äußere Einflussnahme auf den Story-Verlauf dar, sei es durch Eingaben des Anwenders oder durch Vorgaben des Autors.

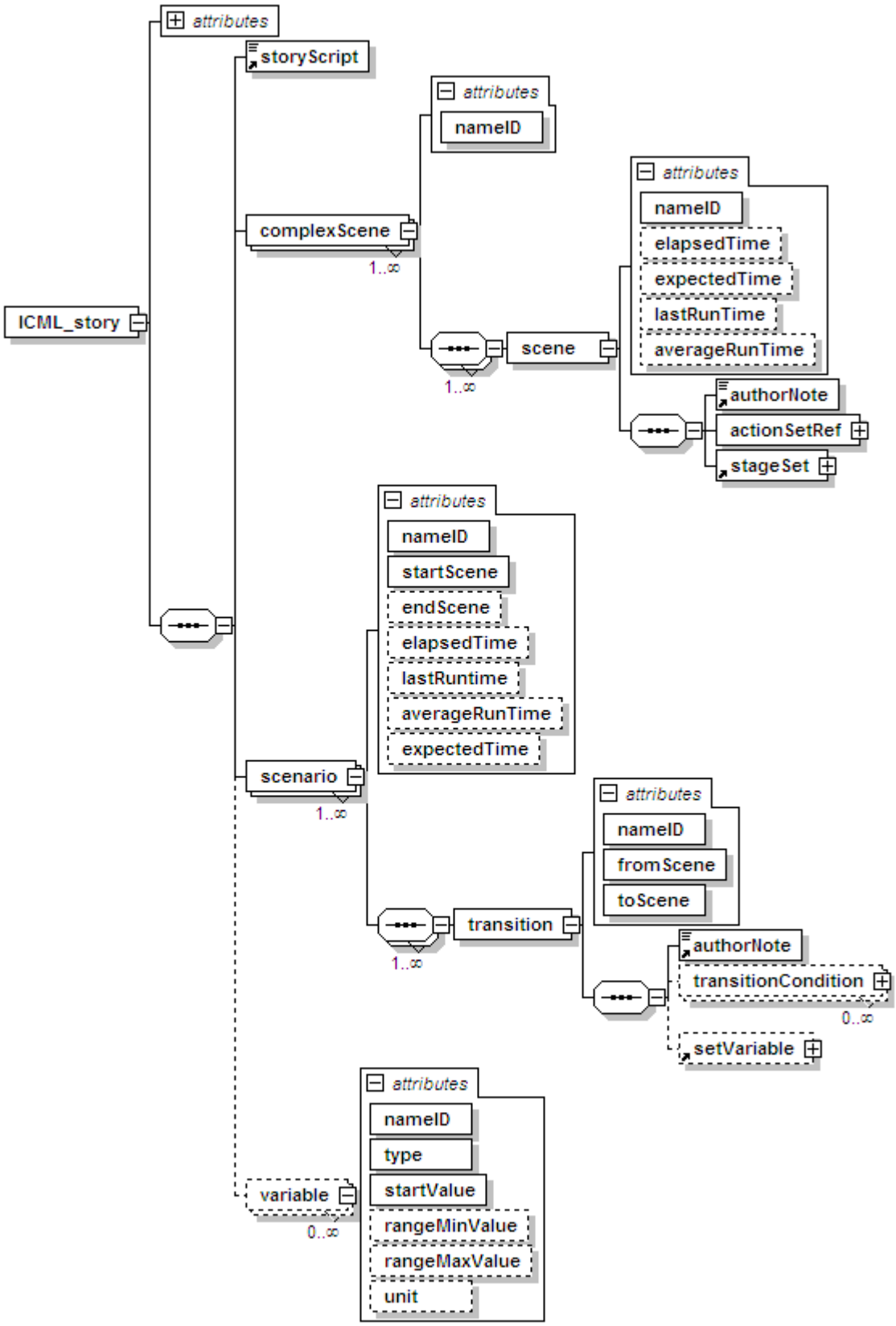

Abbildung 2: XML Schema Definition des ICML Story-Models

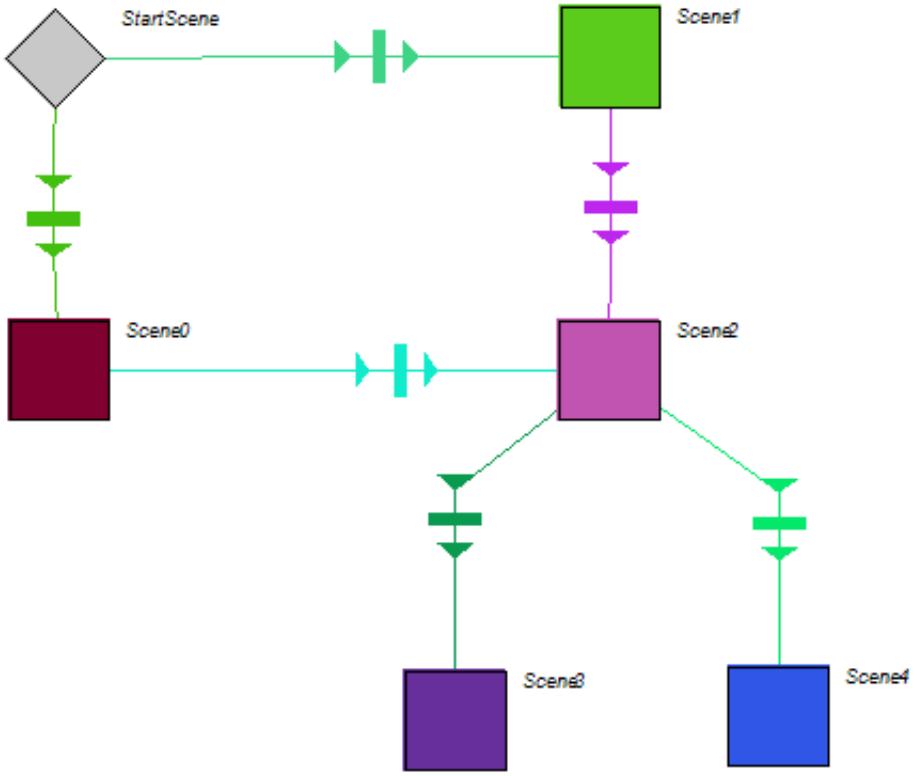

Abbildung 3: Beispiel Story-Graph mit Szenen und Transitionen

```
<ICML version='0.95'>
    <ICML_story>
        <complexScene>
            <scene name='StartScene'>
                <actionSetRef='StartSceneActions'/>
            </scene>
            <scene name='Scene0'>
                <actionSetRef='Scene0Actions'/>
            </scene>
            <scene name='Scene1'>
                <actionSetRef='Scene1Actions'/>
            </scene>
            <scene name='Scene2'>
                <actionSetRef='Scene2Actions'/>
            </scene>
            <scene name='Scene3'>
                <actionSetRef='Scene3Actions'/>
            </scene>
            <scene name='Scene4'>
                <actionSetRef='Scene4Actions'/>
            </scene>
        </complexScene>
```

```
        <scenario name='example'>
            <transition fromScene='StartScene' toScene='Scene0'>
                <transitionCondition>
                    <if_default/>
                </transitionCondition>
            </transition>
            <transition fromScene='StartScene' toScene='Scene1'>
                <transitionCondition>
                    <if_triggeredByStimuli/>
                </transitionCondition>
            </transition>
            <transition fromScene='Scene0' toScene='Scene2'>
                <transitionCondition>
                    <if_equal paramType='' param_left='' param_right=''/>
                </transitionCondition>
            </transition>
            <transition fromScene='Scene1' toScene='Scene2'>
                <transitionCondition>
                    <if_default/>
                </transitionCondition>
            </transition>
            <transition fromScene='Scene2' toScene='Scene3'>
                <transitionCondition>
                    <if_triggeredByStimuli/>
                </transitionCondition>
            </transition>
            <transition fromScene='Scene2' toScene='Scene4'/>
            <transitionCondition>
                <if_triggeredByStimuli/>
            </transitionCondition>
            </transition>
        </scenario>
    </ICML_story>
</ICML>
```

Listing 1: Beispiel einer Story-Definition in ICML

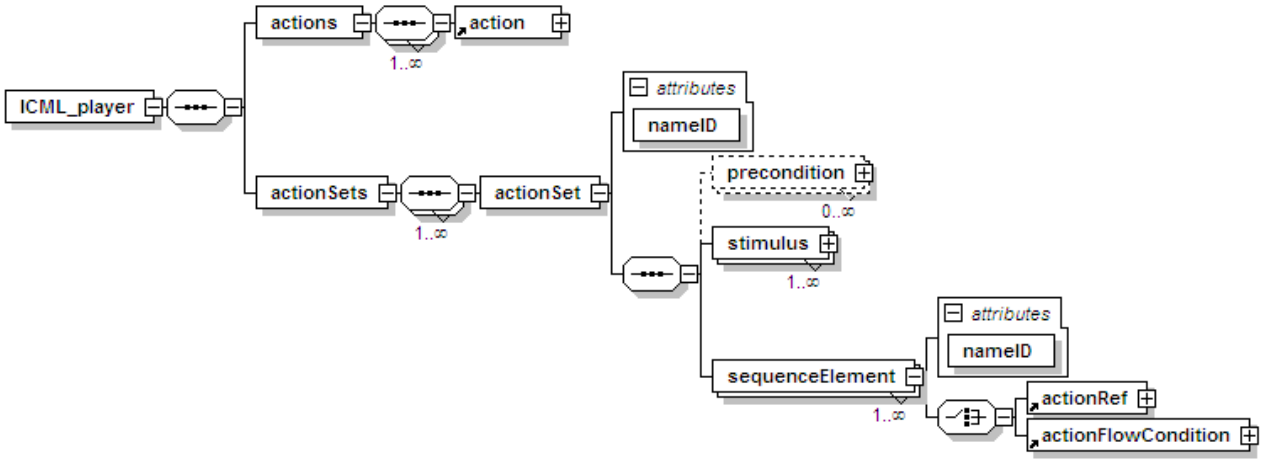

Abbildung 4: XML Schema Definition des ICML Player-Models

3.4 Die NarrationController-Komponente

Alle story-relevanten Inhalte und Aktionen werden dem NarrationController über die Definitionssprache ICML bekannt gemacht. Der NarrationController übernimmt hierbei die Aufgabe eines ICML-Interpreters und erstellt anhand der Definitionen einen Story-Graphen, auf den die anderen Komponenten des Inscape-Frameworks über verschiedene Schnittstellen zugreifen können (Abbildung 5). So nimmt der Story-Editor z.B. Veränderungen am Story-Graphen vor, während die Player-Komponente Steuerungsbefehle an den NarrationController geben kann, um den Story-Graphen oder Teile davon abspielen zu lassen. Weiterhin können sich alle Komponenten über verschiedene Observer[3]-Schnittstellen über eintretende Ereignisse während des Story-Ablaufs informieren lassen.

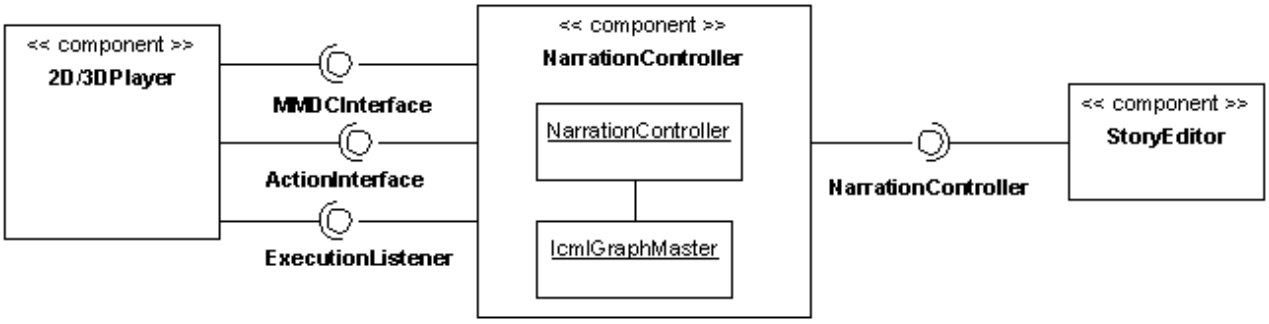

Abbildung 5: UML-Komponentendiagramm eines Teils des Inscape-Frameworks

3 Observer-Pattern, siehe Glossar

3.5 Zusammenfassung

Die Komponente „NarrationController“ und ihre Funktionalität, wie sie zum Beginn der Diplomarbeit vorliegt, unterstützt eine szenenorientierte Narration. Innerhalb der Möglichkeiten, die ICML bietet, kann ein Story-Autor beliebige Stories abspielen lassen. Durch die Verwendung des Observer-Patterns sind weiterhin erste Überwachungsmöglichkeiten der Story während des Abspielens möglich, die jedoch noch nicht genügend Möglichkeiten bieten, um eine zufrieden stellende Ablaufoptimierung zu realisieren. Diese Drama Manager-Funktionalität soll dem NarrationController im Rahmen dieser Diplomarbeit hinzugefügt werden.

4 Konzepte und Verfahren zur Ablaufoptimierung

4.1 Einleitung

Es gibt verschiedene Ansätze und Wege, eine Ablaufoptimierung in interaktiven Storytelling-Systemen zu erreichen. Im Folgenden werden einige dieser Konzepte analysiert und kategorisiert, um aus ihnen ein möglichst generelles Verfahren der Ablaufoptimierung zu entwickeln. Letztendlich sollen einem Story-Autor als Steuerungsmechanismen zur Verfügung gestellt werden, mit denen er dem Computer Richtlinien zur Entscheidung gibt, wann und in welcher Form eine Ablaufoptimierung stattfinden sollte.

4.2 Einordnung der Konzepte in Kategorien

Der Autor einer Story möchte durch sie eine bestimmte Botschaft vermitteln. Damit diese nicht durch das Eingreifen des Anwenders in die Story zerstört wird, muss eine Ablaufoptimierung sowohl für den Erhalt der Botschaft sorgen als auch auf die Bedürfnisse des Anwenders eingehen, um sein Flow-Erlebnis zu gewährleisten.

Beispielsweise könnten Anwendern, die in einer Szene nicht weiterkommen (und somit unverhältnismäßig lange für eine Szene brauchen, was den weiteren Story-Verlauf gefährden könnte), nach einer gewissen Zeitspanne automatisch in die nächste Szene befördert werden. Eine solche Strategie wäre sehr einfach, in einer für Computer leicht verständlichen Bedingung, auszudrücken. Sie schränkt aber zum einen die Interaktivität des Anwenders ein zum anderen ist die Wahrscheinlichkeit sehr hoch, dass ein plötzlicher, unerwarteter Szenenwechsel einen Störfaktor im Flow-Erlebnis des Anwenders darstellt. Eine diesbezüglich bessere Strategie könnte sein, den Anwender durch einen computergesteuerten Aktoren mit Hinweisen zu versorgen, durch die er alleine den Sprung in die nächste Szene schafft. Eine solche Strategie in Bedingungen auszudrücken ist jedoch nicht trivial, da man irgendwie den eher unscharfen Begriff „Hilflosigkeit" in eine für Computer verständliche Definition übertragen müsste.

Neben diesen Beispielen gibt es noch viele andere Möglichkeiten einer Ablaufoptimierung. Die Konzepte unterscheiden sich nicht nur in Komplexität ihrer Definition und Umsetzung, sondern auch in ihrem Anwendungsgebiet. Zur besseren Übersicht wird eine Auswahl möglicher Konzepte nach unterschiedlichen Kriterien kategorisiert. Die Kategorisierung soll dabei helfen, Gemeinsamkeiten in den Konzepten zu finden, um aus ihnen generelle Steuerungsmechanismen zu entwickeln, die sich dann auch auf weitere, hier nicht behandelte, Konzepte anwenden lassen. Zunächst gilt es jedoch, die Kriterien, anhand derer die Kategorisierung vorgenommen werden soll, zu bestimmen.

Die obigen beiden Beispiele gehen z.B. von derselben Ausgangssituation aus, wenden aber unterschiedliche Strategien an, um die Situation aufzulösen. Eine erste sinnvolle Konzeptkategorisierung wäre also eine Einteilung in Konzepte, die die Notwendigkeit eines Eingriffes erkennen und Konzepte, die einen Eingriff vornehmen.

Diese Einteilung erinnert an die Definition von Interaktivität von Chris Crawford (siehe Kapitel 1). Ein teilnehmender Agent muss „hören", „denken" und „reagieren" können. Da das angestrebte Ziel sein soll, über einen Drama Manager den Storyteller zu simulieren, muss auch dieser über solche Fähigkeiten verfügen. „Hören" ist hierbei die Überwachung der Story, „denken" das Treffen einer Entscheidung für oder gegen einen Eingriff und „reagieren" stellt den Eingriff selbst dar. Dieser Definition von Interaktivität folgend, müssen letztendlich alle Konzepte der Ablaufoptimierung eines interaktiven Systems einer dieser drei Kategorien zuzuordnen sein.

Im Folgenden gehen wir also von drei Oberkategorien aus und unterteilen diese weiter:

- Überwachung einer Story (Hören)
- Entscheidungsfindung (Denken)
- Eingriff in eine Story (Reagieren)

Die Überwachung einer Story benötigt hierbei keine weitere Kategorisierung. Der Verlauf der Story wird idealerweise mit Logging-Mechanismen festgehalten, die über Observer-Schnittstellen informiert werden (siehe 4.3). Diese Informationen können dann von verschiedenen „Denk-Konzepten" verarbeitet werden. Eine mögliche Kategorisierung für die Entscheidungsfindung kann sehr gut am vorherigen Beispiel, indem der Begriff „Hilflosigkeit" definiert werden musste, vorgenommen werden (wobei auch andere Anwendungsfälle denkbar wären): Wie schon angedeutet, kann eine Definition von „Hilflosigkeit" auf zwei unterschiedliche Weisen erfolgen: Eine Angabe von einfachen Bedingungen mit Hilfe von boolschen Ausdrücken oder eine komplexe Verhaltensmuster-Erkennung.

Für den Eingriff in die Story scheint es noch mehr Unterteilungen zu geben. Chris Crawford unterscheidet die Möglichkeiten eines Eingriffes in vier Kategorien [1]:

- Veränderung der Umwelt (Environmental Manipulation)

  Bestimmte Objekte in der Story-Welt werden manipuliert (z.B. Positionsänderung), um den Anwender von etwas abzuhalten oder ihn auf etwas aufmerksam zu machen.

- Zieleingabe / Hinweise geben (Goal Injection)

  Es werden bestimmte Aktionen ausgeführt oder Übergänge in bestimmte Szenen bevorzugt, um den Anwender zu einem bestimmten Verhalten zu bewegen.

- Anwender automatisch in der Story weiterbringen (Dropping the Fourth Wall)

  Hierbei handelt es sich um eine Kombination aus einer Veränderung der Umwelt und einer Zieleingabe für den Anwender. Dem Anwender wird die Entscheidungsmöglichkeit abgenommen, z.B. indem er automatisch eine Szene weitergebracht wird.

- Veränderung der Persönlichkeit (Shifting Personalities)

  Andere Aktoren bekommen eine Verhaltensänderung, die wiederum den Anwender beeinflussen soll. Nach Crawford ist dies die eleganteste Ablaufoptimierung.

- Zeitbegrenzung (The Ticking Clock of Doom)

  Der Anwender wird mit Aktionen oder Szenen konfrontiert, die in einer gewissen Zeit abgeschlossen sein müssen, so dass der Anwender gezwungen wird zu reagieren.

Chris Crawford betrachtet hierbei nur die Sichtweise des Story-Autors. Weiterhin werden die Möglichkeiten von zeitlicher Steuerung nur angerissen. Crawford geht mit folgender Begründung davon aus, dass Interactive Storytelling nicht in Echtzeit betrieben werden kann [1]:

> „(...) a story world is composed of closely balanced decisions that could reasonably go either way. These decisions require thought from players; they cannot be made in split second."

Natürlich müssen die Zeiten, die ein Anwender zum Treffen von Entscheidungen braucht, berücksichtigt werden. Dies kann jedoch auch vom Drama Manager übernommen werden, indem die Zeit, die der Anwender für Entscheidungen benötigt, an anderer Stelle wieder eingespart wird oder ihm die Entscheidung ganz abgenommen wird. Bestimmte Funktionalitäten der Ablaufoptimierung, vor allem der zeitlichen Steuerung, könnte man jedoch auch direkt in die Hände des Anwenders legen und ihm somit die Kontrolle über einen Teil der Story-Präsentation geben, die er auf seine eigenen Bedürfnisse anpassen kann.

Die Zeit, innerhalb der eine Story präsentiert wird und vor allem wie viel Inhalt während dieser Zeit vermittelt wird, ist eine nicht zu vernachlässigende Komponente. So sollte ein Anwender selber entscheiden können, wie lange und in welcher Komplexität er sich mit dem Interactive Storytelling-Erlebnis auseinander setzen möchte. Das individuelle Flow-Erlebnis eines jeden Anwenders kann hierdurch stark beeinträchtigt werden. Auch Brian Magerko stellt in seiner „Interactive Drama Architecture" sowohl Inhalt als auch Zeit als wichtiges Kriterium dar und beschreibt diese als die zwei Achsen der Plotdefinition [24].

Darauf aufbauend wird im Weiteren das Eingreifen in den Story-Verlauf in inhaltsbezogene und zeitgesteuerte Konzepte unterteilt. Zwar gäbe es noch weitere Möglichkeiten der Kategorisierung, z.B. eine Einteilung in Konzepte, die

den Anwender beeinflussen oder Konzepte, die computergesteuerte Aktoren beeinflussen. Diese und ähnliche Unterteilungen stellen in der bisher vorgenommenen Kategorisierung jedoch keine eigenständigen Kategorien dar, da sie in allen bisher vorgestellten Unterteilungen angewandt werden können und daher als eine Eigenschaft eines Konzeptes angesehen werden. Auch eine Unterteilung in Konzepte, die der Anwender und jene, die der Story-Autor verwenden kann, macht wenig Sinn. Dies ist eine Geschmacksfrage, die von Autor zu Autor variiert und weitere Anforderungen an die Abspiel-Komponente stellt, die dem Anwender die notwendigen Schnittstellen zur Kontrolle des Story-Verlaufs anbieten müsste.

4.3 Messen und Überwachen („Hören")

Wie schon erwähnt, sind die Möglichkeiten einer Überwachung des Story-Verlaufs überschaubar. Für eine Ablaufoptimierung ergeben sich drei Verwendungszwecke:

- Ereignisse im Story-Verlauf überwachen, um sie bei der Entscheidungsfindung zu berücksichtigen

  Die Erkennung der Notwendigkeit eines Eingriffes in den Story-Verlauf ist an Bedingungen geknüpft, die im Zusammenhang mit den Ereignissen in der Story und somit den Aktionen des Anwenders stehen. Hierfür kann die in Kapitel 3.4 erwähnte Observer-Schnittstelle verwendet werden. Die Überwachung beschränkt sich auf die Benachrichtigung angemeldeter Observer über eingetretene Ereignisse in der Story. Die Bewertung des Ereignisses und somit des Messergebnisses, wird in den jeweiligen Observern vorgenommen, die normalerweise schon der Entscheidungsfindung zugehörig sind.

- Aufzeichnung des Story-Verlaufs zur späteren Auswertung

  Zur Analyse unterschiedlicher Story-Durchläufe, z.B. in einem Test-Labor mit unterschiedlichen Probanden, werden alle eintretenden Story-Ereignisse aufgezeichnet und zur späteren Verwendung gespeichert. Aus diesen Daten lassen sich Statistiken zusammenstellen, die es dem Story-Autor ermöglichen, aus dem konkreten, aufgezeichneten Verhalten der Anwender, Rückschlüsse auf ihr Flow-Erlebnis zu ziehen.

- Speichern bestimmter Messwerte in Variablen zur Abfrage in Bedingungen

  Einige Bedingungen bestehen nur daraus, Variablen auf bestimmte Wertebereiche hin zu überprüfen (siehe 4.4.1). Deshalb sollten bestimmte Messwerte bei Eintritt des entsprechenden Ereignisses automatisch in Variablen gespeichert werden. Hierzu gehören vor allem spezielle Variablen, die vom Story-Autor nur indirekt beeinflusst werden können, wie z.B. der Name oder die bisherige Laufzeit der aktuellen Szene oder Story.

4.4 Entscheidungsfindung („Denken")

4.4.1 Einfache Abfragemechanismen

Bei diesen Konzepten handelt es sich um einfache Bedingungen, die in if/then/else-Blöcken definiert werden können. Die verwendeten Werte zur Abfrage und Beeinflussung betreffen vor allem den aktuellen Story-Status und die Story-Statistik. Der Story-Status setzt sich sowohl aus den vom Autor verwendeten Variablen als auch der Position des Anwenders in der ablaufenden Story zusammen. Die Story-Statistik besteht aus den gesammelten Daten vorheriger Story-Durchläufe, wodurch Vergleiche zwischen vorherigen und dem aktuellen Status angestellt werden können. Aus diesen Messungen und Vergleichen können Kennwerte berechnet werden, um bei Erreichung bestimmter Schwellenwerte, möglicherweise unter Berücksichtigung eines definierten Wertebereiches, einen Eingriff auszulösen (vgl. Magerko [15]). Für diese vergleichsweise simplen Berechnungen und Abfragen bietet sich am ehesten die Verwendung von Skriptsprachen an. Diese müssten dann allerdings auch Zugriff auf Story-Status und Story-Statistik haben.

4.4.2 Voraussage und Erkennung von Verhaltensweisen

Dieser Kategorie gehören komplexe Konzepte an, die sich nur schwer durch Bedingungen ausdrücken lassen und zumeist aufwändige Algorithmen anwenden. Die Kernfragen, die sich bezogen auf die Ablaufoptimierung stellt, ist: „Wie wahrscheinlich ist es, dass das Verhalten des Anwenders den Story-Ablauf gefährdet" (Voraussage) und im Folgeschluss: „Hat das Verhalten des Anwenders den Story-Verlauf gefährdet" (Erkennung) [15].

Die Voraussage und Erkennung von Verhaltensmustern ist dem Aufgabengebiet der Künstlichen Intelligenz zuzuordnen. Mustererkennung ist hierbei eine übliche Problemstellung, die sich mit verschiedenen Techniken realisieren lässt. Allen Techniken ist gemein, dass sie einen großen Datenbestand als Vergleichs- oder Trainingsmöglichkeit benötigen. Das Sammeln dieser Daten wird als Datamining bezeichnet, in diesem Fall handelt es sich um die Daten aus vorherigen Story-Durchläufen, die während der Überwachung einer Story aufgezeichnet werden [25], [26].

Am weitesten verbreitet ist wohl die Realisierung über Neuronale Netze. Nach vom Story-Autor bestimmten Kriterien trainiert, können diese Verhaltensmuster nicht nur erkennen, sondern auch vorhersagen. Ein möglicher Anwendungsfall könnte sein, ein Neuronales Netz auf einen vom Story-Autor als optimal angesehenen Story-Ablauf zu trainieren. Während des Ablaufs der Story könnte ein solches Netz Abweichungen erkennen, woraufhin eine beliebige Gegenmaßnahme ergriffen werden könnte. Diese Vorgehensweise ähnelt einem „Ghost Car" in Computerrennspielen, bei dem während des Rennens ein zweiter Fahrer auf dem Bildschirm dargestellt wird, der den vorherigen Rennverlauf desselben oder eines anderen Anwenders anzeigt. Der Unterschied ist,

dass in diesem Fall eine Anpassung durch den Computer und nicht durch den Anwender erfolgt [27].

Weiterhin wäre vorstellbar, dass ein Neuronales Netz direkt auf verschiedene Verhaltensweisen des Anwenders trainiert wird, so dass es eher unscharfe Begriffe wie z.B. „hilflos" oder „draufgängerisch" erkennen kann. Mit auf solche Weise trainierten Neuronalen Netzen könnte man versuchen, die aktuelle „Gameplay Gestalt" des Anwenders zu erkennen und die „Narrative Gestalt" daraufhin anpassen [20].

Neuronale Netze sind aber, wie schon erwähnt, nicht die einzige Möglichkeit Verhaltensweisen zu erkennen oder zu modellieren. Magerko verwendet in seiner „Interactive Drama Architecture" z.B. eine Art Experten-System. Er definiert hierbei ein User-Modell, das sich aus Regelmechanismen und Wahrscheinlichkeitsangaben zusammensetzt. Anhand dieser Daten wird ermittelt, wie akzeptabel das Verhalten des Anwenders im Vergleich zu den Vorgaben des Autors ist. Sowohl das User-Modell als auch die Vorgaben des Autors werden als eine Wissensdatenbank verwendet, die dann auf die Eingaben des Anwenders angewandt wird [15].

Die Auswahl an Umsetzungsmöglichkeiten einer Mustererkennung ist groß. Welche Techniken sich in welcher Form am besten eignen, muss zuvor in Testlaboren mit Probanden evaluiert werden. So könnte es sich herausstellen, dass aufgrund unzureichendem Datamining nur eine gewisse Anzahl an Verhaltensmustern vom System erkennbar ist und bei einer Überschreitung dieses Limits keine klare Abgrenzung der Verhaltensweisen mehr möglich ist. Evtl. könnte dieses Problem lösbar sein, indem man mehrere Verhaltenszustände zulässt und diese, wie in der Fuzzy-Logic, in mehrere Qualitätsstufen unterteilt. Ein Anwender könnte somit „ein bisschen ängstlich", gleichzeitig aber auch „überdurchschnittlich mutig" sein. Die Probleme, die bei der Verwendung solch komplexer Mechanismen entstehen können, sind vielfältig und ohne ausführliche Tests nicht absehbar. Im Rahmen dieser Diplomarbeit stehen solche Möglichkeiten nicht zur Verfügung. Trotzdem wird versucht, bei der Deklaration der Schnittstelle für Konzepte der Entscheidungsfindung darauf zu achten, dass komplexere Verfahren zu einem späteren Zeitpunkt Anwendung finden können [25], [26].

4.5 Eingriff in den Storyverlauf („Sprechen")

4.5.1 Zeitgesteuerte Konzepte („StoryPacing")

4.5.1.1 Zeitkontrolle

Für bestimmte Anwendungsfälle kann es von Nutzen sein, die Ausführung der Story zeitlich zu begrenzen. Hierbei ist nicht die Zeit gemeint, die innerhalb der Story vergeht. Hier kann theoretisch von einer Szene zur nächsten ein Jahr vergehen, wenn dies von der Dramaturgie so erwünscht ist. Diese Art der zeit-

lichen Steuerung ist eine inhaltliche und wird somit vollständig vom Story-Autor gesteuert [17].

Eine zeitgesteuerte Ablaufoptimierung betrachtet nur die Zeit, die ein Anwender beim Erleben der Story verbringt. Unabhängig davon, wie schnell die Zeit innerhalb der Story vergeht, könnte diese Präsentationsgeschwindigkeit für den Anwender z.B. beschleunigt oder verlangsamt werden. Dies ist auch ein gutes Beispiel für ein Konzept, dass durch den Anwender selbst kontrolliert werden kann, so dass er, ähnlich wie bei einem Videoabspielgerät, selber entscheidet, in welcher Geschwindigkeit er die Story erlebt.

Der von Chris Crawford angedachte Ansatz beschreitet den entgegengesetzten Weg: Die Ereignisse innerhalb einer Story laufen unabhängig von den Aktionen des Spielers ab (siehe „The Ticking Clock of Doom in Kapitel 4.2). Eine solche Zeitkontrolle kann sowohl lokal als auch global erfolgen: Nur in einer bestimmten Szene (Zeitdruck für den Moment), oder für alle Szenen (Zeitdruck während der gesamten Story). Dieses Konzept bringt die Gefahr mit sich, dass der Anwender bestimmte Ereignisse der Story verpasst, da sie auch geschehen, ohne dass der Anwender die entsprechende Szene oder den entsprechenden Story-Status erreicht hat.

Um diese Gefahr zu entschärfen, könnte dem Anwender eine Zeitanzeige zur Verfügung gestellt werden, die angibt, wie viel Zeit bis zum nächsten wichtigen Ereignis bleibt oder um die Story noch erfolgreich abzuschließen. Die Zeitanzeige könnte anhand der Angaben des Story-Autors und der aus aufgezeichneten Statistiken berechneten Durchschnittszeit der bisherigen Story-Durchläufe ermittelt werden. Dies würde dem Anwender eine Vorstellung davon geben, wie „schnell" er in der Story im Vergleich zu vorherigen Durchläufen voranschreitet, und zu welchem Zeitpunkt ein weiteres Ereignis eintritt. Mit diesem Konzept würde es aber nahezu vollständig dem Anwender überlassen bleiben, ob bzw. in welcher Form er ein bestimmtes Ereignis wahrnehmen möchte. Der Eingriff in die Story geht vom Anwender aus, so dass nicht gewährleistet ist, dass sich ihm die Botschaft der Story vollständig erschließt.

4.5.1.2 Priorisierung

Auch wenn das Ziel bei Interactive Storytelling sein soll, dem Anwender möglichst viele Freiheiten zu bieten, so sollte dies nicht auf Kosten der Botschaft gehen, die der Story innewohnt (siehe Kapitel 2.3). Ereignisse, die diese Botschaft vermitteln, sollten vom Anwender also in jedem Fall erlebt werden. Der Story-Autor benötigt hierfür eine Möglichkeit, das Eintreten dieser Ereignisse zu forcieren, ohne dass der Anwender sie verpasst. Hierfür müsste eine Ablaufoptimierung erkennen, auf welchen Wegen der Anwender zu einem bestimmten Ereignis gelangen kann, also Kenntnisse über das Transitionsnetzwerk des Story-Graphen haben. Mit diesem Wissen könnte die verbleibende Zeit bis zum Eintreten des gewünschten Ereignisses berechnet werden. Wenn nötig werden daraufhin Aktionen ausgelöst, die den Anwender weiter in der Story vorantreiben und somit näher an das gewünschte Ereignis heranführen.

Um herauszufinden, ob ein Eingriff notwendig ist, müssen die für die Botschaft der Story wichtigen Ereignisse, bzw. die Szenen in denen sie vorkommen, auf eine bestimmte Art und Weise markiert werden. Die Szenen bekommen also eine Priorität zugeordnet, anhand der man ihre Wichtigkeit für den Story-Ablauf ablesen kann. Dies fällt in den Aufgabenbereich der Entscheidungsfindung. Die Priorität und deren Abfrage kann also auf einfachen (z.B. eine vom Story-Autor definierte Konstante) oder komplexen Mechanismen (siehe oben, Berechnung anhand verbleibender Zeit) beruhen. Der Vergleich der Prioritäten findet szenenübergreifend statt und muss somit global während des gesamten Story-Ablaufs erfolgen.

4.5.2 Inhaltsbezogene Konzepte

Wie schon bei der Konzeptkategorisierung in Kapitel 4.2 aufgeführt, handelt es sich hierbei um Konzepte, die tatsächlich den Inhalt der Story ändern. Bis auf die Zeitbegrenzung gehören alle von Chris Crawford genannten Konzepte in diese Kategorie. Bei genauerer Betrachtung der von Chris Crawford vorgenommenen Kategorisierung fällt auf, dass sich die Konzepte, die Zeitbegrenzung außer acht gelassen, nur in ihrer Manifestation unterscheiden (Eingriff in die Umwelt, in die Persönlichkeit anderer Aktoren usw.), sich in ihrer Zielsetzung jedoch überschneiden: Der Anwender soll entweder zu einer Aktion verleitet oder davon abgehalten werden. In beiden Fällen kann er entweder einen Hinweis bekommen, der ihn weiter bringt oder mit ihn blockierenden Maßnahmen aufgehalten werden. Diese etwas allgemeinere Sichtweise vertritt auch Andrew Glassner, der sie als „Bridging and Blocking" bezeichnet [5]:

- Create a Bridge / Eine Brücke bauen

  Dies verschafft dem Anwender eine neue Handlungsmöglichkeit, z.B. indem, bildlich gesprochen, eine Brücke über einen Fluss geschaffen wird (Veränderung der Umwelt), oder indem ein vom Computer gesteuerter Aktor den Anwender mit einer Aufgabe versieht (Veränderung der Persönlichkeit eines Aktors).

- Remove a Bridge / Eine Brücke entfernen

  Nimmt dem Anwender eine Handlungsmöglichkeit. Hiermit lassen sich zuvor gegebene Optionen wieder rückgängig machen, z.B. um den Anwender zu zwingen, einen anderen Lösungsweg zu finden, der ihm bisher entgangen war oder um ihn zu verlangsamen, da er ansonsten Gefahr läuft, ein wichtiges Ereignis zu verpassen.

- Create a Block / Ein Hindernis erzeugen

  Wie auch „Remove a Bridge" soll dies den Anwender an bestimmten Tätigkeiten hindern. Hierbei wird allerdings ein Hindernis erzeugt, anstatt den Anwender eine zuvor vorhandene Handlungsmöglichkeit zu nehmen.

- Remove a Block / Ein Hindernis entfernen

  Hiermit können den Anwender blockierende Hindernisse wieder entfernt werden. Dies kann aufgrund von Aufgaben, die der Anwender erledigt hat, geschehen oder weil erkannt wurde, dass der Anwender durch dieses Hindernis im Story-Verlauf nicht weiterkommt.

Glassner betont, dass es sich hierbei nur um eine allgemeine Sichtweise auf mögliche Eingriffsmöglichkeiten und kein festes Kategorisierungs-Schema handelt. Tatsächlich handelt es sich bei den 4 genannten Aktionsmöglichkeiten um Elemente, die in jeder Story, unabhängig von einer Ablaufoptimierung, verwendet werden, um eine spannende und für den Anwender interessante Story zu erstellen. Eine Ablaufoptimierung, die möglichst unauffällige Eingriffe vornimmt, sollte sich aber genau dieser Mittel bedienen, um keine Störfaktoren zu erzeugen, die das Flow-Erlebnis des Anwenders negativ beeinflussen könnten [5].

Die beschriebenen Elemente arbeiten grundsätzlich auf der Story-Ebene. So könnten z.B. physikalische Veränderungen in der Umwelt oder auch das Geben von Hinweisen, in speziell dafür vorgesehenen Szenen geschehen. Inhaltsbezogene Konzepte zur Ablaufoptimierung werden idealerweise durch das Auslösen von Ereignissen wie z.B. Übergänge von einer Szene in einer andere oder die Durchführung spezifischer Aktionen innerhalb einer Szene, realisiert. Daher muss es möglich sein, die Bedingungen für eine Auslösung dieser Ereignisse lokal zu überprüfen und auch dementsprechend im Story-Modell zu definieren.

4.6 Verfahrensherleitung

Um einen befriedigenden Ablauf der Story nicht zu gefährden, muss die Ablaufoptimierung den Anwender zu einem bestimmten Verhalten bringen oder ihn davon abhalten. Ein derartiger Eingriff soll nach Möglichkeit nicht sein Flow-Erlebnis unterbrechen, sondern im Gegenteil versuchen, dieses durch den Eingriff noch zu verstärken. Daher sollten Eingriffe subtil und so selten wie möglich vorkommen. Sowohl die Bedingungen als auch die darauf folgenden Eingriffe können vielfältig und nahezu beliebig miteinander kombinierbar sein.

Magerko verwendet in seiner „Interactive Drama Architecture" keine expliziten Definitionen von Eingriffen bzw. Gegenmaßnahmen, stattdessen werden Entscheidungen aufgrund von Vorgaben der Story-Definition und den Regeln in seinem User-Modell angewandt. Diese Vorgaben und Regeln werden auf drei Fälle überprüft [15]:

- Ist die Wahrscheinlichkeit hoch, dass die Story-Vorgaben des Autors zum aktuellen Zeitpunkt nicht mehr erreicht werden können?
- Ist der erfolgreiche Abschluss der Story im aktuellen Stadium nicht mehr möglich?
- Gibt es zusätzliche Vorgaben und Bedingungen, die zum aktuellen Zeitpunkt schon erfüllt sein sollten?

Diese drei Erkennungsstufen sollen es dem Story-Autor ermöglichen, unterschiedlich auf die jeweilige Situation reagieren zu können. Die Wahrscheinlichkeit, dass eine Bedingung in naher Zukunft eintreten wird, ist im Vergleich zum tatsächlichen Eintritt als weniger kritisch anzusehen. Weiterhin ist die Nicht-Erfüllung bestimmter Story-Vorgaben, im Vergleich zu Bedingungen, die darauf hinweisen, dass ein erfolgreicher Abschluss der Story nicht mehr möglich ist, zu vernachlässigen. Die Prioritäten der Bedingungen müssen hierbei vom Autor angegeben werden. Im Idealfall kommt es nur zur Auslösung der ersten Erkennungsstufe, die bei erhöhter Wahrscheinlichkeit Gegenmaßnahmen ergreift, die ein Eintreten der anderen beiden Erkennungsstufen verhindern [15].

Eine solche Unterscheidung und Priorisierung von Erkennungsstufen ist sehr nützlich, um verschiedene Schweregrade von Eingriffssituationen definieren zu können und sollte von einem Verfahren zur Ablaufoptimierung unterstützt werden. Genauso sollte es, anders als in Magerkos Modell, auch möglich sein, genau zu definieren, welche Gegenmaßnahmen aufgrund welcher Vorgaben ergriffen werden sollen. Die vorgegebenen Erkennungsstufen sind nicht ausreichend, um jede von einem Story-Autor erdenkliche Eingriffsmöglichkeit abzudecken.

Ein Verfahren zur Ablaufoptimierung sollte weiterhin Strukturen zur Verfügung stellen, die eine Kombination aus verschiedenen Bedingungen/Vorgaben und Eingriffen/Gegenmaßnahmen zusammenfasst. Eine solche Struktur könnte man als eine Strategie bezeichnen. Eine vom Story-Autor definierte Strategie setzt sich somit aus einem oder mehreren Konzepten der Kategorien Entscheidungsfindung und Eingriff zusammen. Mithilfe der Strategien findet eine Überwachung der Story statt, indem die enthaltenen Konzepte der Entscheidungsfindung fortwährend ausgeführt werden. Diese sollten sich sowohl über aktuelle Ereignisse im Story-Verlauf informieren können als auch Zugriff auf die Story-Statistik haben, um dies bei ihrer Entscheidungsfindung berücksichtigen zu können. Hierbei ist darauf zu achten, dass nicht jede Strategie dafür geeignet ist, die gesamte Story zu überwachen. Wie in den vorherigen Kapiteln erwähnt, gibt es Konzepte, die sich sowohl global als auch lokal anwenden lassen oder sogar auf eine lokale Anwendung beschränkt sind. Das bedeutet, dass bestimmte Strategien nur zu bestimmten Zeitpunkten oder in bestimmten Situationen ausgeführt werden können oder sollen. Diese Situationen können jedoch wiederum durch andere Strategien hervorgerufen werden.

Um wirklich eine Vielzahl an unterschiedlichen Konzepten unterstützen zu können, muss nicht nur deren Kombinierbarkeit, sondern auch deren Implementierungsunabhängigkeit gewährleistet sein. Eine Entscheidungsfindung, die komplizierte Verhaltensmustererkennung in C++-Code vornimmt, sollte mit einem über eine Skriptsprache realisierten Eingriff kombinierbar sein. Die unterschiedlichen Implementierungen benötigen wiederum eine einheitliche Schnittstelle, um auf Story-Modell, -Statistik und -Verlauf zuzugreifen. Eingriffe, die eine inhaltliche Veränderung vornehmen, müssen diese auch über die Möglichkeiten, die durch die Story-Definition abgedeckt werden, durchführen

können. Die Nutzung vorhandener Story-Strukturen vermeidet weiterhin Redundanzen bei der Definition von Strategien.

4.7 Zusammenfassung

Die möglichen Konzepte einer Ablaufoptimierung sind vielfältig, und eine Abdeckung aller Fälle nicht zu realisieren, zumal viele Konzepte auch story-spezifisch sein können. Trotzdem lassen sich bei einer Kategorisierung der Konzepte Gemeinsamkeiten finden, aus denen sich ein Verfahren herleiten lässt, das eine Kombination unterschiedlicher Konzepte möglich macht.

Die vorgestellten Konzepte wurden in drei Hauptbereiche aufgeteilt, die sich weitestgehend miteinander kombinieren lassen. Mit dem beschriebenen Verfahren lassen sich konkrete Kombinationen aus Konzepten zu Strategien zusammenfassen, die auf eine Story angewandt werden können. Dieses Verfahren stellt einige Anforderungen an ein Interactive Storytelling-System, die genau analysiert werden müssen, um eine Software-Architektur zu entwickeln, die dem gerecht wird.

5 Anforderungsanalyse

5.1 Einleitung

Das entwickelte Verfahren stellt bestimmte Anforderungen an die Architektur des NarrationController als Interactive Storytelling-System und ICML als Definitionssprache für die Story und dessen Plot. Im weiteren Verlauf wird eine Analyse und Vorstellung dieser Anforderungen vorgenommen. Neben den Verfahrens-Anforderungen gibt es außerdem externe Anforderungen, die hier nun zusammenfassend präsentiert werden.

5.2 Anforderungen an ICML

- Definition neuer ICML-Elemente

  Die Strategien zur Ablaufoptimierung sollen vom Story-Autor als Kombination aus Bedingungen und Gegenmaßnahmen definiert werden können, wobei diese Definitionen implementierungsunabhängig sein sollen. Die Definitionssprache XML ist bekannt dafür, dass sie implementierungsunabhängig arbeitet [28], wodurch sich der bisher schon verwendete XML-Dialekt ICML hervorragend für eine Erweiterung um Strategie-Definitionen eignet. Die hierfür notwendigen ICML-Elemente sind zu definieren und der Markup-Language hinzuzufügen, wobei auf Gemeinsamkeiten mit bisherigen Elementen zu achten ist, um diese evtl. zu einer einheitlichen Struktur zusammenzufassen.

- Trennung von Definition und Nutzung der Strategien

  Um eine hohe Wiederverwendbarkeit zu erreichen, sollen die vom Story-Autor definierten Strategien zur Ablaufoptimierung nach Möglichkeit in anderen Stories und Szenarien anwendbar sein. Um dieses zu erreichen, muss es für den Autor die Möglichkeit geben, die definierten Strategien pro Szenario auszuwählen.

- Speicherung von ICML-fremden XML-Daten

  Um es dem Story-Autor zu ermöglichen, eigene Erweiterungen einer Story-Definition oder Beschreibung von Bedingungen und Aktionen vorzunehmen, sollen auch ICML-fremde XML-Daten in der ICML-Datei gespeichert werden können. Eine Kollision mit ICML-konformen Elementen ist hierbei zu vermeiden, da dies zu invalidem ICML führen würde. Um dies zu überprüfen, soll die Möglichkeit der XML Schema-Validierung von ICML gegeben sein.

- Zugriff auf Skriptsprache(n) in ICML

  Um komplexere Abfragen und Berechnungen durchführen zu können, sollte ICML den einfachen Zugriff auf eine oder mehrere Skriptsprachen erlauben.

5.3 Anforderungen an den NarrationController

Die "Drama Manager"-Funktionalität des NarrationControllers hat die Aufgabe, die Vorgaben des Story-Autors bei dem Erleben der Geschichte einzuhalten. Um dieses zu ermöglichen, werden die folgenden Funktionalitäten benötigt:

- Überwachung des Story-Verlaufs

  Sowohl während, als auch nach dem Abspielvorgang sollen die im Story-Verlauf eintretenden Ereignisse ausgewertet werden können. Während des Abspielvorgangs sollen die Informationen dazu dienen, die Vorgaben des Story-Autors zu überprüfen und einzuhalten. Nach Beendigung der Anwendung sollen die Daten eine statistische Auswertung ermöglichen, um z.B. eine Analyse des Anwenderverhaltens vorzunehmen. Der Story-Autor soll somit schon während der Entwicklung der Story ein Feedback von potentiellen Anwendern erhalten können. Hierfür ist es notwendig, dass unterschiedliche, voneinander unabhängige Überwachungs-Instanzen über jedes Ereignis informiert werden. Hierbei soll es sich auch um externe Komponenten handeln können. Weiterhin ist zu beachten, dass nicht jede Überwachungs-Instanz über jedes Ereignis informiert werden möchte. Die vorhandene Version des NarrationControllers verfügt bereits über Basis-Funktionalitäten in diesem Bereich. Diese sollen auf ihre Gebrauchstauglichkeit und Erweiterbarkeit hinsichtlich der genannten Punkte geprüft und notwendige Änderungen vorgenommen werden.

- Überprüfung der Vorgaben des Story-Autors

  Die in ICML definierten Vorgaben des Story-Autors müssen während des Abspielvorgangs überprüft werden. Komplexere Berechnungen, wie z.B. die Erkennung des Anwenderverhaltens, die sich nicht zufriedenstellend in einer Definitionssprache wie ICML abbilden lassen, müssen dem NarrationController auf eine andere Weise vermittelt werden können. Es ist wünschenswert, dass der NarrationController um eine Plugin-ähnliche Architektur erweitert wird, die das Hinzufügen von komplexen Mechanismen erlaubt und somit für eine hohe Erweiter- und Austauschbarkeit sorgt.

- Ausführen der vom Story-Autor definierten Gegenmaßnahmen

  Bei Erfüllung der an die Vorgaben geknüpften Bedingungen ist ein Eingriff in den Story-Verlauf durchzuführen. Die Art des Eingriffes ist in ICML definiert, die eigentliche Implementierung muss jedoch an anderer Stelle erfolgen und dem NarrationController bekannt gemacht werden können. Wie auch bei der Überprüfung der Vorgaben soll eine Plugin-ähnliche Architektur zur Verfügung gestellt werden, um dem NarrationController beliebige Eingriffsmaßnahmen hinzufügen zu können.

- Interpretation neuer ICML-Elemente

  Die neuen Elemente, um die ICML erweitert wird (siehe 5.2), müssen vom NarrationController eingelesen und interpretiert werden können.

- Einhaltung allgemeiner Paradigmen der objektorientierten Software-Entwicklung

  Gemeinsamkeiten zwischen bestehenden Funktionalitäten und den neuen Anforderungen sollen verallgemeinert werden, um Redundanz zu vermeiden und eine bessere Wartbarkeit und Wiederverwendbarkeit zu gewährleisten, wobei eine möglichst offene und erweiterbare Architektur entstehen soll.

  Da der NarrationController in verschiedenen Einsatzgebieten zur Anwendung kommen soll, ist weiterhin darauf zu achten, dass der NarrationController unter verschiedenen Ausführungsbedingungen verwendet werden kann (z.B. als Standalone-Applikation oder Plugin).

  In dieser Hinsicht ist auch auf die Portabilität der Anwendung zu achten. Zum einen sollte der NarrationController sowohl unter Visual Studio 7.1, als auch mit Visual Studio 8 kompilieren, um ihn in unterschiedlichen Projektumgebungen verwenden zu können. Neben dieser Compilerunabhängigkeit ist weiterhin eine Plattformunabhängigkeit erwünscht, um auch in der Wahl des Betriebssystems flexibel sein zu können.

5.4 Zusammenfassung

Die Anforderungen an den NarrationController und das verwendete Datenmodell ICML rücken vor allem die Punkte Erweiterbarkeit, Wartbarkeit und Wiederverwendbarkeit in den Vordergrund. Neben dem eigentlichen Architektur-Design der neuen Funktionalitäten sind auch die bisherigen daraufhin zu überprüfen und gegebenenfalls in einem Refactoring anzupassen.

6 Architektur-Design und Refactoring

6.1 Einleitung

Anhand der gestellten Anforderungen an den NarrationController wird ein Architektur-Design entworfen und implementiert. Neben den zusätzlichen Funktionalitäten, die eine Erweiterung um eine Ablaufoptimierung mit sich bringen, gehört es ebenso zu den Anforderungen das bestehende Architektur-Design auf Schwachstellen zu überprüfen und eine möglichst hohe Wiederverwendbarkeit und Austauschbarkeit unter den einzelnen Komponenten zu gewährleisten. Mit dem Entwurf der neuen Software-Architektur wird daher gleichzeitig ein Refactoring des bestehenden Codes durchgeführt, welches für mehr Übersichtlichkeit und bessere Code-Qualität sorgen soll. Bei der Umstrukturierung des Codes werden vor allem Design-Patterns angewandt, um von den Erfahrungen anderer Programmierer zu profitieren [29], [30], [31], [32].

6.2 Allgemeine Richtlinien zur Dokumentation des Architektur-Designs

Alle verwendeten Design-Patterns in aller Ausführlichkeit zu erklären ist im Rahmen dieser Diplomarbeit nicht möglich. Zur Verdeutlichung findet sich jedoch eine Übersicht der verwendeten Patterns im Glossar.

Das Architektur-Design wird mit Hilfe von UML 2.0-Diagrammen und XML Schema-Darstellungen erklärt. Die Bedeutung der in den Abbildungen verwendeten Symbole sind in den Quellen [33] und [34] erklärt.

Allen UML-Abbildungen ist gemein, dass sie grundsätzlich nur die für die jeweilige Erklärung der Vorgehensweise wichtigen Komponenten darstellen. Bereits in Oberklassen deklarierte Methoden werden im Allgemeinen in den abgeleiteten Klassen nicht wieder aufgeführt, es sei denn sie nehmen eine besondere Stellung in der jeweiligen Abbildung ein. Eine umfassende Dokumentation der Architektur befindet sich auf der beiliegenden CD (siehe Anhang 2).

Komponenten-, Klassen- und Schnittstellen werden in der Folgenden Beschreibung in der selben Notation wie im Quellcode angegeben. Sie fangen immer mit einem Großbuchstaben an und auf jedes neue Wort in der Bezeichnung folgt ein weiterer Großbuchstabe. Beispiel: StoryVisitor, nicht Storyvisitor.

6.3 Überwachung der Story

6.3.1 Observer

Wie in Kapitel 5.3 gefordert, sollen sich verschiedene Überwachungs-Instanzen über eintretende Ereignisse in der Story informieren lassen können. Damit dies

möglich ist, müssen sich die einzelnen Überwachungs-Instanzen beim NarrationController anmelden, damit dieser die an der Überwachung der Story-Ereignisse interessierten Instanzen kennt und sie beim Eintritt des jeweiligen Ereignisses informieren kann.

Ein solches Verfahren ist eine Standardanforderung an die meisten Softwaresysteme. Dies hat dazu geführt, dass ein Design-Pattern entwickelt wurde, welches sich dieses Problems annimmt. Es handelt sich um das zuvor schon erwähnte Observer-Pattern. Dieses ist weit verbreitet und findet als Bestandteil des Model-View-Controller-Patterns Anwendung in einem Großteil von GUI-Bibliotheken wie z.B. in Java Swing.

In der vorliegenden NarrationController-Version existieren bereits drei Observer-Implementationen, namentlich MMDCInterface, ActionInterface und ExecutionListener (vgl. Kapitel 3.4). Sie erfüllen genau den Zweck, externe Komponenten über unterschiedliche Ereignisse im Story-Verlauf zu informieren. Die drei vorhandenen Implementationen haben bis auf die gleiche Funktionsweise (sie verwenden dasselbe Design-Pattern) keine Gemeinsamkeiten, unterscheiden sich also nur in ihrer Schnittstellen-Definition. Jede Schnittstelle informiert hierbei über andere Ereignisse. Dies ist eine häufig angewandte Vorgehensweise: Das Design-Pattern gibt nur die Architektur vor, nicht die konkrete Implementierung.

Wenn weitere Möglichkeit zur Überwachung der Story benötigt werden, die sich von den bestehenden unterscheidet, so müsste eine weitere Observer-Implementation für die neue Schnittstellendefinition vorgenommen werden. Dies wäre z.B. nötig, um eine Logging-Funktionalität zu implementieren, die für die Aufzeichnung von Ereignissen zur späteren Auswertung zuständig sein soll. Die schon vorhandene ExecutionListener-Schnittstelle bietet nicht genügend Informationen, um ein umfassendes Logging zu realisieren, da nur der Name des ausgeführten Elementes übergeben wird. Dadurch kommt man um die aufwändige Erstellung einer weiteren Observer-Implementation nicht herum.

Eine solche Vorgehensweise sorgt allerdings für Probleme, sobald sich etwas an der grundlegenden Verhaltensweise der sich ähnelnden Observer-Implementationen ändern soll. Notwendigen Änderungen am Code müssen an mehreren unterschiedlichen Stellen vorgenommen werden, was eine erhöhte Fehleranfälligkeit provoziert. Zum besseren Verständnis dieser Problematik sind an dieser Stelle nochmals die grundlegenden Entscheidungen, die bei der Implementation des Observer-Patterns zu treffen sind, beschrieben:

- Ein Subject bietet die Möglichkeit, sich von einem Observer auf Änderungen überwachen zu lassen. Hierfür müssen Methoden angeboten werden, über die sich beim Subject bekannte Observer-Typen an- und abmelden können. Daher wird das Subject manchmal auch als „Observable" bezeichnet. Die Verwaltung der angemeldeten Observer durch das Subject kann auf unterschiedliche Weise erfolgen (Verwendung einer Liste, Hashmap o.ä.).

- Bei einer Änderung der Daten im Subject setzt dieses alle bei ihm angemeldeten Observer darüber in Kenntnis. Hierfür benötigt das Subject eine entsprechende Methode (meist „notify" genannt), die wiederum eine Methode in den angemeldeten Observern aufruft (z.B. eine update()-Methode). Dieser Vorgang kann auf zwei Wegen realisiert werden: Zum einen kann das Subject den Observern die konkreten Änderungen direkt in der update()-Methode übergeben (push-Modell), oder aber der Observer hat Zugriff auf das Subject und kann sich innerhalb der update()-Methode die Daten, die für ihn von Belang sind, selbst besorgen (pull-Modell).
- Während ein Subject seine Observer über aktuelle Änderungen informiert, kann es vorkommen, dass sich bestimmte Observer beim Subject an- oder abmelden. Genauso können während dieser Benachrichtigung weitere Änderungen am Subject erfolgen, was einen erneuten Benachrichtigungsvorgang auslöst. Diese Fälle müssen von Observer und/oder Subject berücksichtigt werden, um Inkonsistenzen zu vermeiden, die schlimmstenfalls Abstürze oder Endlosschleifen hervorrufen würden.

Aufgrund dieser Vielfalt an Variationsmöglichkeiten die bei der Implementation des Observer-Patterns beachtet werden müssen, wäre es wünschenswert, wenn eine allgemeine Observer-Schnittstelle bereitgestellt werden würde, aus der sich beliebige Observer- und Subject-Typen ableiten ließen, wobei das genaue Verhalten spezifisch festlegbar sein sollte.

Tatsächlich gibt es diese Möglichkeit. Für die Lösung eines solchen Problems eignet sich die Verwendung von generischer Programmierung unter Zuhilfenahme eines auf Policies basierenden Designs, wie es Andrei Alexandrescu in seinem Buch „Modern C++ Design" beschreibt und wie folgt definiert [35]:

> „Verfahrensweisen (Policies) und Policy-Klassen unterstützen die Implementierung sicherer, effizienter und im hohen Maße anpassungsfähiger Designelemente. Eine Verfahrensweise definiert eine Klassenschnittstelle oder eine Klassen-Template-Schnittstelle. Die Schnittstelle besteht aus einem oder allen folgenden Elementen: innere Typdefinition, Member-Funktionen und Variablen"

Bei auf Policies basierendem Design werden also einzelne Funktionalitäten einer Klasse in eigenständige, unabhängige und somit austauschbare Verfahrensweisen eingeteilt, aus denen dann die namensgebenden Policy-Klassen entstehen. Die ursprüngliche Klasse, die die eigentlichen Funktionalitäten in sich vereinen soll, wird je nach individuellem Einsatzzweck mit verschiedenen Policy-Klassen ausgestattet. Üblicherweise kommt hierbei Mehrfachvererbung zum Einsatz, wobei die Policy-Klassen als Template-Parameter übergeben werden [35]. Die Angabe der zu erbenden Super-Klasse als Template-Parameter ist eine Vorgehensweise, die auch als Mixin bekannt ist [36].

Im C++ User Journal macht Andrei Alexandrescu einen Vorschlag für eine generische, auf Policies basierende Observer-Implementierung und geht dabei

auch auf die Probleme ein, die bei der Implementierung des Observer-Patterns entstehen können [37], [38]. Basierend auf den dort gemachten Vorschlägen wird im Weiteren eine eigene Implementierung eines auf Policies basierenden Observers vorgenommen. Grundlegend ist die Idee in Andrei Alexandrescu's Design, die einzelnen Policies der Subject-Klasse des Observer-Patterns hierarchisch in Form des Decorator-Patterns zu realisieren. Beim Decorator-Pattern wird zusätzliche Funktionalität durch einfache Vererbung realisiert. Zu beachten ist, dass die Vererbung wiederum über Template-Parameter und somit als Mixin erfolgt (siehe Listing 2).

```
template<class Subject>
class BaseSubscriptionSubject : public Subject
{
    ...
};
```

Listing 2: Beispiel einer generischen Vererbung

Das Decorator-Pattern findet also nicht wie sonst üblich zur Laufzeit, sondern zur Kompilier-Zeit der Applikation Anwendung. Die einzelnen Policy-Klassen werden dem Subject nicht auf einen Schlag durch Mehrfachvererbung hinzugefügt, sondern „dekorieren" das Subject Stück für Stück. Diese Vorgehensweise wurde gewählt, weil sowohl der An-/Abmelde-, als auch der Benachrichtigungs-Mechanismus stark voneinander abhängig sind. Die Mechanismen können sich, wie zuvor erwähnt, gegenseitig aufrufen, wobei sich die Art und Weise des Aufrufs der entsprechenden Methoden je nach Implementation unterscheiden kann. Man spricht auch davon, dass diese beiden Mechanismen nicht orthogonal zueinander sind. Orthogonalität ist bei Verwendung von Mehrfachvererbung jedoch ein wichtiges Kriterium, damit jede einzelne Policy über einen definierten Aufgabenbereich verfügt. Bei Überschneidungen im Aufgabenbereich ist es möglich, dass sich die Signaturen der Policies nicht unterscheiden, somit keine Eindeutigkeit mehr besitzen und folglich nicht mehr zugeordnet werden können. Bei geschickter Vorgehensweise lassen sich Abhängigkeiten zwischen verwandten, nicht zueinander orthogonalen Policies, ohne den Verlust der Eindeutigkeit realisieren. Doch im Falle der Verwobenheit der Observer-Funktionalitäten ist dieses mit zu viel Aufwand verbunden [35], [37], [38].

Durch die Anwendung des Decorator-Patterns lässt sich die Orthogonalität auf eine andere Weise hervorrufen: Da die Policies das Subject dekorieren, fügen sie dem Subject nach und nach weitere Funktionalitäten hinzu. Statt An-/Abmelde- und Benachrichtigungs-Funktionalität in eigenständige Policies aufzuteilen, müssen alle Policies die notwendigen Methoden der einzelnen Funktionalitäten implementieren. Dies wird über die Definition einer abstrakten BaseSubject-Klasse realisiert, die von den Policies dekoriert wird und somit vorgibt, welche Methoden die Policies zu implementieren haben. Die Verwendung des Decorator-Patterns bringt einige Besonderheiten mit sich, die es zu beachten gilt: Da die jeweiligen Methoden der zuletzt hinzugefügten Policy-Klassen in

der Vererbungs-Hierarchie zuerst aufgerufen werden, können später hinzugefügte Klassen die Funktionalitäten der höheren Hierarchieebenen mit nutzen. Dadurch spielt jedoch die Reihenfolge in der Policy-Klassen hinzugefügt werden, eine wichtige Rolle. Es ist vor allem darauf zu achten, dass alle Klassenmethoden ihre jeweilige Super-Methode aufrufen. Ansonsten würde die Funktionalität von Policy-Klassen höherer Hierarchieebenen wirkungslos gemacht werden.

Zunächst wird eine abstrakte BaseSubject-Klasse benötigt, die die zu verwendete Schnittstelle definiert, sowie eine Observer-Klasse, die sich bei BaseSubject anmelden kann (siehe Abbildung 6). Um auch hier möglichst flexibel zu sein, wird der Observer-Typ als Template-Parameter übergeben. Auf welche Weise die Observer vom Subject verwaltet werden, wird an dieser Stelle noch nicht definiert. Auch eine Entscheidung, ob das Pull- oder Push-Modell verwendet wird, soll hier noch nicht getroffen werden. Es wird nur eine Parameterübergabe bei Benachrichtigung in Form eines Events vorgesehen, welches wiederum als Template-Parameter individuell modifizierbar ist. Wenn ein Observer nach dem Pull-Modell betrieben wird, kann als Event eine leere Klasse verwendet werden, die von einem C++-Compiler üblicherweise nicht berücksichtigt wird, so dass beim Benachrichtigen keine Parameterübergabe erfolgt.

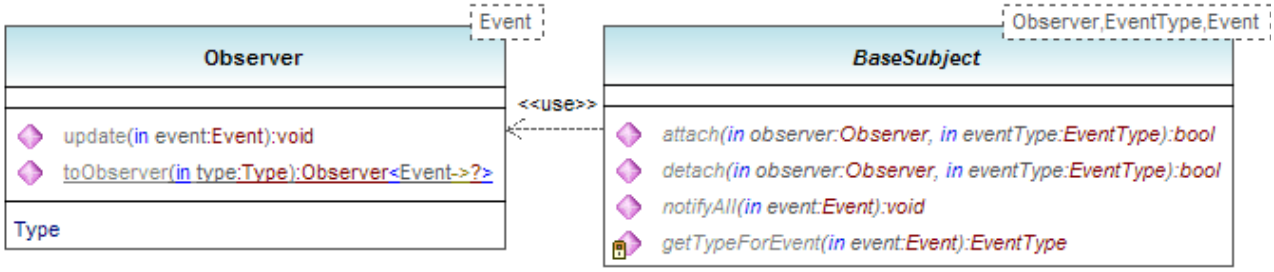

Abbildung 6: UML-Klassendiagramm der Basisklassen des Observer-Patterns

Die ersten Policies die benötigt werden sind Subscription-Policies. Sie bestimmen, auf welche Weise ein An- und Abmeldevorgang durchgeführt werden kann (Abbildung 7). So wäre es nach Bedarf möglich, dass sich Observer nur für ein bestimmtes Ereignis bei einem Subject anmelden können. Für den NarrationController werden zwei Implementationen dieser Policy vorgenommen: BaseSubscriptionSubject realisiert eine Anmeldung ohne Berücksichtigung von bestimmten Ereignissen, während es von EventSubscriptionSubject vorgesehen wird. Sowohl die Verwaltung angemeldeter Observer als auch deren Benachrichtigung erfolgt bei beiden Policies auf unterschiedliche Weise. BaseSubscriptionSubject kann einen einfachen Vector aus der Standard Template Library als Observer-Liste verwenden, in dem alle angemeldeten Observer referenziert sind (ObserverID) und über den bei Benachrichtigungen iteriert werden kann. EventSubscriptionSubject muss sich zusätzlich zur ObserverID merken, für welches Event die Observer angemeldet wurden, um bei Benachrichtigungen nur die für das eingetretene Event angemeldeten Observer zu informieren.

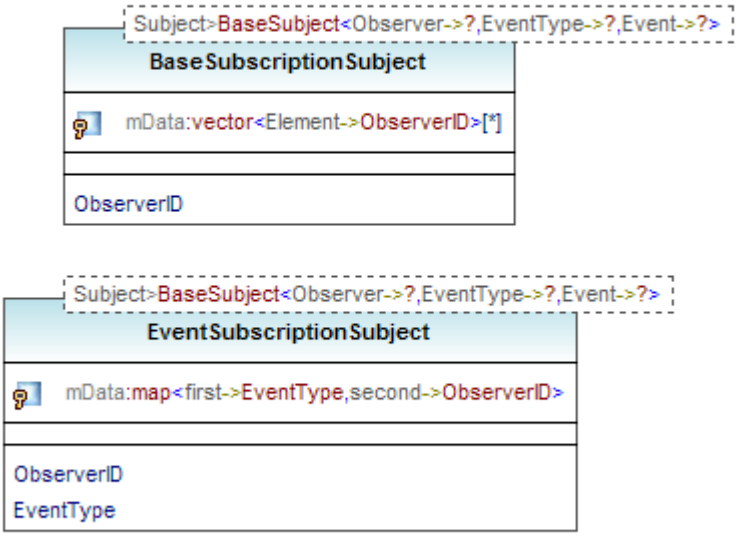

Abbildung 7: UML-Klassendiagramm der Subscription-Policies

Weitere Policies könnten Memorymanagement-Policies sein (Abbildung 8). In bestimmten Fällen kann es sinnvoll sein, wenn die Existenz der Observer an die des Subjects gebunden ist, so dass sich das Subject selbst um das Freigeben des Speichers kümmert, wenn die angemeldeten Observer nicht mehr benötigt werden.

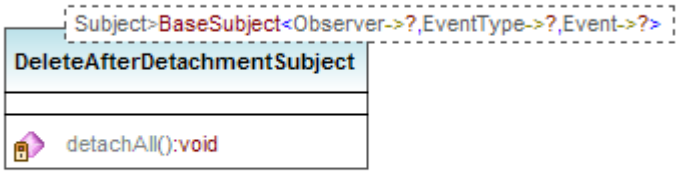

Abbildung 8: UML-Klassendiagramm einer Memorymanagement-Policy

Zu diesem Zweck wird die DeleteAfterDetachment-Policy implementiert. Dank des Decorator-Patterns in Kombination mit Mixin kann diese sowohl auf BaseSubscriptionSubject als auch auf EventSubscriptionSubject angewendet werden.

Zuletzt sei noch die Verwendung von Notification-Policies beschrieben, die sich dem erwähnten Problem annehmen, wenn während einer Benachrichtigung eine An- oder Abmeldung von Observern ausgelöst wird und somit eine Inkonsistenz der Observer-Liste verursacht wird. Die hier verwendete Lösung sieht die Implementation einer Warteschlange vor: Die Policy-Klasse CommandQueueingSubject (Abbildung 9) speichert alle Benachrichtigungs- und An-/Abmelde-Anforderungen als Commands in einen Queue.

Genau genommen verwendet CommandQueueingSubject zwei Queues, deren Inhalt regelmäßig ausgetauscht wird. In einen werden alle angeforderten Commands gespeichert. Er ist somit „offen" für das Hinzufügen weiterer Commands. Der andere ist ein „geschlossener" Queue, was bedeutet, dass er zum Zeit-

punkt der Benachrichtigung nicht veränderbar ist und keine weiteren Commands zulässt, wodurch seine Konsistenz gewährleistet wird. Während des Benachrichtigungsvorganges werden alle Commands indem geschlossenen Queue ausgeführt. Neue Commands, die während des Benachrichtigungsvorganges hinzugefügt werden, werden in den offenen Queue zwischengespeichert und beim nächsten Benachrichtigungsvorgang abgearbeitet. Die von der Super-Klasse vorgeschriebene Überschreibung der Methoden attach und detach wird so vorgenommen, dass deren Super-Methoden nicht aufgerufen werden. Stattdessen wird die jeweilige Anforderungen über einen Aufruf der Methode CommandQueueingSubject::queueCommand umgewandelt und in den offenen Queue gespeichert.

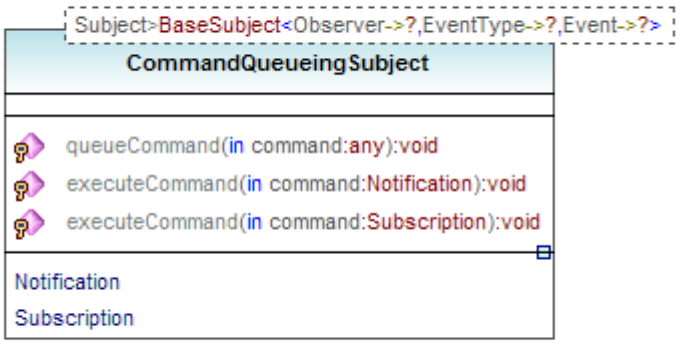

Abbildung 9: UML-Klassendiagramm einer Notification-Policy

Beim Aufrufen der notify()-Methode wird zunächst die Benachrichtigungsanforderung auch per queueCommand als Command gespeichert, dann der Offen/Geschlossen-Status der Queues vertauscht, um die Commands im zuvor offenen Queue nun im geschlossen Status abarbeiten zu können. Die eigentliche Ausführung der Commands erfolgt über die polymorphe Methode CommandQueueingSubject::executeCommand(). Die Variante, die ein Notify-Command als Parameter akzeptiert, ruft hierbei die notify()-Methode der Super-Klasse auf, während die mit Subscription-Command parametrisierte Methode die attach()- bzw. detach()-Methode der Super-Klasse aufruft.

Andrei Alexandrescu schlägt mit ClosedNotificationSubject eine weitere Notification-Policy vor. Hierbei wird der Aufruf von attach()/detach() während des Benachrichtigungsvorganges durch das Auslösen einer Exception verhindert. Ein solches Verhalten ist in der hier erstellten Architektur jedoch nicht wünschenswert und wird daher nicht implementiert.

Die Zusammenstellung eines konkreten Observers ist durch dieses Design sehr einfach und flexibel und soll hier am Beispiel des ExecutionListeners erfolgen. Der ExecutionListener ist als eine innere Klasse des NarrationControllers realisiert, der es diesem ermöglicht, über Szenenwechsel und Transitionen zu informieren. Jedes Ereignis wurde bisher von einer eigenen Methode repräsentiert, die den Namen der Szene oder Transition durch eine Parameterübergabe erhielt. Für mögliche neue Ereignisse im Story-Verlauf müssten zusätzliche Methoden hinzugefügt werden, was sehr mühsam ist. Eine bessere

Alternative ist die Verwendung eines Event-Systems. Die Klasse StoryEvent repräsentiert hierbei ein Ereignis im Story-Verlauf. StoryEvent bietet hierbei nicht nur Zugriff auf den Namen oder Typ des Ereignisses, sondern auch Zugriff auf das entsprechende IcmlElement, welches die Ursache für das Ereignis ist (siehe Abbildung 10). Um einen StoryObserver über das Eintreten von StoryEvents zu informieren, ist zunächst ein StorySubject nötig, bei dem sich der StoryObserver anmelden kann. Listing 3 zeigt, wie ein solches StorySubject über eine Typdefinition erstellt werden kann.

```
typedef EventSubscriptionSubject
                  <
                        BaseSubject<Observer, StoryEvent::EventType, StoryEvent>
                  > StorySubject;
```

Listing 3: Definition des StorySubjects durch die Komposition mehrere Policies

Zunächst wird festgelegt, welche Observer-Schnittstelle, welche Event-Klasse, und somit auch welcher Event-Typ von BaseSubject verwendet werden soll. Damit sich die Observer auch nur zu bestimmten StoryEvents anmelden können, wird das BaseSubject noch mit EventSubscriptionSubject dekoriert. Durch die Typdefinition kann nun statt der Template-Definition der neue Typ StorySubject Verwendung finden. Sollte später die Notwendigkeit bestehen, weitere Policies zu StorySubject hinzuzufügen, so muss diese Änderung praktischerweise nur bei der Typdefinition vorgenommen werden. In Listing 4 erfolgt eine solche Definition auch für StoryObserver.

```
typedef Observer<StoryEvent> StoryObserver;

class _NC_EXPORT ExecutionInterface
{
public:
    typedef Observer<StoryEvent> ExecutionListener;
    ...
};
```

Listing 4: Definition von StoryObserver und ExecutionListener

Für einfacheren Zugriff durch externe Komponenten wird die Typdefinition für StoryObserver auch in ExecutionInterface vorgenommen. ExecutionInterface verfügt über eine addExecutionListener()-Methode (siehe Kapitel 6.4.3), bei der ExecutionListener automatisch für alle notwendigen Events beim eigentlichen StorySubject angemeldet wird. Eine externe Komponente, die über Ereignisse im Story-Verlauf informiert werden möchte, muss somit nur noch von ExecutionInterface::ExecutionInterface ableiten, die entsprechende update()-Methode implementieren und sich über die addExecutionListener()-Methode des ExecutionInterface bei StorySubject anmelden.

6.3.2 Story-History

Die neue Logging-Funktionalität wird über die Klasse StoryHistory implementiert. Wie in Abbildung 10 zu sehen ist, erbt StoryHistory von der Typdefinition StoryObserver (in der Abbildung der Einfachheit halber als Klasse dargestellt), um sich als solcher bei einem StorySubject anzumelden. Bei jedem über die update()-Methode eintreffenden StoryEvent erstellt StoryHistory einen Log-Eintrag mit Zeitstempel. Hierbei kann sowohl die vergangene Zeit seit Start der Story berücksichtigt werden (ExecutionTime), als auch die Dauer der aktuellen Szene (EventTime). Zuvor muss jedoch ein gültiges History-Dokument im Speicher liegen. StoryHistory:: createScenarioHistory() ist hierbei für das Erstellen eines neuen History-Dokumentes zuständig. Über StoryHistory::loadScenarioHistory() lassen sich jedoch auch schon vorhandene History-Dokumente einlesen und über den Parameter „historyEntries" eine beliebige Anzahl vorheriger Story-Durchläufe mit in den Speicher laden, z.B. um statistische Auswertungen vornehmen zu können.

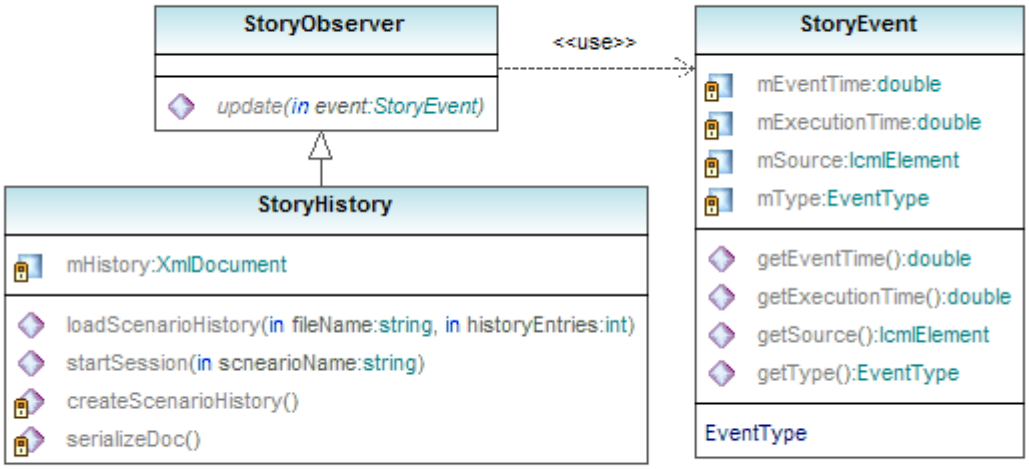

Abbildung 10: UML-Klassendiagramm von StoryHistory und dessen Abhängigkeiten

Vor dem Beginn eines neuen Story-Durchlaufs muss über StoryHistory::startSession() eine neue Session angelegt werden, in die die bevorstehenden StoryEvents als ICML-ähnliche XML-Elemente gespeichert werden. Da die meisten Events auf der Interpretation von ICML-Elementen beruhen (Szenenbeginn, Aktionsausführung, Transitions- oder Stimuli-Auslösung), lassen sich die schon vorhandenen ICML-Strukturen für das Event-Logging wieder verwenden. StoryEvent::getSource() ermöglicht den Zugriff auf das ICML- Element, welches das StoryEvent ausgelöst hat.

Die aufgezeichneten Daten sollen während und nach der Ausführung zur Auswertung verwendet werden können, so dass ein dynamischer Zugriff auf die XML-Daten vorhanden sein sollte. Hierfür bietet sich die Skriptsprache XPath an, die z.B. auch in XSLT-Transformationen zu diesem Zweck verwendet wird

[28]. Durch die Verwendung von XPath wird auch die Anforderung erfüllt, den Zugriff auf eine Skriptsprache über ICML zu ermöglichen (vgl. Kapitel 5.2). Allerdings wird XPath von der bisher verwendeten XML-API nicht unterstützt. Neben dieser weist die bisherige XML-API weitere funktionelle Schwachstellen auf, weswegen sie durch eine andere ersetzt wird, worauf in den Kapiteln 6.4.4 und 7.3 eingegangen wird. An dieser Stelle gehen wir im Weiteren davon aus, dass wir die Möglichkeit haben, in einem XML-Dokument per XPath auf die einzelnen XML-Elemente zuzugreifen.

Mit StoryHistory::serializeDoc() werden die Log-Einträge schließlich in eine Datei geschrieben. Diese trägt denselben Namen wie die ICML-Datei mit der Storydefinition, jedoch mit der Endung „_history.xml“. Der Aufbau eines solchen History-Dokumentes ist in Abbildung 11 aufgeführt. Es wird ein neues ICML-Wurzel-Element namens „history“ benutzt, indem sich beliebig viele Sessions befinden können, die einen Storydurchlauf repräsentieren. Jede Session wird, um ihre Eindeutigkeit zu gewährleisten, mit dem aktuellen Datum, Zeitstempel, Storynamen und Szenarionamen versehen. Mit Hilfe dieser Attribute können einzelne oder mehrere Sessions gefunden und deren Daten ausgewertet werden. Eine Session enthält alle während eines Story-Verlaufs eingetretenen story-relevanten Ereignisse, die sich für eine Auswertung eignen.

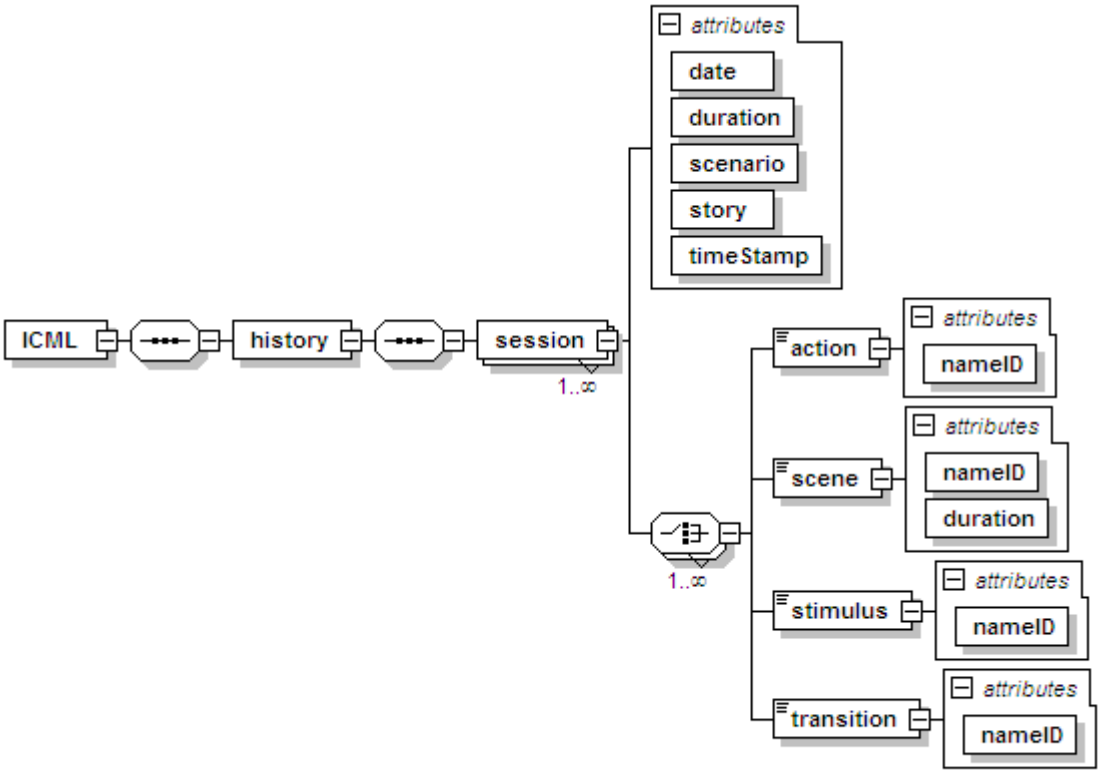

Abbildung 11: XML Schema Darstellung eines History-Dokumentes

6.3.3 Zeitkontrolle

Für die Überwachung der Story ist es wichtig zu wissen, wann welches Ereignis in der Story passiert ist. Es ist also eine Messung der Zeit notwendig. Die bis-

herigen Möglichkeiten, Zeiten zu messen, waren auf das Aufrufen einer globalen Methode, die die vergangene Zeit seit Programmstart zurückgibt, beschränkt. Dies ist eine sehr ungenaue und unzureichende Möglichkeit der Zeiterfassung. Noel Llopis erklärt in dem Kompendium „Game Programming Gems 4" das Problem, das durch unkontrolliertes Abfragen von Zeitwerten auftreten kann und schlägt ein Lösungskonzept mit Timern vor. Die Architektur ermöglicht nicht nur eine plattformunabhängige Implementierung, es ist auch möglich, die Timer dazu zu benutzen, die Ablaufgeschwindigkeit unabhängig von der eigentlichen Systemzeit zu verlangsamen oder zu beschleunigen [39].

Das Problem noch einmal zusammengefasst: Wenn Situationen für einen bestimmten Zeitwert berechnet werden müssen und sich diese Berechnung auf mehrere Klassen aufteilt, muss dieser Zeitwert in irgendeiner Form übergeben werden, damit an allen Stellen mit demselben Wert gerechnet wird. Ein eigenständiger Aufruf der Methode zur Zeitwertermittlung durch jede Klasse führt zu Inkonsistenzen, da die Zeit während der Ausführung des Programmes fortlaufend voranschreitet.

Timer übernehmen die Kapselung des Wissens um die Zeit, so dass Klassen, die Timer benutzen (z.B. durch Vererbung), zeitabhängige Berechnungen durchführen können, ohne sich um die Messung und Verwaltung der Zeit kümmern zu müssen. Die Timer selbst müssen hierfür über eine Veränderung der Zeitwerte informiert werden. Dies ist erneut ein klassischer Anwendungsfall für das Observer-Pattern. Dank unserer generischen Implementation in Kapitel 6.3.1 ist es ein Leichtes, diesen Teil der Funktionalität zu realisieren.

Für die Messung der Zeit soll die Klasse Clock zuständig sein. Sie ist also das Subject, bei dem sich die unterschiedlichen Timer als Observer anmelden können (siehe Abbildung 12). Als Event wird hierbei ein double-Wert mit der

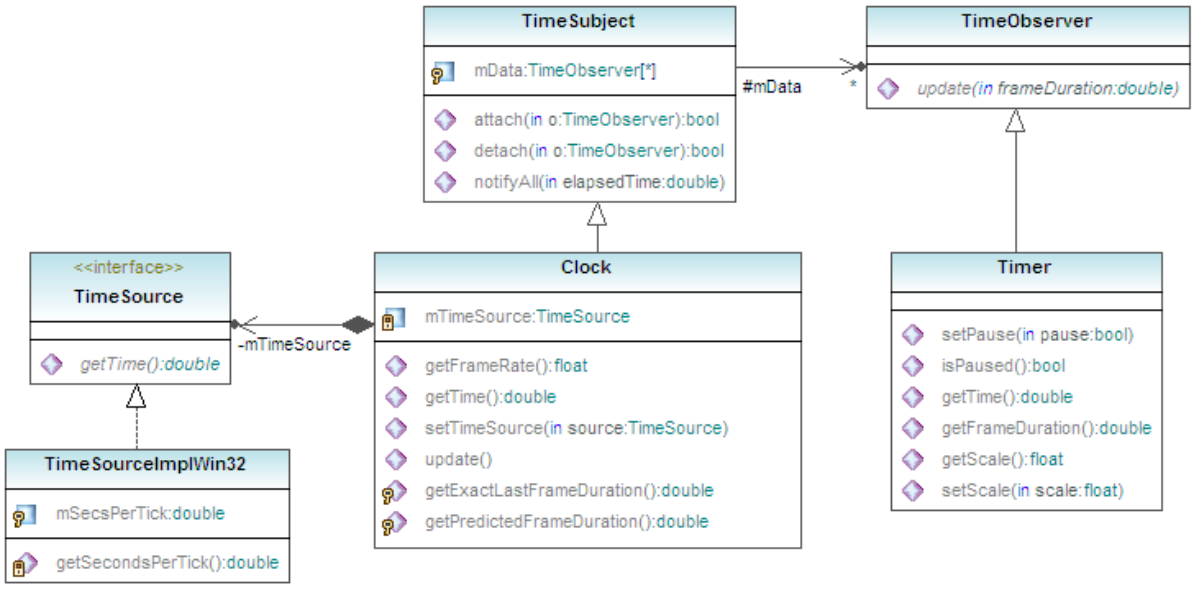

Abbildung 12: UML-Klassendiagramm des Zeitmessers

vergangenen Zeit seit der letzten Benachrichtigung und somit seit der letzten Aktualisierung verwendet. Die Aktualisierung von Clock erfolgt über dessen update()-Methode. Diese muss immer wieder aufgerufen werden, um die Zeitmessung während der Ausführung aufrecht zu erhalten. Die Zeitwerte zwischen den Update-Zyklen werden hierbei interpoliert, um nach einem längeren Zyklus keine zu starken Zeitsprünge zu erhalten, die durch aufwändige Berechnungen oder sehr langsame Rechner entstehen könnten. Die Timer verfügen weiterhin über die Möglichkeit der Skalierung der Zeit, wodurch ein Verlangsamungs- oder Beschleunigungs-Effekt erzielt werden kann. Sowohl Clock als auch Timer können jederzeit pausiert werden. Timer verfügen über eine lokale Zeit-Verwaltung, so dass sie die Zeit der Clock nicht beeinträchtigen. Dadurch kann eine Beschleunigung oder Pausierung auf einzelne Timer angewandt werden, während die restlichen Timer normal weiterlaufen. Wird Clock jedoch pausiert, bleiben automatisch auch alle bei Clock angemeldeten Timer stehen.

Die Methoden zur Zeitmessung sind auf verschiedenen Betriebssystemen unterschiedlich implementiert. Daher findet diese nicht direkt in Clock statt. Es wird eine weitere Abstraktionsschicht „TimeSource" eingeführt, wodurch Plattformunabhängigkeit und somit die Portierung auf ein anderes System ermöglicht wird. TimeSource verwendet hierbei das Adapter-Pattern und wird über eine Komponente, die das Factory-Pattern realisiert, instanziiert. Eine genauere Erläuterung dieser Vorgehensweise wird in Kapitel 6.4.3 vorgenommen.

6.4 Trennung von Datenmodell und Ablauflogik

6.4.1 Problemstellung

Um die neuen Möglichkeiten zur Ablaufoptimierung verwenden zu können, müssen Änderungen an der Ablauflogik vorgenommen werden. Zum einen muss eine Instanz der eben beschriebenen Clock-Klasse hinzugefügt und aktualisiert werden, um die bei ihr angemeldeten Timer auszulösen. Zum anderen müssen die alten Observer-Schnittstellen an die neue StoryObserver-Architektur angepasst und über eintretende StoryEvents benachrichtigt werden. Beide Änderungen betreffen die Hauptschleife des NarrationControllers. Diese ist in der bisherigen Architektur auf die zwei Hauptkomponenten des NarrationControllers aufgeteilt: Den NarrationController selbst und den IcmlGraphMaster (vgl. Abbildung 5). Der IcmlGraphMaster sorgt für die Abbildung des Datenmodells im Systemspeicher und ist zuständig für das Laden, Speichern und Verändern des Story-Modells und somit ICML. Die ICML-Elemente werden durch die Klasse IcmlElement repräsentiert, die als Composite-Pattern realisiert ist, um ein einfaches Traversieren durch den Story-Graphen zu ermöglichen. Jedes Element besitzt eine eigene Klasse, die von IcmlElement ableitet. Der Einfachheit halber wurde die gesamte Logik, die ein solches Element repräsentiert, innerhalb der Composite-Struktur implementiert. Das ist in sofern praktisch, da die meisten ICML-Elemente nicht nur Daten enthalten, sondern auch eine Funktionalität zu erfüllen haben (z.B. Aus-

werten von Berechnungen bei Condition-Elementen). Die zur Ausführung der Funktionalität benötigten Daten stehen somit in den jeweiligen Element-Klassen zur Verfügung.

Dies sorgt jedoch im Umkehrschluss dafür, dass ein Zugriff auf die Funktionalität über das Datenmodell erfolgen muss und somit über die Komponente IcmlGraphMaster. Deshalb wurde sich dafür entschieden, zusätzlich zur Verwaltung des Datenmodells auch den Abspielvorgang einer Story im IcmlGraphMaster und seinen Kind-Elementen zu implementieren. Die Aufgabe der NarrationController-Komponente beschränkt sich, neben der Bereitstellung einer Schnittstelle zur Abspielsteuerung und der Bearbeitung von ICML-Daten (Facade-Pattern), bisher darauf, externen Komponenten die Registrierung als Observer zu ermöglichen. Dadurch werden sie über Änderungen im Abspielvorgang informiert.

Die hieraus entstehende Problematik ist in Abbildung 13 gut ersichtlich. Durch die Aufteilung der Zuständigkeiten innerhalb der Hauptschleife herrscht ein ständiger Datenaustausch zwischen NarrationController und IcmlGraphMaster. Die Kommunikation zwischen Observern und Subject erfolgt in beiden Richtungen nicht direkt, sondern immer über den NarrationController. Teilweise, so wie bei NarrationController::mmdcTriggerStimulus(), werden sogar Validierungen der Events, in diesem Beispiel eines Stimulus, vorgenommen. Da der NarrationController jedoch über keine Informationen des aktuellen Story-Status verfügt, leitet er die Validierung an IcmlGraphMaster::isValidStimulus() weiter. Würden Observer und Subject direkt kommunizieren bzw. die Observer von IcmlGraphMaster verwaltet, könnte die Validierung intern in IcmlGraphMaster vorgenommen werden, wodurch die Schnittstelle entschlackt und unnötige Methodenaufrufe vermieden werden könnten.

Die bisherige Vorgehensweise erschwert Wartbarkeit und Erweiterbarkeit der Anwendung, da nicht ersichtlich ist, an welcher Komponente welche Änderungen vorgenommen werden müssen, um neue Funktionalitäten hinzuzufügen oder bestehende zu überarbeiten. Weiterhin sorgt die Schnittstelle des NarrationControllers für Unübersichtlichkeit und somit eine hohe Fehleranfälligkeit, da sie sowohl die Methoden zur Erstellung einer Story als auch zur Abspielsteuerung enthält.

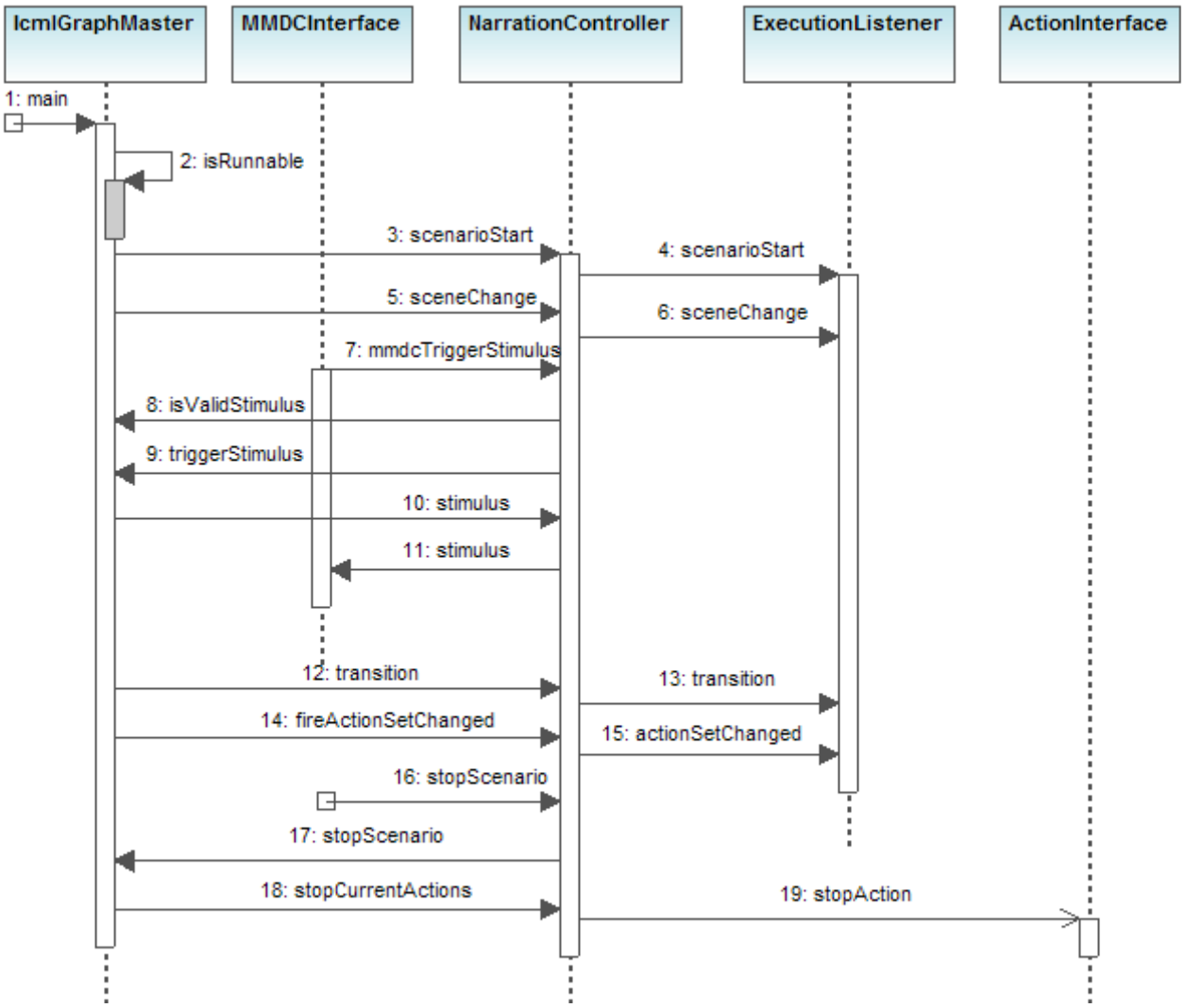

Abbildung 13: UML-Sequenzdiagramm der alten Ablauflogik

6.4.2 Lösung

Die Ablauflogik und die dafür notwendigen Strukturen werden aus IcmlGraphMaster und NarrationController extrahiert und in eine separate Klasse NarrationEngine gekapselt. Die bisherige Schnittstellendefinition der NarrationController-Klasse wird zu der abstrakten Klasse ExecutionInterface umgestaltet, die nur noch die Methoden zur Abspielsteuerung enthält. ExecutionInterface dient hierbei weiterhin als Facade auf die eigentlichen, intern verwendeten Schnittstellen. Im Gegensatz zur ursprünglichen NarrationController-Klasse reicht es jedoch nur die Anfragen zur An- und Abmeldung weiter und speichert oder verwaltet keine Informationen, die von einer anderen Komponente abgefragt werden müssen.

Zum Verwalten des Datenmodells muss eine neue Schnittstelle angeboten werden, über die ICML-Elemente bearbeitet werden können. Diese Schnittstelle bietet IcmlGraphMaster ab sofort selbst. Mit der nicht mehr vorhandenen

Abhängigkeit vom NarrationController kann der IcmlGraphMaster in Zukunft als vollständige, unabhängige Komponente betrachtet und verwendet werden. Die Verwaltung des Datenmodells wird deswegen in eine eigene Bibliothek (DLL) ausgelagert, auf deren Schnittstellen der NarrationController zum Abspielen der Story zugreifen kann.

Bei einem solch umfassenden Refactoring der Hauptkomponenten können auch gleich weitere Anforderungen, die an die neue NarrationController-Architektur gestellt werden, Berücksichtigung finden. So ist es gewünscht, die Implementierungsdetails und somit die Abhängigkeit von Hauptschleife und Ablauflogik zu trennen. Dadurch ist es möglich, unterschiedliche Hauptschleifen zu verwenden. Dies wird über Spezialisierungen des ExecutionInterfaces realisiert. So wird es z.B. eine Implementierung geben, die über eine eigene Hauptschleife verfügt, und eine weitere, die eine übergeordnete Hauptschleife einer anderen Applikation benötigt, aus der heraus eine Aktualisierung der Ablauflogik vorgenommen wird. Denkbar wäre eine weitere Implementierung, die zur Realisierung der eigenen Hauptschleife einen Thread verwendet. Die jeweils benötigte Implementierung kann dann über eine Factory, wie im folgenden Kapitel beschrieben, instanziiert werden.

6.4.3 Deklaration und Spezialisierung des ExecutionInterfaces

Die Deklaration des ExecutionInterfaces sorgt nicht nur für eine Trennung zwischen Datenmodell und Ablauflogik, durch die unterschiedlichen Spezialisierungsmöglichkeiten sorgt es auch für eine Trennung von Hauptschleife und Ablauflogik und somit allgemein für eine Trennung sämtlicher externer Komponenten von der eigentlichen NarrationController-Architektur. Änderungen an der internen Implementierung betreffen externe Komponenten dadurch nicht, da die verwendete Schnittstelle die eigentlichen Implementierungsdetails verbirgt.

ExecutionInterface bietet die Möglichkeit zur Abspielsteuerung. Diese Möglichkeiten existieren im bisherigen NarrationController schon und können von dessen Schnittstellen-Deklaration übernommen werden. Um eine Spezialisierung der Hauptschleife vornehmen zu können, wird zusätzlich eine abstrakte update()-Methode hinzugefügt. Eine von ExecutionInterface erbende Klasse könnte diese abstrakte Methode implementieren und somit eine mögliche Hauptschleife darstellen. Allerdings gibt es einige grundlegende Funktionalitäten, wie z.B. das Laden einer ICML-Datei oder der Zugriff auf die eigentliche NarrationEngine, die unabhängig von der Art der Hauptschleife benötigt werden. All diese Funktionalitäten werden in der Klasse ExecutionImplBase implementiert. Diese lässt die zur Spezialisierung der Hauptschleife notwendigen Methoden abstrakt, so dass eine von ExecutionImplBase abgeleitete Klasse nur noch diese Methoden implementieren muss, um die gewünschte Spezialisierung zu ermöglichen.

Abbildung 14 verdeutlicht die Anwendung dieser Vererbungshierarchie, indem als Beispiel eine Standalone- und eine Plugin-Implementation dargestellt werden. Die Standalone-Implementation verwendet eine blockierende Endlosschleife in ihrer update()-Methode, in der sie in jedem Schleifendurchlauf die in ExecutionImplBase referenzierte NarrationEngine aktualisiert. Die Plugin-Implementation reicht den Aufruf der eigenen update()-Methode direkt an die NarrationEngine weiter, so dass die eigentliche Hauptschleife in einer übergeordneten Applikation erfolgen muss.

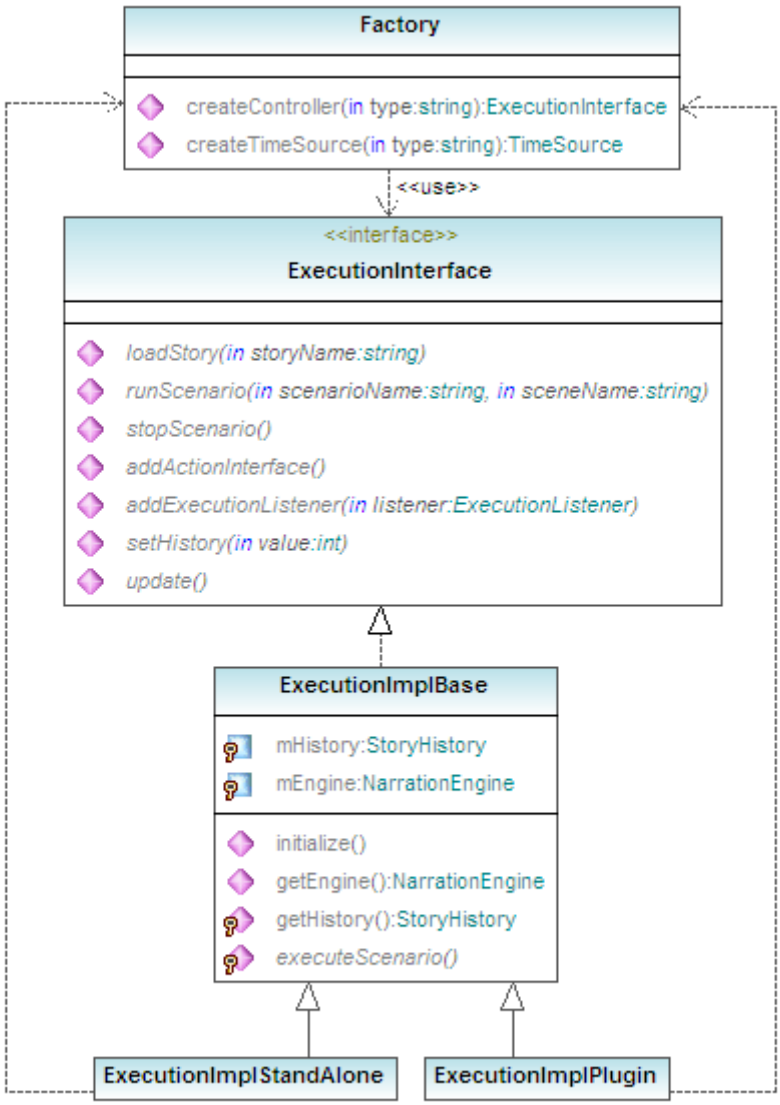

Abbildung 14: UML-Klassendiagramm aller von ExecutionInterface abhängigen Klassen

Die Auswahl und somit Instanzierung einer Implementierung erfolgt über eine Factory (Factory-Method-Pattern). Hierfür müssen die verfügbaren Implementationen der Factory bekannt gemacht werden. Üblicherweise wird hierbei jede Implementation mit einer eindeutigen Typisierung ausgestattet. Mit Hilfe der Typisierung lässt sich dann zur Laufzeit ein Objekt eines beliebigen der Factory bekannten Typen erzeugen und über eine gemeinsame Schnittstelle

nutzen. Sowohl die Art der Typisierung als auch die Wahl der Schnittstelle und die Methode zur Objekterzeugung hängen vom jeweiligen Verwendungszweck ab. In diesem Fall ist zumindest die Schnittstelle durch ExecutionInterface eindeutig vorgegeben, die Wahl der Typisierung und Objekterzeugung kann im Verlauf der Entwicklung jedoch durchaus wechseln. Es wäre also hilfreich, neue Entscheidungen diesbezüglich auf einfache Weise treffen zu können.

Diese Möglichkeit bietet uns die schon in Kapitel 6.3.1 verwendete, auf Policies basierende Programmierung. Andrei Alexandrescu stellt einen möglichen Lösungsansatz für eine solche generische Factory in seinem Buch „Modern C++ Design" vor und implementiert diesen in seiner Software-Bibliothek „Loki" [35].

Sowohl Schnittstelle als auch Typisierungsart und Objekterzeugung werden als Policies einer Factory angesehen, die dieser über Template-Parameter hinzugefügt werden. Dadurch lässt sich dieselbe Factory-Implementierung für unterschiedliche Anwendungsfälle verwenden und auch im Verlauf der Entwicklung nachträglich auf einfache Weise modifizieren. So ist eine Verwendung derselben Implementierung sowohl für die in Kapitel 6.3.3 entworfene TimeSource-Schnittstelle als auch den GraphMaster möglich. GraphMaster benötigt zum Instanziieren der unterschiedlichen IcmlElemente eine Factory. Die bisher eigenständige Implementierung wird im Zuge des Refactorings von IcmlGraphMaster in Kapitel 6.4.4 zur Vereinheitlichung und besseren Wartbarkeit durch die hier vorgestellte, generische Lösung ersetzt.

Sowohl bei ExecutionInterface als auch bei TimeSource und IcmlElement wird die Objekterzeugung in Zukunft über die Hilfsklasse Creator vorgenommen (Listing 5). Diese bietet eine create()-Methode an, die eine neue Instanz der jeweilig benötigten Klasse zurück liefert. Um die Hilfsklasse in allen drei Anwendungsfällen benutzen zu können, verlangt sie einen Template-Parameter, der die jeweilige zu erzeugende Klasse angibt.

```
template<class T>
class Creator
{
public:
    ~Creator(){};
    T* create()
    {
        return new T;
    }
};
```

Listing 5: Creator-Klassen-Template

Die Verwendung von Creator bei der damit verbundenen Anmeldung eines konkreten Typs bei einer Factory ist in Listing 6 am Beispiel von ExecutionPlugin dargestellt. ExecutionPlugin implementiert ExecutionInterface bzw. ExecutionImplBase als Plugin und somit ohne eigene Hauptschleife. Weiterhin ist zu sehen, dass für die Typisierung der konkreten Implementierungen von Execu-

tionInterface der Containertyp string aus der STL gewählt wurde. Wie gemeinhin üblich ist auch hier die Factory neben dem Factory-Method-Pattern nozusätzlich mit Hilfe des Singleton-Patterns realisiert. Auch hier wird diese Möglichkeit eines globalen Zugriffes auf die Factory verwendet, wofür die statische Member-Methode Factory::Instance() bereitgestellt wird. Diese liefert die Instanz der Factory zurück, so dass nun auch nicht-statische Member-Methoden aufgerufen werden können. Die Methode zur Registrierung eines konkreten Typs der jeweiligen Factory-Implementierung verlangt als Übergabeparameter sowohl den Typnamen als auch die Klasse und deren Methode, die für die Objekterzeugung zuständig sind. An dieser Stelle kommt auch die auf ExecutionPlugin spezialisierte Creator-Hilfsklasse ins Spiel. Um nicht für jede Registrierung eine globale Variable zu erzeugen, erfolgt sie innerhalb eines anonymen Namespaces [35].

```
namespace
{
    Creator<ExecutionPlugin> creator;
    const bool registered = Factory::Instance()
                        .registerExecutor( "plugin",
                                           &creator,
                                           &Creator<ExecutionPlugin>::create
                                          );
}
```

Listing 6: Anmeldung bei einer Factory

6.4.4 Refactoring des Datenmodells

Der IcmlGraphMaster ist ab sofort nur noch für die Repräsentation und Verwaltung des Datenmodells zuständig. Die Implementierung der Logik der einzelnen IcmlElemente wird aus ihnen herausgenommen und in die NarrationEngine integriert. Da der Zugriff auf die Elemente nun nicht mehr über den NarrationController erfolgt, muss IcmlGraphMaster eigene Schnittstellen anbieten, um Zugriff auf das Datenmodell zu gewährleisten. Hierfür wird die Klasse GraphMaster zur Verfügung gestellt. Über sie werden ICML-Dateien geladen und gespeichert sowie Zugriff auf die einzelnen Wurzel-Elemente des ICML-Graphen gewährt. Die zweite Schnittstelle ist die Klasse IcmlElement, die verschiedene Methoden deklariert, um direkten Zugriff auf die Daten der einzelnen Kind-Elemente zu gewähren (siehe Abbildung 15).

Eine der Anforderungen aus Kapitel 5.2 verlangt, dass beim Speichern des Graphen ICML-fremde Elemente nicht gelöscht oder überschrieben werden. Bisher war es so, dass beim Laden einer Datei nur bekannte ICML-Elemente eingelesen wurden und alle anderen (z.B. XML-Kommentare oder Elemente aus einem anderen XML-Namespace) verworfen wurden. Beim Speichern des Dokumentes sind diese Elemente dann verloren gegangen. Dies wird nun geändert, indem das vollständige XML-DOM (Document Object Model), unabhängig von seinem Inhalt, geladen und im Speicher gehalten wird. Die einzelnen

Knotenelemente werden hierbei in die jeweiligen IcmlElement-Spezialisierungen gekapselt und konsistent gehalten. ICML-fremde Elemente werden durch die neue Klasse UnknownElement repräsentiert. Im Gegensatz zur vorherigen Implementierung wird die Konsistenz nicht durch ein Kopieren der XML-Daten in eigene Strukturen erhalten. Stattdessen verwendet IcmlElement direkt die entsprechende XmlNode.

Bisher wurde die XML-Funktionalität von Qt verwendet, um XML-Dokumente zu laden und zu speichern. Die XML-API von Qt bietet jedoch, wie schon erwähnt, nicht genügend Funktionalität für die neuen Anforderungen an ICML, wie z.B. die notwendige Validierung des XML Schemas von ICML oder die Anwendung von XPath-Expressions. Es wird eine andere XML Bibliothek benötigt, die die neuen Anforderungen unterstützt. Während dieser Umstellung ist es ratsam eine zusätzliche Abstraktionsschicht zu erstellen, die die gängigen, nach W3C-Standard benötigten, XML-Zugriffsmethoden anbietet. Die eigentliche Implementierung wird somit vor dem GraphMaster verborgen, eine Umstellung auf eine andere XML-API in Zukunft wesentlich erleichtert. Die Auswahl der verwendeten Implementierung erfolgt wieder über eine Factory, der GraphMaster selbst verwendet als Schnittstelle nur noch die abstrakten Klassen XmlParser, XmlDocument und XmlNode (siehe Abbildung 16). Die hier verwendete XmlFactory realisiert jedoch nicht wie die bisher vorgestellten Factories das FactoryMethod-Pattern, sondern das AbstractFactory-Pattern. Dies ist notwendig, da in diesem Fall pro konkreter Implementierung nicht eine, sondern drei Klassen vorhanden sind. XmlFactory ist also eine abstrakte Klasse, die ebenso wie XmlParser, XmlDocument und XmlNode für die jeweilige Implementierung spezialisiert werden muss. Neben den zu implementierenden Methoden deklariert sie jedoch noch die statischen Methoden setFactory() und getFactory() für die Anmeldung und den Zugriff auf eine konkrete Implementierung. Somit ist die Verwendung einer konkreten XmlFactory möglich, die wiederum das Wissen um die Erzeugung der konkreten Klassen XmlParser, XmlDocument und XmlNode kapselt, ohne deren eigentliche Implementierung zu kennen.

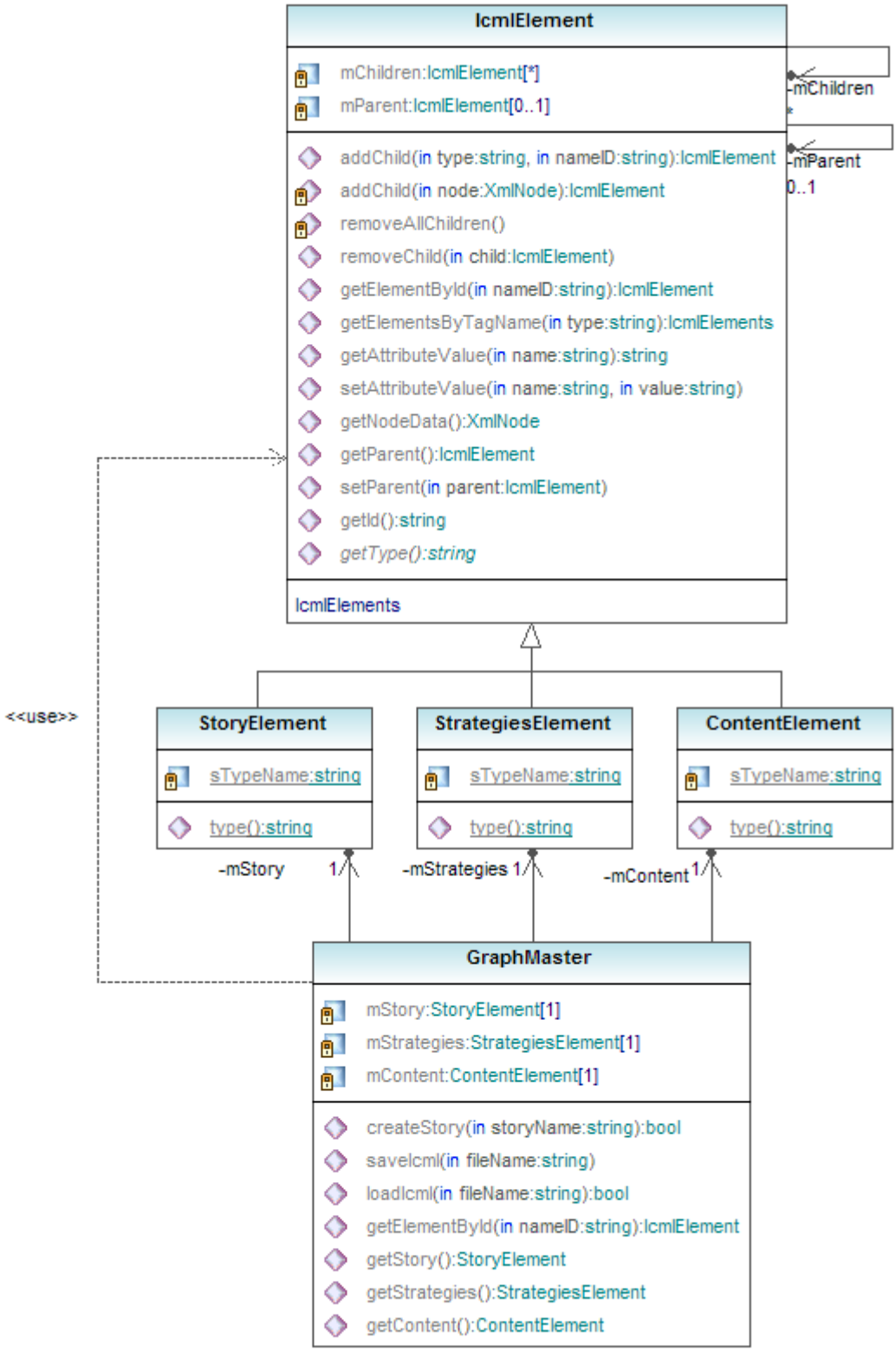

Abbildung 15: UML-Klassendiagramm der Schnittstellen von IcmlGraphMaster

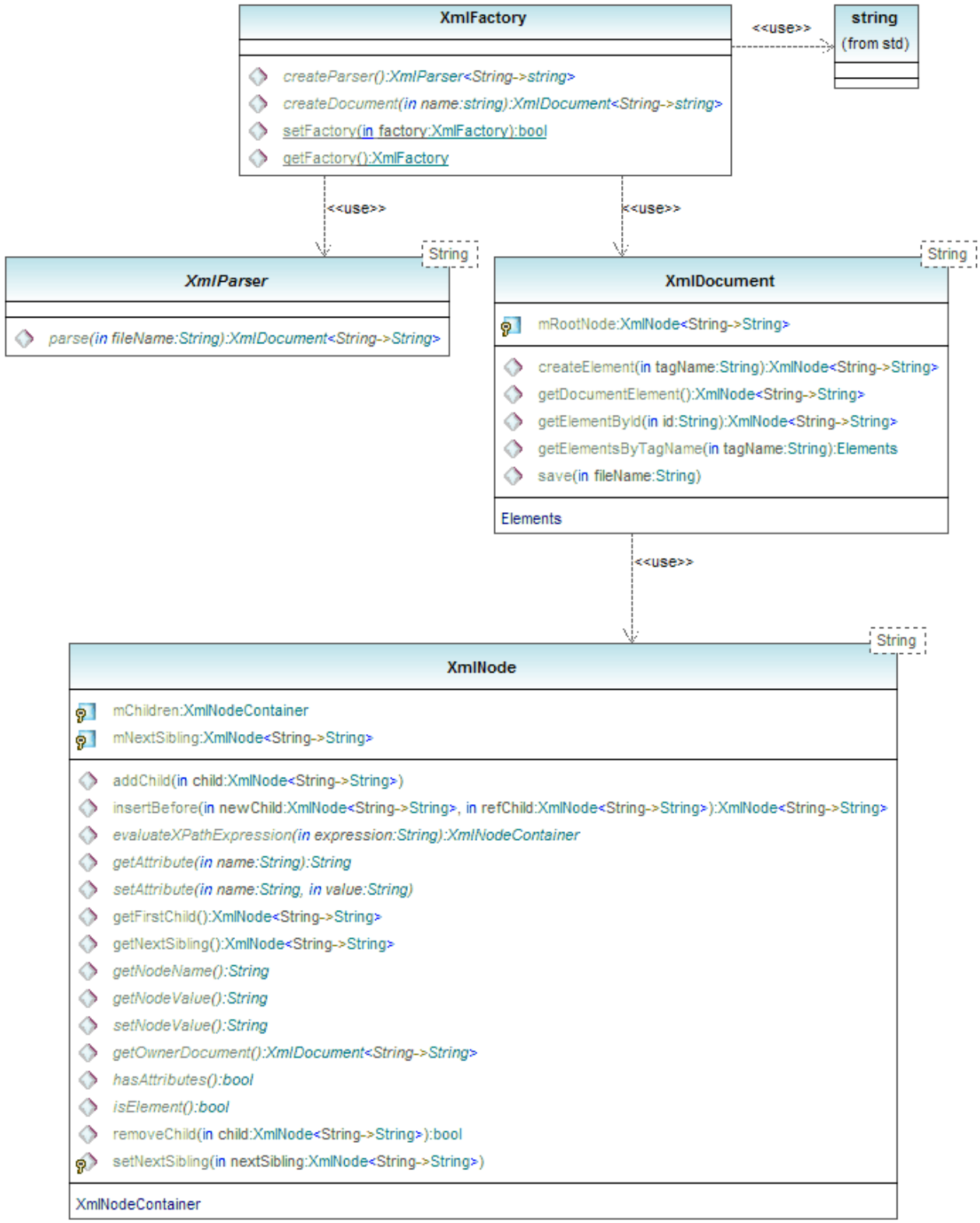

Abbildung 16: UML-Klassendiagramm der XML-Abstraktionsschicht

6.4.5 Refactoring der Ablauflogik

Durch die Trennung von Datenmodell und Ablauflogik ist es in Zukunft nicht mehr möglich, auf dieselbe Art und Weise Zugriff auf die Daten zu nehmen, wie es noch in der IcmlGraphMaster-Komponente der Fall war. Wie schon in 6.4.2 gefordert, wird die Ablauflogik in die neu zu implementierenden Klasse NarrationEngine gekapselt, deren Architektur in Abbildung 17 dargestellt ist. Diese verfügt über eine update()-Methode, die von einer beliebigen Hauptschleife (Standalone/Plugin/Thread o.ä.) aufgerufen werden kann (siehe 6.4.3). Die update()-Methode übernimmt wiederum die schon erwähnte, notwendige Aktualisierung von Clock (siehe 6.3.3). Clock löst dabei die bei ihr angemeldeten Timer aus. Genau genommen handelt es sich um Tasks, Spezialisierungen von Timer, die bestimmte story-relevante Aufgaben erfüllen. Die beiden wichtigsten Tasks sind hierbei die Überprüfung von Bedingungen (CheckConditionTask) und die Durchführung der ActionSet-Elemente (ExecuteActionSetTask).

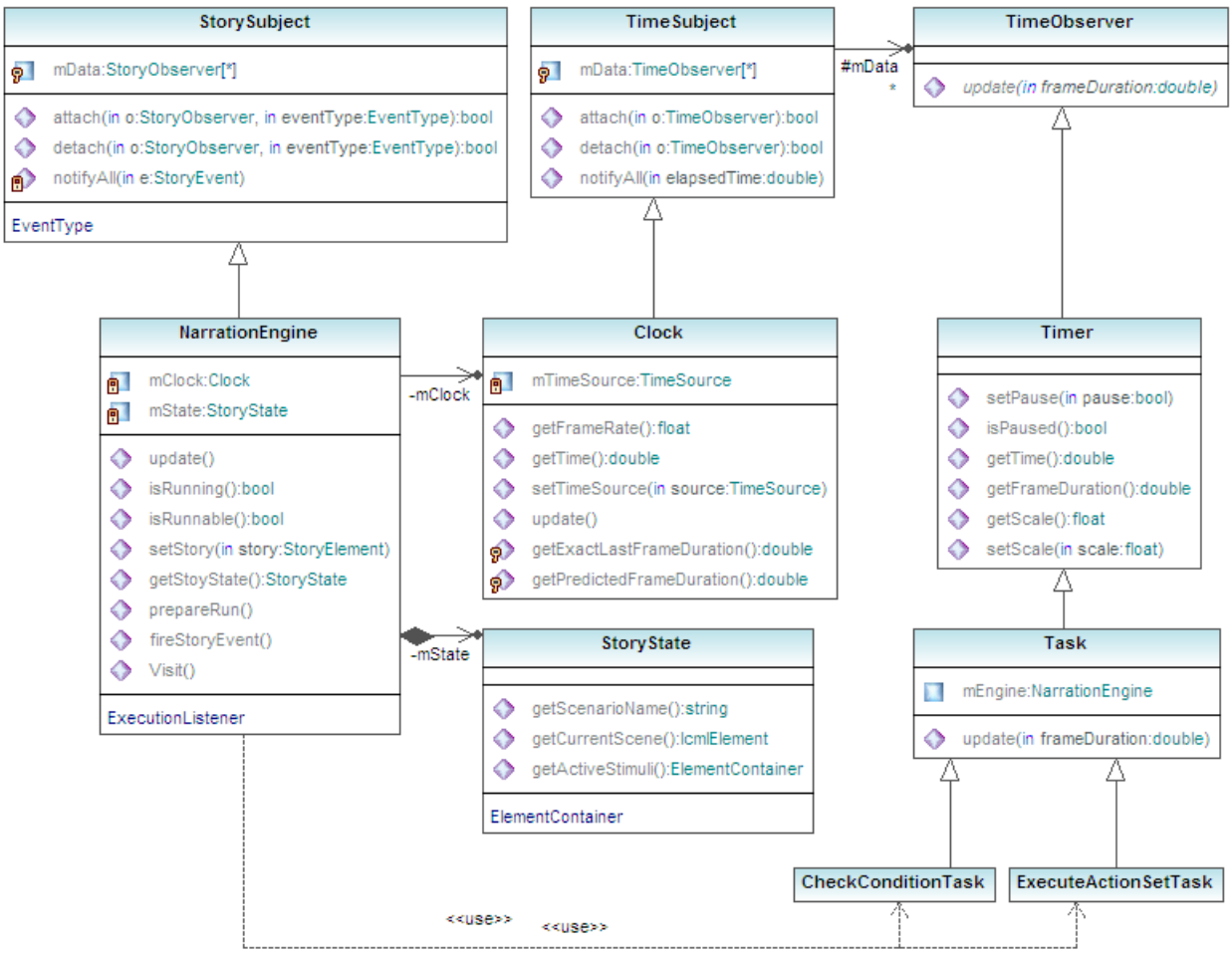

Abbildung 17: UML-Klassendiagramm der an der Ablauflogik beteiligten Klassen

Die Erstellung der Tasks hängt von der Story und somit vom Datenmodell ab. Neben dem Aufruf von Clock::update() ist NarrationEngine::update() auch für die Kontrolle der restlichen Ablauflogik zuständig. Zunächst wird überprüft, ob es überhaupt möglich ist, die Story abzuspielen. So ist vor dem ersten Aufruf von NarrationEngine::update() die Methode NarrationEngine::prepareRun() aufzurufen, wobei Story, Szenario und optional die Startszene übergeben werden. Letzteres ist hierbei nicht zwingend, da die Startszene normalerweise auch im Datenmodell hinterlegt ist. Dieser Aufruf wird bei korrekter Verwendung von ExecutionImplBase automatisch durchgeführt, wie Abbildung 18 zeigt. Die Story wird als StoryElement, einer Ableitung von IcmlElement (siehe Abbildung 15) übergeben. Somit erhält NarrationEngine Zugriff auf das ICML-Datenmodell.

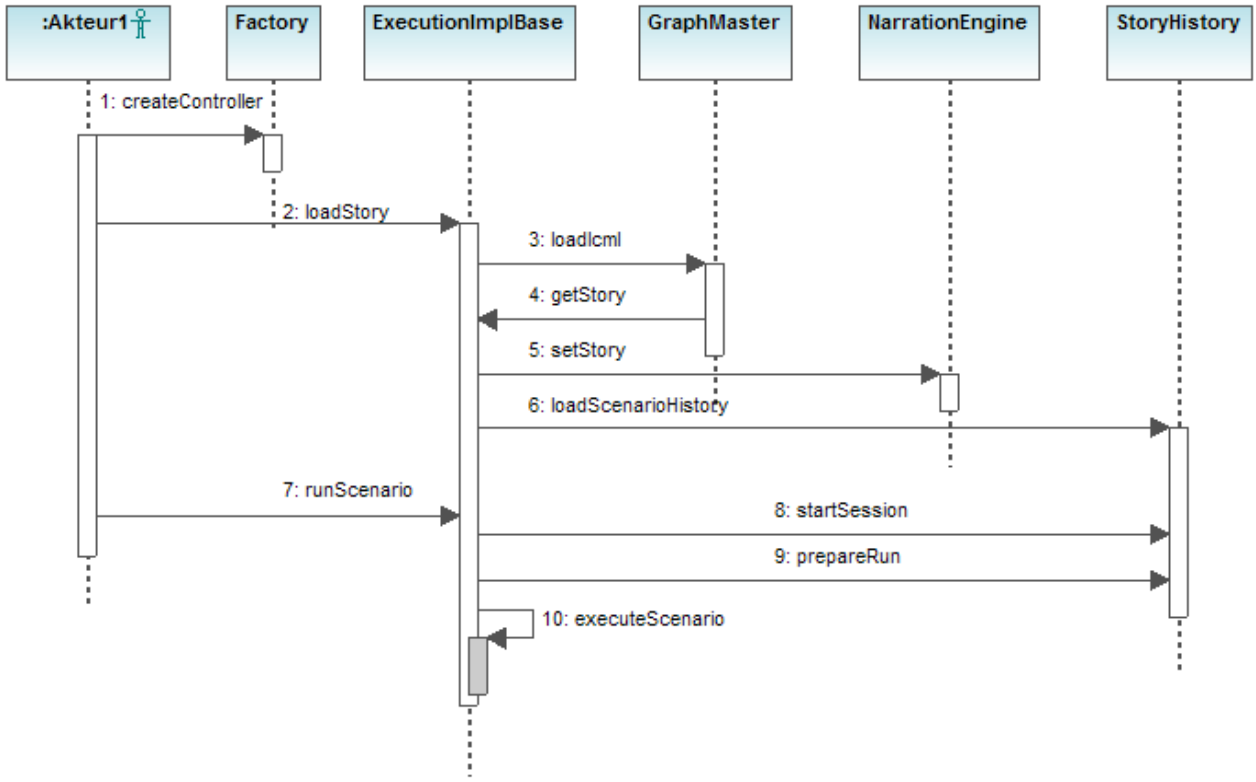

Abbildung 18: UML-Sequenzdiagramm der Initialisierung des NarrationControllers

Im weiteren Verlauf von NarrationEngine::update() wird die aktuelle Szene geprüft und, sofern noch nicht geschehen, mit deren Ausführung begonnen. Hier erfolgt über das StoryElement der Zugriff auf das Datenmodell. Durch die Verwendung des Composite-Patterns ist es möglich, über alle Kind-Elemente des StoryElementes zu iterieren. Bei der Suche nach einem Element mit einer bestimmten ID (wie z.B. das Suchen nach einer Szene mit einem bestimmten Namen) steht sogar die Methode IcmlElement::getElementById() zur Verfügung, die ein IcmlElement mit der entsprechenden ID zurück liefert, wenn es existiert (siehe Abbildung 15). Das zurückgegebene IcmlElement bietet Methoden, die die Abfrage von Typ, Attributen und weiteren Kind-Elementen des jeweiligen Elementes erlaubt. Um herauszufinden welches Verhalten die Ablauf-

logik als nächstes zeigen soll, muss der Typ des Elementes abgefragt werden. Dies würde bei der Vielfalt an ICML-Elementen allerdings zu extrem langen if/then/else- oder switch-Blöcken führen, was unübersichtlich und schlecht wartbar ist.

Eine elegantere Vorgehensweise ermöglicht das Visitor-Pattern. Hierbei wird eine automatische Typisierung vorgenommen und der jeweilige Typ als Argument einer polymorphen Methode übergeben, die dann das Verhalten des jeweiligen Elementes implementiert (siehe Abbildung 19). Der Methodenaufruf selbst sorgt somit dafür, dass der Typ bekannt ist. Um an die Typinformation zu gelangen, wird beim Visitor-Pattern wie folgt vorgegangen: Das IcmlElement, das besucht werden soll, erhält eine accept()-Methode, die den entsprechenden Visitor übergeben bekommt. Diese accept()-Methode wiederum ruft die visit()-Methode des Visitors auf und übergibt dabei den this-pointer. Damit dieser dem konkreten Elementtyp entspricht und nicht der Oberklasse IcmlElement, muss IcmlElement::accpect() als virtuelle Methode deklariert und von jedem konkreten Elementtyp reimplementiert werden. Durch den virtuellen Aufruf verweist der this-Pointer immer auf den konkreten Elementtyp. Dies führt jedoch auch dazu, dass die Visitor-Klasse schon in der Komponente IcmlGraphMaster bekannt sein muss. Der Visitor wird somit als abstrakte Klasse StoryVisitor der IcmlGraphMaster-Komponente hinzugefügt.

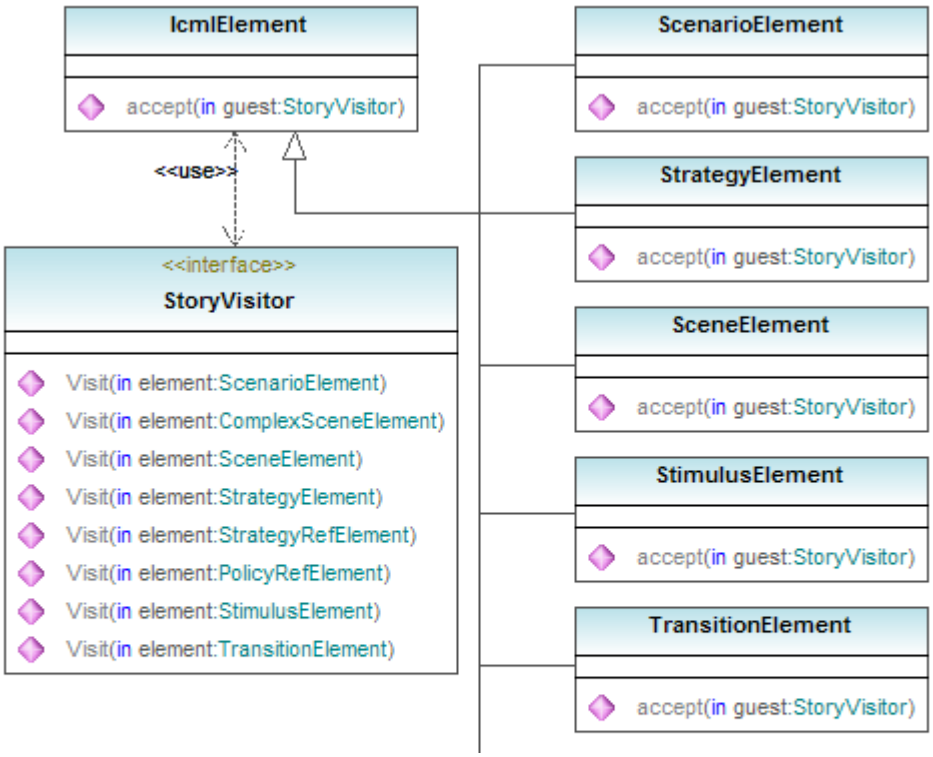

Abbildung 19: UML-Klassendiagramm des StoryVisitors

Das Visitor-Pattern hat den Nachteil, dass es eine zyklische Abhängigkeit zwischen IcmlElement, dessen Ableitungen und der Visitor-Klasse selbst erzeugt. Eine Änderung der einen Komponente wirkt sich auch auf die jeweilige andere aus. Wenn neue, besuchbare Elemente hinzugefügt werden, muss die Visitor-Schnittstelle angepasst werden, was sehr mühsam sein kann. Dadurch, dass die für die NarrationEngine benötigten und somit besuchbaren Elemente jedoch bekannt sind und feststehen, ist dieses Problem zu vernachlässigen. Die NarrationEngine ist nur an den Elementen interessiert, die einen Story-Status beschreiben oder indirekten Einfluss auf diesen nehmen. Die Funktionalität von StoryVisitor wird in NarrationEngine durch Vererbung zur Verfügung gestellt. Dabei implementiert NarrationEngine alle von StoryVisitor vorgegebenen abstrakten visit()-Methoden.

Sollten doch einmal weitere Elemente besucht werden müssen, erleichtert die in der Loki-Biblothek vorgenommene generische Implementierung des Visitor-Patterns die Änderung der Schnittstelle, die quasi vom Compiler selbst durchgeführt wird. Die notwendigen abstrakten Methodendeklarationen werden anhand der vom Entwickler angegebenen besuchbaren Elemente automatisch erzeugt. Diese sehr nützliche Eigenschaft war ausschlaggebend für die Verwendung der generischen Variante des Visitor-Patterns im NarrationController. Auch für die aufwändige Reimplementierung der accept()-Methode in den einzelnen besuchbaren Elementen schlägt die Loki-Bibliothek mit der Verwendung eines Makros eine bequeme Lösung vor (siehe Listing 7) [35].

```
#define ICML_DEFINE_CYCLIC_VISITABLE(SomeVisitor) \
    virtual SomeVisitor::ReturnType accept(SomeVisitor& guest) \
    { return guest.GenericVisit(*this); }
```

Listing 7: Visitor-Makro-Definition

Die Verwendung von StoryVisitor innerhalb der Ablauflogik wird anhand von Abbildung 20 deutlich. Wie schon erwähnt, wird beim Aufruf von NarrationEngine::update() als Erstes eine Überprüfung vorgenommen, um die Lauffähigkeit zu testen (2). Wenn dies der erste Aufruf der update()-Methode ist, oder im vorherigen Update-Zyklus eine Transition in eine andere Szene ausgelöst wurde, so wird nun eine Szene geladen (3). Der Zugriff erfolgt hierbei über die visit()-Methode des StoryVisitors. Um genau zu sein, wird in diesem Fall die mit SceneElement überladene visit()-Methode aufgerufen (5-6). Die Implementierung dieser Methode sieht vor, die für die neue Szene gültigen Transitionen zu laden (7), deren Auslösungs-Bedingungen sogleich als Tasks hinzugefügt werden (8). Wenn das Laden der neuen Szene abgeschlossen ist, wird Clock aktualisiert und somit alle ausstehenden Tasks bearbeitet (9).

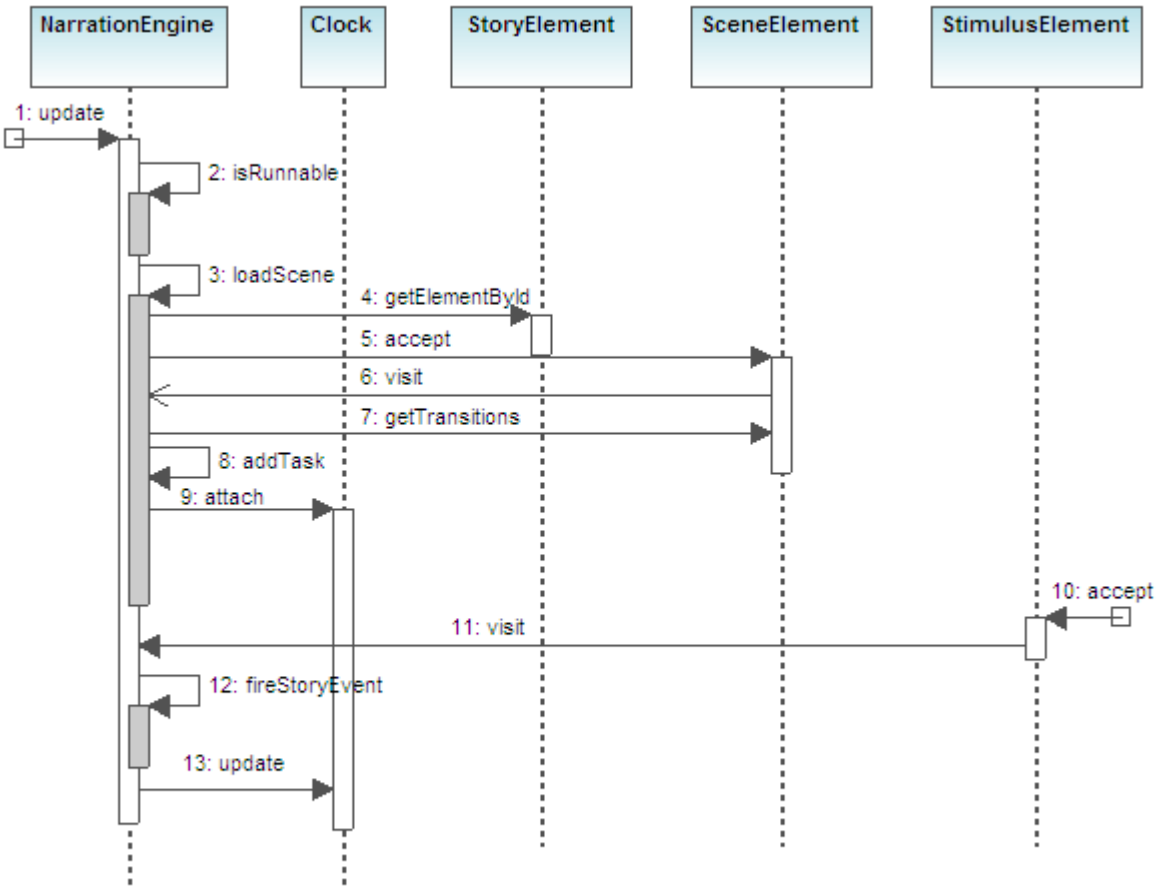

Abbildung 20: UML-Sequenzdiagramm der Ablauflogik

Bis zum nächsten Update-Zyklus wird nun auf eintretende Ereignisse von außerhalb gewartet, wie z.B. das Auslösen eines Stimulus durch Anwendereingaben. Durch welche Komponente dieses Ereignis ausgelöst wird, ist dank des Visitor-Patterns für die Ablauflogik von NarrationEngine uninteressant. Für sie tritt ein Ereignis in Kraft, sobald die jeweilige Visit-Methode des ereignisauslösenden IcmlElements aufgerufen wird. Ein Aufruf von StimulusElement::accept() reicht also aus, um einen Stimulus auszulösen (10-12). Hierbei ist natürlich zu beachten, dass accept() als Übergabeparameter die NarrationEngine als StoryVisitor erwartet. Einer ereignisauslösenden Komponente muss die NarrationEngine bekannt sein, während die NarrationEngine selbst vollkommen unabhängig von einer solchen ist. Damit externe Komponenten keinen direkten Zugriff auf die NarrationEngine-Instanz benötigen, wird u.a. die Klasse StimulusInterface zur Verfügung gestellt. Über die Methode StimulusInterface::triggerStimulus() ist es somit möglich, Stimuli auszulösen. Weiterhin wird StimulusInterface von StoryObserver abgeleitet, so dass auch eine Benachrichtigung über eingetretene Stimuli erfolgt.

Da sich die ablaufende Story durch eintretende Ereignisse ständig verändern kann, ist das Festhalten des aktuellen Status der Story nötig. Die NarrationEngine ist nur für die Ablauflogik zuständig, daher wird der Story-Status in die Klasse StoryState gekapselt. Die NarrationEngine hat Zugriff auf StoryState, um diesen wenn nötig auf den aktuellen Stand zu bringen.

6.5 Entscheidungsfindung und Eingriff (Plugin-Architektur)

6.5.1 Architektur

Durch die Überwachung der Story stehen Messwerte zur Verfügung, aus denen sich in einer Entscheidungsfindung Kennwerte berechnen lassen. Diese Kennwerte können über verschiedene Algorithmen ausgewertet werden, um über die Notwendigkeit eines Eingriffes zu entscheiden. Wie in Kapitel 4 erläutert wurde, kann es viele, teilweise individuell auf die Story zugeschnittene, Konzepte zur Entscheidungsfindung und zum Eingriff geben. Die Austauschbarkeit dieser beiden Komponenten muss durch die Architektur abgedeckt werden. Nach Möglichkeit soll, trotz der Verbindung, durch die ein Eingriff erst ausgelöst wird, wenn es sich für die Entscheidungsfindung als notwendig herausstellt, keine Abhängigkeit zwischen den einzelnen Komponenten bestehen.

Sowohl die Definition der Entscheidungsfindung als auch die Art des Eingriffes sollen in ICML erfolgen, wodurch entsprechende Elemente benötigt werden, die eine ausreichende Beschreibung der durchzuführenden Operationen ermöglichen. Bisher gibt es in ICML zur Abfrage von Bedingungen Condition-Elemente bzw. Variable-Elemente. Für einen Eingriff in den Story-Verlauf dienen Action-Elemente (siehe Kapitel 3.3). Hierbei handelt es sich jedoch um auf ganz spezielle Anwendungsfälle zugeschnittene Bedingungen und Aktionen, die Variablentypen sind auf numeric, string, und bool beschränkt. Für eine komplexere Entscheidungsfindung, auf die möglichst beliebige Eingriffe folgen können, sind die vorliegenden Elemente nicht geeignet. Abstrahiert betrachtet werden aber nur diese drei Elemente benötigt, um aus den Kombinationen der in Kapitel 4 vorgenommenen Konzeptkategorien eine Ablaufoptimierung zu erreichen. Daher wird im Folgenden versucht, die vorhandenen Elemente für eine generische Verwendung zu abstrahieren, um eine einfach zu bedienende und dennoch individuell erweiterbare Definitionsmöglichkeit von Strategien zur Ablaufoptimierung zu realisieren.

6.5.2 Erweiterung der Variablentypen

Für die unterschiedlichen Berechnungen zur Entscheidungsfindung der individuellen Konzept-Realisierungen werden geeignete Datentypen benötigt. Theoretisch ist es möglich, dass jedes Konzept zur Ablaufoptimierung eigene Variablentypen verwendet. Sollte z.B. jemand die Skriptsprache LUA zur Realisierung von Conditions und somit für die Entscheidungsfindung nutzen, so wäre ein Zugriff auf LUA-Variablen sicherlich nützlich. Ein weiteres Beispiel wäre eine Zugriffsmöglichkeit auf Attribute der ICML-Elemente über XPath.

Um dieses zu erlauben, aber trotzdem eine Kompatibilität zwischen den einzelnen Konzepten zu gewährleisten, ist eine allgemeine Schnittstelle zu definieren, über die ein Zugriff auf die unterschiedlichen Variablentypen möglich ist. Die Umsetzung erfolgt über die abstrakte Klasse VariableAccessor. Für den Zugriff auf Variablen eines bestimmten Typs werden hier die Methoden getVa-

riable und setVariable deklariert. Die Realisierung eines Variablentyps erfolgt durch Ableitung von VariableAccessor und Implementierung der beiden genannten Methoden. Die Signatur der Methoden darf sich dadurch trotz eines anderen Variablentyps nicht von der Oberklasse unterscheiden. Daher muss VariableAccessor einen Standard-Typen vorschreiben, der zum Setzen und Abfragen beliebiger Variablentypen verwendet werden kann. Hier wurde sich für string als Datentyp entschieden, alle zusätzlichen Variablentypen müssen somit eine Umwandlung des eigenen Datentypes in string unterstützen.

Unter Verwendung der Stream-Operatoren (operator<< und operator>>) lassen sich solche Umwandlungen sehr einfach realisieren. Diese Operatoren können für die jeweilig verwendete Stream-Klasse beliebig überschrieben werden, und der Stream als string ausgegeben werden. Eine zusätzliche Erleichterung einer derartigen Umwandlung erfolgt durch die Hilfsklasse VariableHelper. Diese deklariert eine statische Template-Methode namens convert(). Als Template-Parameter wird hierbei der umzuwandelnde Datentyp angegeben, und ein Aufruf des jeweiligen Stream-Operators automatisch durchgeführt (siehe Listing 8). Für den seltenen Fall, dass eine Umwandlung nicht über Stream-Operatoren erfolgen kann oder nicht gewünscht ist, lässt sich eine solche Template-Methode auf einen bestimmten Typ spezialisieren, wie in Listing 8 für den primitiven Datentyp bool vorgenommen.

```
template <class T>
static void convert(const std::string& value, T& result)
{
    std::istringstream stream(value);
    stream >> result;
}

template <>
static void convert(const std::string& value, bool& result)
{
    std::string lStr=value;
    std::transform(lStr.begin(),lStr.end(),lStr.begin(),::tolower);
    result=((lStr=="true") || (lStr=="1"));
}
```

Listing 8: Template-Methoden der Klasse VariableHelper zur Umwandlung von Datentypen

Sobald ein Datentyp auf eine der beschriebenen Weisen in einen string umgewandelt werden kann, lässt er sich durch Spezialisierung von VariableAccessor dem NarrationController als Variablentyp hinzufügen. Hierauf wird in Kapitel 6.5.5 genauer eingegangen.

6.5.3 Abstraktion von Conditions

Es gibt bisher keine allgemein verwendbaren Condition-Elemente in ICML. Bisher war die Möglichkeit von Abfragen auf Transitions und Actions beschränkt, was in den Elementen transitionConditions und actionFlowConditions resultierte. Diese geben zunächst nur den Kontext an indem sie auftreten können. Ihre eigentliche Logik wird, wie in Abbildung 21 dargestellt, in weitere Elementtypen aufgeteilt.

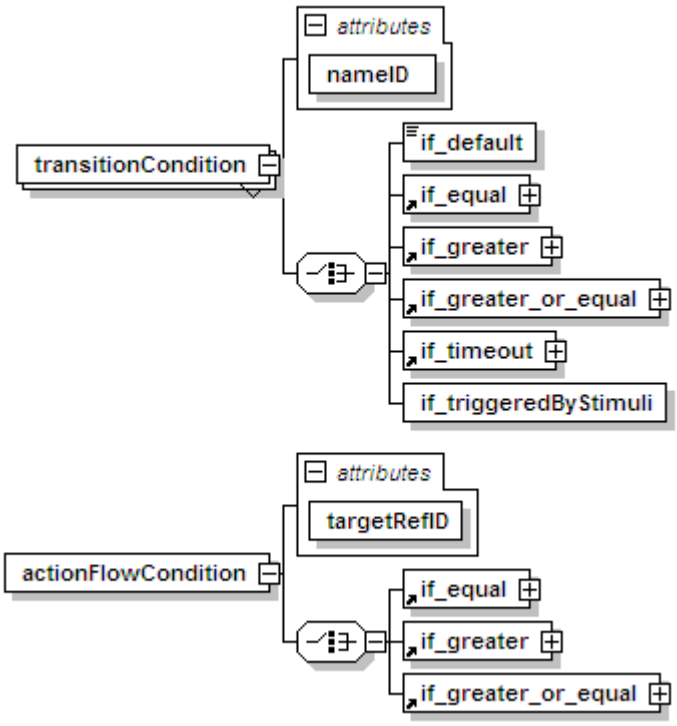

Abbildung 21: XML Schema Darstellung verschiedener Condition-Elemente

Die bisherigen Überlegungen haben ergeben, dass es noch andere Bedingungen geben könnte, die z.B. über Skriptsprachen oder komplexe Algorithmen ausgedrückt werden sollen. Wenn jedoch für jeden möglichen Condition-Typ ein eigenes Element angelegt wird, führt dies schnell zu Unübersichtlichkeit und schlechter Erweiterbarkeit. Weiterhin ist der Kontext, in dem die Bedingung auftauchen kann, für die Condition-Struktur selbst nicht relevant. Statt auf Elementebene könnte der Typ somit auf Attributebene angeben werden.

Durch die Bindung an den Kontext (z.B. an eine Transition oder eine Action) war die Definition von Conditions bisher nicht sehr intuitiv: Zuerst wurde der Kontext definiert, dann die Condition, bei deren Erfüllung der Kontext ausgeführt wird, z.B.: Löse eine Transition aus, wenn eine bestimmte Zeitdauer abgelaufen ist. Von den meisten Programmier- und Skriptsprachen ist man es jedoch gewohnt, in einem if/then-Konstrukt zu denken: Wenn eine Bedingung wahr ist (Ablauf einer Zeitdauer), so führe bestimmte Anweisungen aus (Auslö-

sen einer Transition). Um die Lesbarkeit und Bedienbarkeit von ICML zu erhöhen, ist es sinnvoll, dieses Prinzip in Zukunft anzuwenden.

Ein weiteres Problem sind die unterschiedlichen Parameter die bei den bisherigen Condition-Elementen auf Attributebene angegeben wurden. Anzahl, Name und Wertebereich der übergebenen Parameter können sich bei den unterschiedlichen Bedingungsarten jedoch stark unterscheiden. Eine Angabe der Parameter auf Elementebene, als Kind-Elemente des jeweiligen Condition-Elementes, löst dieses Problem. Dadurch erhalten Parameter auch eine eigene Attributebene, über die sie sich zusätzlich typisieren lassen (Namenskonventionen, Wertebereiche, etc.pp.).

Abbildung 22 zeigt die vereinheitlichten Condition-Elemente. Eine Typisierung findet nur noch auf Attributebene statt, Parameter werden als Kind-Elemente übergeben, die Aktion, die bei Erfüllung der Condition ausgeführt werden soll, wird ebenso als Kind-Element hinzugefügt. Um logische UND-/ODER-Verknüpfungen zu realisieren, wird weiterhin die Möglichkeit gegeben, weitere Condition-Elemente als Kind-Element hinzuzufügen. Listing 9 zeigt ein Beispiel für die Realisierung solcher Verknüpfungen.

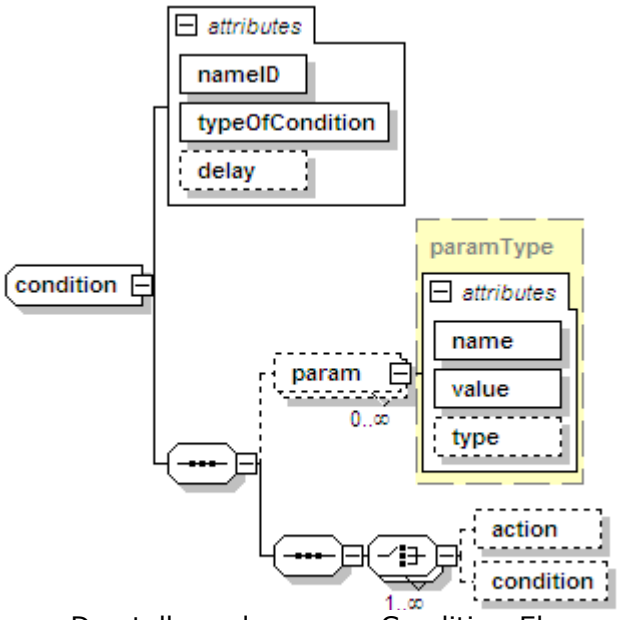

Abbildung 22: XML Schema Darstellung des neuen Condition-Elements

Die Abstraktion der Conditions ist nicht nur auf ICML-Ebene sinnvoll: Die Repräsentation der Condition im Code sah es, wie bisher bei allen ICML-Elementen, vor, dass die Logik in den einzelnen Elementen implementiert wird. Für jede Condition wurde somit eine eigene Klasse benötigt, obwohl die Logik der Conditions grundsätzlich die gleiche war und sich nur die Daten unterschieden. Hier bietet es sich eher an, die Logik von den Daten zu trennen und pro Condition-Typ nur eine Instanz der logischen Implementierung im Speicher zu halten. Diese wird dann mit den entsprechenden Element-Daten versorgt und wertet sie aus. Eine solche Vorgehensweise entspricht dem Flyweight-Pattern,

an das sich bei der Implementierung der Schnittstelle für Condition-Typen gehalten wird.

```
<condition type="default"> <!-- default type is always true -->
    <condition type="equal">
        <param name="left" value="4"/>
        <param name="right" value="4"/>
        <condition type="greater">
            <param name="left" value="4"/>
            <param name="left" value="3"/>
            <!-- this action only gets triggered if
                 4 equals 4 and 4 is greater than 3 -->
            <action type="triggerTransition" target="transition1"/>
        </condition>
    </condition>
</condition>
```

Listing 9: Eine mit AND verknüpfte Bedingung zum Auslösen einer Transition

6.5.4 Abstraktion von Actions

Bisher wurde für jede mögliche Aktion, die in einer Story passieren kann, ein eigenes Action-Element angelegt (siehe Abbildung 23). Das Problem war nicht nur, dass 3D- und 2D-Player unterschiedliche Aktionen benötigen, sondern dass dadurch Abhängigkeiten zwischen den Playern und dem NarrationController entstehen. Für den NarrationController sollten die Actions, die ein Player durchführen kann, irrelevant sein. Dem Player wird nur mitgeteilt, wann er welche Action mit welchen Parametern durchführen soll. Der NarrationController übt also eine reine Weiterleitungs-Funktion aus.

Der Player benötigt den Typ allerdings, um spezifische Operationen für die jeweilige Action durchzuführen. Ganz abschaffen lässt sich die Typisierung also nicht. Statt für jeden Typ in ICML ein eigenes Element zu definieren und somit auch eine zusätzliche konkrete Klassenimplementation zu erzwingen, sollte es nur ein Action-Element geben. Die Typisierung kann, wie nach der Abstraktion der Condition-Elemente auch, auf Attributebene stattfinden. Damit ist der Typ für den NarrationController nur ein String, der z.B. an den Player geschickt werden kann, wodurch beliebige Typen möglich werden. Die Attribute der spezifischen Actions werden in der neuen Action-Struktur als Parameter übergeben. Auch hier wird dasselbe Prinzip wie bei den Condition-Elementen verfolgt (siehe Abbildung 24).

Konkrete Actions benötigen eine Implementation, die dem NarrationController bekannt gemacht werden muss. Wie auch bei Variable- und Condition-Elementen wird hierfür eine Schnittstelle definiert, die im Folgenden Kapitel näher beschrieben wird.

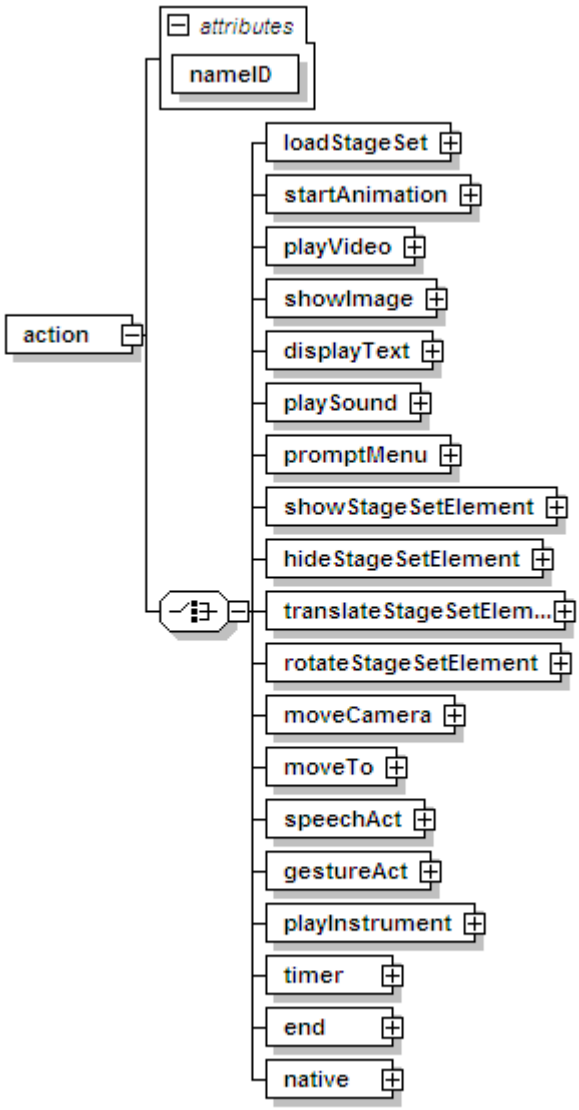

Abbildung 23: XML Schema Darstellung der bisherigen Action-Elemente

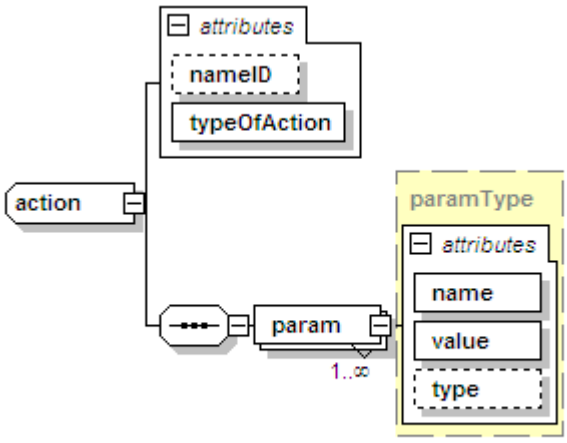

Abbildung 24: XML Schema Darstellung des neuen Action-Elements

6.5.5 PluginHolder

Dem NarrationController sollen sowohl Variable- als auch Condition- und Action-Typen hinzugefügt werden können. Wie schon erwähnt wird hierfür jeweils eine Schnittstelle definiert: VariableAccessor, ConditionEvaluator und ActionType. Diese Schnittstellen deklarieren unterschiedliche abstrakte Methoden, über die die jeweilige Funktionalität realisiert werden soll und die somit von konkreten Typen implementiert werden müssen. Zusätzlich zu diesen funktionalen Methoden deklarieren alle drei Schnittstellen eine Methode getType(), die eine eindeutige Bezeichnung des jeweiligen Typs zurückliefern muss. Das Hinzufügen konkreter Implementationen ist durch die Ähnlichkeit der drei Schnittstellen identisch, so dass eine einheitliche Verwaltung angeboten werden kann. Dieses übernimmt die Klasse PluginHolder, die Referenzen auf die ihr hinzugefügten Typen der unterschiedlichen Schnittstellen enthält (siehe Abbildung 25).

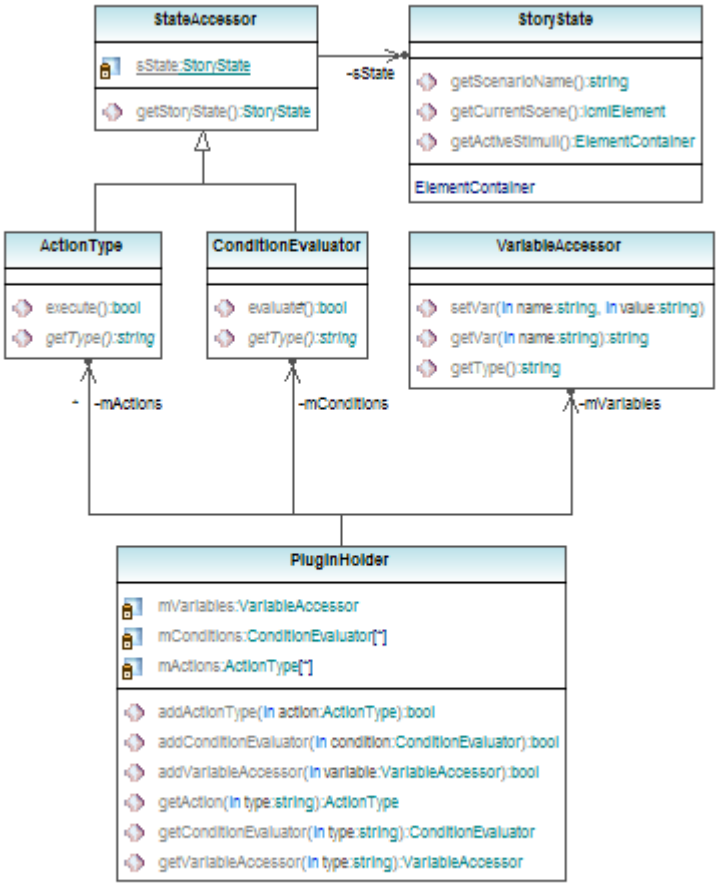

Abbildung 25: UML-Klassendiagramm der PluginHolder-Architektur

Das Hinzufügen von Typen wird ähnlich wie bei einer Factory umgesetzt: Durch die Verwendung des Singleton-Patterns ist ein globaler Zugriff auf PluginHolder möglich. Statt einer Erzeuger-Klasse wird, wie bei einer Prototyp-Factory, direkt ein Objekt vom jeweiligen Typ übergeben und somit dem PluginHolder hinzugefügt (siehe Listing 10). Da die einzelnen Schnittstellen das Flyweight-

Pattern verwenden (siehe Kapitel 6.5.3), ist von jedem Typen auch nur eine Instanz nötig. Der Zugriff auf die Instanz erfolgt über ihre jeweiligen getter-Methoden. Es ist sowohl bei ActionType- als auch bei ConditionEvaluator-Spezialisierungen wahrscheinlich, dass sie je nach Story-Status unterschiedliche Verhaltensweisen aufzeigen oder ihn gar verändern sollen. Um dies zu ermöglichen, wird ihnen durch Ableitung von der Schnittstelle StateAccessor der Zugriff auf den aktuellen Story-Status gewährt. Dieser wird von NarrationEngine mit jedem Update-Zyklus aktualisiert (siehe Kapitel 6.4.5).

```
namespace
{
    bool registered = PluginHolder::Instance()
                            .addVariableAccessor(new StandardStoryVariables());
}
```

Listing 10: Verwendung von PluginHolder am Beispiel des Hinzufügen eines Variablentyps

6.5.6 Strategien

Durch die bisherigen Änderungen ist es möglich, an bestimmten Stellen der Story eine Ablaufoptimierung vorzunehmen. Die Anforderungen verlangen jedoch auch eine globale Überwachung, die vom Story-Autor je nach Szenario an- und abschaltbar sein sollte. Dieser globale Mechanismus wird im Folgenden durch die Definition von Strategien umgesetzt. Hierfür werden zwei neue ICML-Elemente eingeführt, zum einen das Strategy-Element zum anderen das Wurzelelement „strategies“, welches Strategy-Elemente enthalten kann. Wie auch bei den bisherigen Möglichkeiten zur Ablaufoptimierung besteht eine Strategie-Definition aus Bedingungen, bei deren Erfüllung bestimmte Aktionen ausgeführt werden. Es werden hier also wieder die schon bekannten Condition- und Action-Elemente verwendet, wobei die Condition-Elemente während des gesamten Story-Ablaufes überwacht werden, vorausgesetzt die Strategie wurde auch einem Szenario in der Story zugeordnet.

Diese Zuordnung erfolgt über Referenzen. Wie auch bei ActionSets, die Referenzen auf Actions beinhalten (vgl. Abbildung 4), werden Referenz-Elemente für Strategien eingeführt. Um eine bessere Übersicht zu erlangen, werden diese Strategie-Referenzen jedoch nicht direkt einem ScenarioElement hinzugefügt, sondern zuvor wie Actions in einem Set zusammengefasst: dem Policy-Element. Die Bezeichnung „Policy“ wurde hierbei unter anderem als Anlehnung an die auf Policies basierende Programmierung gewählt. So wie Policy-Klassen Verfahrensweisen einer Klasse darstellen, so beschreiben Policy-Elemente die Verfahrensweisen innerhalb einer Story. Ein Policy-Element kann verschiedene Strategien beinhalten, die dasselbe Ziel verfolgen. Aus diesen Überlegungen heraus ergibt sich die in Abbildung 26 dargestellte ICML-Struktur.

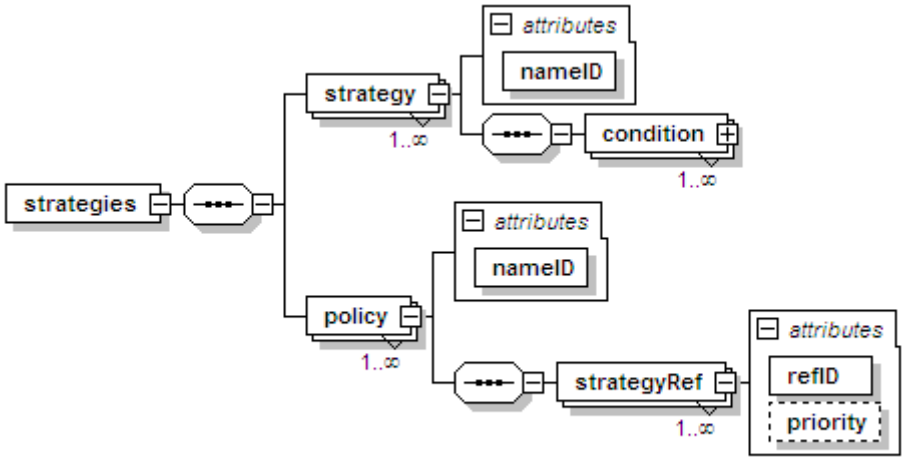

Abbildung 26: XML Schema Darstellung Strategien repräsentierender ICML-Elemente

6.6 Zusammenfassung

Um die Anforderungen in Kapitel 5 zu erfüllen, war es nicht nur nötig, ein Architektur-Design zu entwerfen, das eine Ablaufoptimierung unterstützt. Es hat sich herausgestellt, dass die bestehende NarrationController-Komponente einem umfassenden Refactoring unterzogen werden musste, damit sie für die gewünschten Einsatzzwecke über eine ausreichende Flexibilität verfügt. Für eine bessere Wiederverwendbarkeit wurde die Verwaltung von ICML vollständig aus der NarrationController-Komponente extrahiert und als eine eigenständige Komponente umgesetzt, die auch ohne den NarrationController, z.B. in einem Story-Editor Anwendung finden kann (siehe Abbildung 27, vgl. Abbildung 5).

Unter Anwendung von generischer und auf Policies basierender Programmierung konnte ein erfolgreiches Refactoring durchgeführt werden, wodurch die Architektur einfacher auf zukünftige Änderungen anzupassen ist. Die in Kapitel 4 durch das entwickelte Verfahren gestellten Anforderungen wurden durch die Klasse PluginHolder und seine Schnittstellen ActionType, ConditionEvaluator und VariableAccessor erfüllt. Diese erlauben es, den NarrationController um zusätzliche Funktionalitäten zu erweitern und ermöglichen auf diese Weise die Bereitstellung von Steuerungsmechanismen für eine Ablaufoptimierung in Form von zusammensetzbaren und miteinander kombinierbaren Strategien. Zum besseren Verständnis und Überblick befindet sich in Anhang 3 ein UML-Klassendiagramm mit einer Komplettübersicht der NarrationController-Komponente.

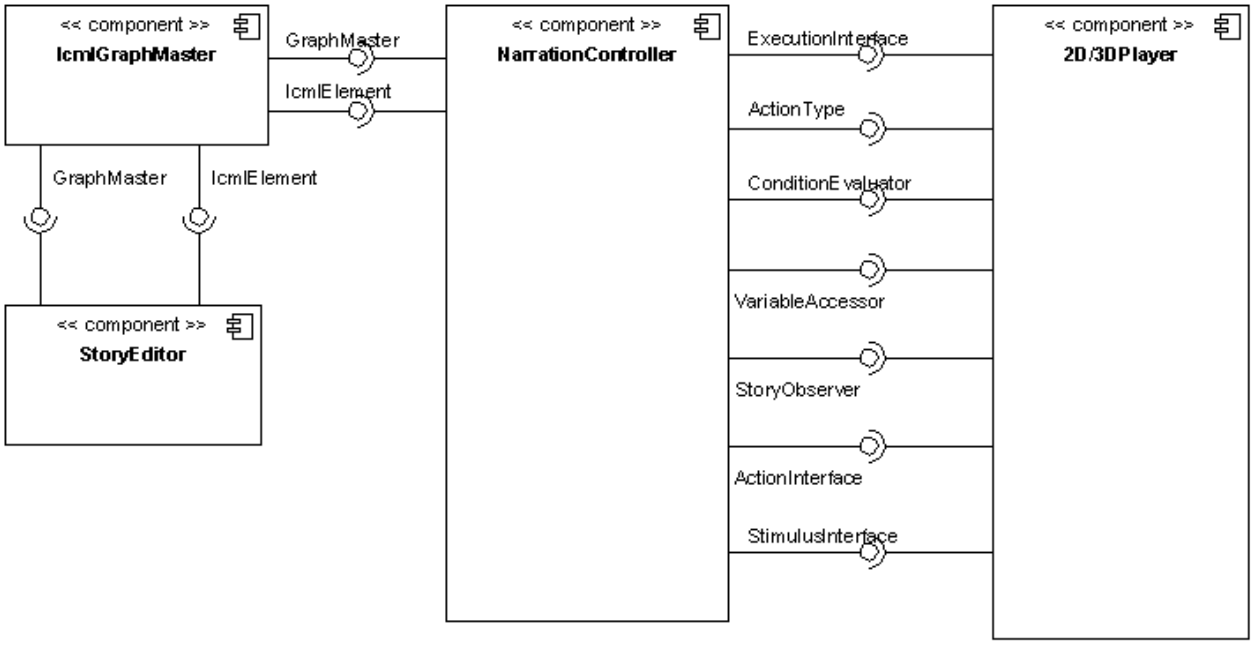

Abbildung 27: UML-Komponentendiagramm der neuen Architektur

7 Implementierungsdetails

7.1 Einleitung

Die NarrationController- und IcmlGraphMaster-Komponenten wurden, wie in Kapitel 6 beschrieben, umgestaltet und erweitert. Bei der Implementierung und dem Testen der neuen Funktionalitäten stellten sich weitere Detailfragen, die durch das Architektur-Design nicht vollständig vorgeschrieben sind. Dies ermöglicht eine partielle Implementierungsunabhängigkeit der Architektur. Für das Testen und die Verwendung im Inscape-Projekt müssen somit Entscheidungen zur konkreten Implementierung und den zu verwendenden Hilfsbibliotheken getroffen werden. Dieses Kapitel diskutiert eine Auswahl der zu entscheidenden Implementierungsdetails.

7.2 Verwendete Bibliotheken

Die Architektur wurde mit den Anforderungen entworfen, möglichst wenige explizite Abhängigkeiten zu weiteren Bibliotheken aufzuweisen. Das heißt, dass sie in einigen Anwendungsbereichen nicht auf spezielle Bibliotheken angewiesen ist, sondern die Bibliotheken leicht ausgetauscht werden können. Ein Beispiel hierfür ist die in IcmlGraphMaster benötigte XML-Bibliothek. Bisher wurde die XML-API der Qt-Bibliothek von Trolltech verwendet [40]. Diese bietet jedoch nicht alle Funktionalitäten, die durch die neuen Anforderungen vorausgesetzt werden. Daher muss die bisherige API durch eine neue ausgetauscht werden. Dank der in Kapitel 6.4.4 geschaffenen Abstraktionsschicht hält sich der Aufwand hierfür in Grenzen. In Kapitel 7.3 wird der Vorteil dieser Unabhängigkeit von einer konkreten XML-Bibliothek am Beispiel der Realisierung der XML-Funktionalität mit Hilfe der Bibliotheken Xerces und Xalan von Apache gezeigt [41], [42].

Die einzige explizite Abhängigkeit von NarrationController und IcmlGraphMaster besteht neben der Standard Template Library (STL) durch die Bibliotheken Loki und Boost. Diese bieten unter Zuhilfenahme von generischer Programmierung einige nützliche Funktionalitäten an und ermöglichen somit in vielen Bereichen die geforderte Flexibilität der erstellten Software-Architektur. Da Loki statisch mit dem NarrationController verknüpft wird und aus der Boost-Bibliothek nur einige Header-Dateien benötigt werden, sind diese Bibliotheken zum Ausführen des NarrationControllers nicht mehr erforderlich [43], [44].

Weitere verwendete Bibliotheken sind CppUnit für die Durchführung von UnitTests und Qt für die Entwicklung einer Demonstrations-Anwendung. In beiden Fällen sind NarrationController und IcmlGraphMaster jedoch nicht abhängig von diesen Bibliotheken. Stattdessen wurde mit ihrer Hilfe jeweils eine ausführbare Anwendung erstellt, die wiederum NarrationController und IcmlGraphMaster verwenden. Hierauf wird in den Kapiteln 7.4 und 7.5 weiter eingegangen [40], [45].

7.3 Implementierung einer konkreten XML-API

Die in Kapitel 5 gestellten Anforderungen an NarrationController und ICML bestanden unter anderem darin, sowohl eine Validierung als auch die Verwendung einer Skriptsprache innerhalb von ICML zu ermöglichen. Während der Entwicklung des Architektur-Designs in Kapitel 6 wurde sich dafür entschieden, XPath als Skriptsprache zu verwenden. Die bisher verwendete XML-API von Qt bietet beide Möglichkeiten nicht. Als neue XML-API wurde sich für die Nutzung der Apache-Bibliotheken Xerces und Xalan entschieden, da diese in Kombination einen großen Funktionsumfang bieten und auch einige Erweiterungsmöglichkeiten aufweisen. Auch die kostenlose Verfügbarkeit dank der OpenSource-Lizenz von Apache war ein ausschlaggebender Punkt. Xerces übernimmt die Aufgaben der bisherigen XML-API von Qt und erweitert diese um die Validierungsmöglichkeit von XML-Dokumenten. Diese wird im Gegensatz zu XPath von der C++-Version von Xerces nativ unterstützt. Durch Xalan lässt sich Xerces allerdings auch um XPath erweitern.

Unter Verwendung der in Kapitel 6.4.4 erstellten Abstraktionsschicht wird im Folgenden eine Implementierung der XML-Funktionalitäten von Xerces vorgenommen. Hierfür müssen die durch die Abstraktionsschicht vorgegebenen Adapter-Klassen XmlParser, XmlDocument und XmlNode spezialisiert werden. Da sowohl die Schnittstellen der entwickelten Abstraktionsschicht als auch die von Xerces W3C-konform entworfen sind, finden sich die drei genannten Klassen in ähnlicher Form auch in Xerces wieder. Es reicht also aus, die in den Adapter-Klassen deklarierten Methoden so zu überschreiben, dass sie den Aufruf an eine entsprechende Xerces-Klasse weiterleiten. Bei den so entstehenden Adapter-Spezialisierungen handelt es sich also um ObjectAdapter, wie auch das Beispiel anhand der Methode hasAttribute() in Listing 11 zeigt. Bei mNode handelt es sich um ein Member-Attribut, das auf eine Objekt-Referenz einer Xerces-Klasse zeigt.

```
bool XmlNodeImplXerces::hasAttributes()
{
    return mNode->hasAttributes();
}
```

Listing 11: Weiterleitung eines Methodenaufrufes an die Xerces-Implementierung

Der interessante Teil bei der Implementierung der konkreten XML-API besteht in der Realisierung des XPath-Zugriffs. Wie schon erwähnt, wird dieser von Xalan bereitgestellt. Wie Xerces bietet auch Xalan W3C-konforme Schnittstellen zum Zugriff auf XML-Daten an. Die Implementierung unterscheidet sich jedoch stark von Xerces. So ist Xalan als Bibliothek für XSLT-Transformationen darauf optimiert, verschiedene Anweisungen auf ein sich nicht veränderndes XML-Dokument anzuwenden, wie es z.B. auch bei XPath der Fall ist. Xerces hingegen ist auf eine möglichst effektive Veränderung von XML-Dokumenten optimiert. Es wäre sinnvoll, Xerces nur innerhalb der IcmlGraphMaster-

Komponente zu verwenden, die die Verwaltung und Veränderung von ICML und somit auch der Story-Definition enthält. Im NarrationController, wo das Abspielen der Story stattfindet und somit keine Veränderung an der eigentlichen Story-Definition mehr nötig ist, würde sich eher Xalan anbieten. Neben der Story-Definition existiert jedoch noch die Story-History, die vom NarrationController während des Abspielvorgangs aufgezeichnet wird. Diese wird ebenso wie die Story-Definition im XML-Format gehalten. Weiterhin sollte es möglich sein, auch während des Editierens einer Story innerhalb von IcmlGraphMaster per XPath Zugriff auf die ICML-Elemente zu haben.

Die Verbindung zwischen Xerces und Xalan wird somit nur in der Methode XmlNode::evaluateXPathExpression() benötigt. Beim Methodenaufruf findet eine Umwandlung des zugehörigen XML-Dokumentes in die entsprechende Xalan-Repräsentation statt. Über die nun verfügbare Xalan-Schnittstelle lässt sich eine XPath-Anweisung auf das Dokument anwenden. Das hierbei entstehende Resultat muss schließlich in seine Xerces-Repräsentation umgewandelt werden, um anderen Komponenten den gewohnten Zugriff auf die darin gekapselten Daten zu liefern.

7.4 UnitTests

Um die Applikation schon während der Entwicklung auf ihre korrekte Funktionalität hin zu überprüfen, werden UnitTests vorgenommen. Die notwendige Funktionalität liefert die Software-Bibliothek CppUnit, die dafür verschiedene Klassen zur Verfügung stellt. In dieser Implementierung wurden alle Tests über Ableitungen von der Klasse TestCase realisiert. Mit Hilfe von Makros werden sowohl die abgeleitete Klasse als auch deren Member-Methoden dem CppUnit-Framework bekannt gegeben. Das CppUnit-Framework stellt hierfür eine Registry zur Verfügung, die ähnlich einer Factory die einzelnen Test-Klassen in sich aufnimmt und zum Zeitpunkt der Ausführung des Tests instanziiert.

Die Member-Methoden der Test-Klassen enthalten die Anweisungen, deren Funktionalität getestet werden soll. Hierbei handelt es sich wiederum um Methodenaufrufe von Objekten der Klassen der NarrationController- bzw. IcmlGraphMaster-Komponente. Die Ergebnisse der Methodenaufrufe werden auf ihre Richtigkeit überprüft. Hierfür stellt CppUnit weitere Makros zur Verfügung. Als Beispiel wird in Listing 12 einem IcmlElement ein Chil-Element hinzugefügt. Zunächst wird geprüft, ob die Methode eine erfolgreiche Durchführung zurückmeldet. Im Anschluss wird kontrolliert, ob dem jeweiligen Parent-Element auch

```
IcmlElement* scenario = graph.getStory()
                            ->addChild(ElementType::SCENARIO, "testScenario");
CPPUNIT_ASSERT(scenario != NULL);
IcmlElement* strategySetRef = scenario
                            ->addChild(ElementType::POLICY_REF, "policyRef1");
CPPUNIT_ASSERT(strategySetRef != NULL);
CPPUNIT_ASSERT_EQUAL(strategySetRef, scenario->getElementById("policyRef1"));
```

Listing 12: UnitTests mit Hilfe von Makros aus der CppUnit-Bibliothek

wirklich das entsprechende Child-Element hinzugefügt wurde.

Um den Test möglichst komfortabel zu gestalten, wird er automatisiert. Mit Hilfe von Post-Build-Events in den Compiler-Einstellungen ist es möglich, direkt nach Kompilation des Codes Anweisungen auszuführen. Dieses wird sich hier zu nutze gemacht, indem die einzelnen Compiler-Erzeugnisse in ein Verzeichnis kopiert werden und von dort direkt der UnitTest ausgeführt wird. Dadurch wird automatisch nach jedem Kompiliervorgang der UnitTest gestartet. Wie in Abbildung 28 zu sehen ist, lässt sich somit sofort feststellen, ob neue Änderungen am Code die bisherige Funktionalität beeinträchtigen. So lassen sich mögliche Fehlerquellen frühzeitig erkennen und entfernen.

Die durchgeführten Tests beschränken sich auf den Aufbau des Story-Graphen und die notwendigen Schritte zur Vorbereitung des Abspielens einer Story. Der Abspielvorgang lässt sich schlecht mittels UnitTests überprüfen, da dieser von der Interaktion des Anwenders abhängt und somit ein klar definierbarer Test-Fall, der auf ein bestimmtes Ergebnis überprüft werden kann, nicht realisierbar ist. Der Abspielvorgang muss demnach manuell getestet werden, wofür eine weitere Applikation erstellt wird, die eine Player-Komponente implementiert.

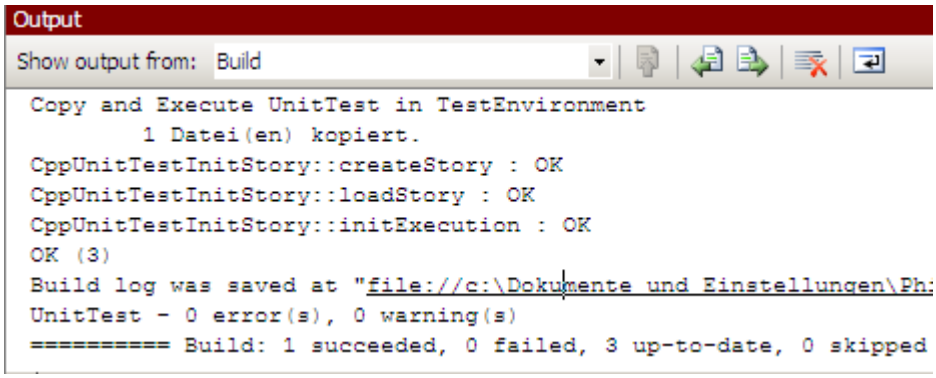

Abbildung 28: Ausgabefenster des Compilers inklusive UnitTest-Ergebnis

7.5 Test-Player-Implementation

Um den Abspielvorgang, vor allem aber verschiedene Strategien der Ablaufoptimierung auf ihre Auswirkungen hin zu überprüfen, wird eine Abspiel-Komponente benötigt, die im Inscape-Framework Player genannt wird. Bei DemonstrationApp handelt es sich um einen sehr simplen Player auf Textbasis. Die einzelnen Ereignisse während des Story-Verlaufs werden in ein Textfenster ausgegeben, die Interaktionsmöglichkeiten des Anwenders über eine Auswahlliste an Stimuli dargestellt. Weiterhin bietet die Anwendung die Möglichkeit, die verwendete Story und deren History in einem einfachen XML-TreeView zu betrachten. Mit ihrer Hilfe soll die Korrektheit der Ablauflogik manuell überprüft werden können.

Für die Realisierung wurde Qt verwendet. Zum einen stellt diese Software-Bibliothek ein objektorientiertes Framework zur Erstellung von GUIs zur Verfügung zum anderen enthält sie weitere nützliche Funktionalitäten, wie z.B. eine Timer-Klasse. Ein weiterer Vorteil von Qt ist dessen Plattformunabhängigkeit.

Der NarrationController wird mit seiner Plugin-Implementation initialisiert (siehe Kapitel 6.4.3), so dass die Hauptschleife vom Player bereitgestellt werden muss. Hierfür wird ein QTimer benutzt, mit dem sich bestimmte Methoden zu bestimmten Zeitpunkten aufrufen lassen. In diesem Fall wird die update()-Methode des ExecutionInterfaces alle 0.6 Sekunden aufgerufen. Um dies zu ermöglichen, wird das Hauptfenster der Applikation, die Klasse DemonstrationWidget, bei dem Timer angemeldet. Jede Fenster-Klasse in Qt muss von der allgemeinen Klasse QObject ableiten, die eine Methode timerEvent() besitzt und von Timern aufgerufen werden kann. Es handelt sich um ein ähnliches Prinzip wie die Clock des NarrationControllers (siehe Kapitel 6.3.3). In DemonstrationWidget wird diese Methode reimplementiert, um schließlich die update()-Methode aufzurufen. Dies wird innerhalb eines try/catch-Blockes getan, um auftauchende Exceptions abzufangen, die ansonsten zu einem Absturz der Anwendung führen würden (Listing 13).

```
void DemonstrationWidget::timerEvent(QTimerEvent* e)
{
    try
    {
        mNarrationController->update();
    }
    catch(const Exception& e)
    {
        cout << endl << "An exception has been thrown: " << endl;
        cout << e;
        mTimer.stop();
    }
}
```

Listing 13: Realisierung einer Hauptschleife über QTimer

Bei DemonstrationWidget handelt es sich nicht nur um ein QObject, das die einzelnen Fenster-Elemente erzeugt und verwendet. Es handelt sich zusätzlich um eine Spezialisierung von ExecutionListener und ist somit in der Lage, über eintretende StoryEvents informiert zu werden. Hierfür muss die abstrakte update()-Methode von ExecutionListener implementiert werden. Die vorgenommene Implementation sieht vor, eintretende StoryEvents zunächst über einen switch/case-Block zu typisieren, um dann eine entsprechende Textmeldung über den Eintritt des StoryEvents auszugeben.

Der NarrationController implementiert bestimmte Basis-Typen für Conditions und Actions, die unabhängig von einer Player-Komponente verwendet werden können. Zum Testen des Plugin-Systems implementiert der Player zusätzlich eine eigene Action, die dem NarrationController hinzugefügt wird. Da es sich um einen textbasierten Player handelt, wurde sich für eine simple „displayText"-Action entschieden. Der Sourcecode der von ActionType abgeleiteten Klasse DisplayTextAction ist auf der beiliegenden CD zu finden (siehe Anhang 2).

Am wichtigsten für einen Test des Abspielvorgangs ist es jedoch, dem Anwender die Möglichkeit zu geben, in die Story eingreifen zu können und somit Stimuli auszulösen. Hierfür wird das vom NarrationController angebotene StimulusInterface spezialisiert. Die Klasse StimuliListener erbt von StimulusInterface und wird über das DemonstrationWidget als solches beim ExecutionInterface registriert. Dadurch wird StimuliListener über aktive Stimuli während des Abspielvorgangs informiert. Diese Informationen werden über das Signal/Slot-Prinzip von Qt an das DemonstrationWidget weitergeleitet und dem Anwender auf dem Bildschirm in einer Auswahlliste angezeigt. Hier kann er nun einen Stimuli auslösen, um einen Einfluss auf die Story zu nehmen.

7.6 Zusammenfassung

Um die neuen Funktionalitäten des NarrationControllers zu testen, ist es notwendig gewesen, eine konkrete Implementation bestimmter Bestandteile der Komponente vorzunehmen. Durch die in Kapitel 6.4.4 geschaffene Abstraktionsschicht für XML-APIs musste eine Bibliothek gefunden werden, die die abstrakten Klassen spezialisiert und somit die XML-Funktionalität zur Verfügung stellt. Dies ist mit Xerces und Xalan geschehen. Weiterhin stellte es sich als sinnvoll heraus, sowohl automatisierte als auch manuell durchführbare Tests über einfach zu bedienende Oberflächen. Vor allem die Realisierung der DemonstrationApp stellt sich als eine große Hilfe heraus, um simple Strategien auf textbasis auszuprobieren. Dadurch kann herausgefunden werden, welche Einsatzgebiete für den NarrationController schon vorstellbar sind und welche Erweiterungen noch nötig sind, um komplexere Strategien zur Ablaufoptimierung zu entwickeln.

8 Einsatzmöglichkeiten

8.1 Einleitung

Die während dieser Diplomarbeit entstandene Software-Architektur erlaubt durch ihre Flexibilität nahezu beliebige Konzepte der Ablaufoptimierung zu verschiedenen Strategien zusammenzustellen. Einmal entwickelte Strategien lassen sich theoretisch auf beliebige Stories anwenden. Eine Anpassung auf eine spezifische Story wird sich aber in vielen Fällen, vor allem bei Konzepten zur inhaltlichen Ablaufoptimierung, nicht vermeiden lassen. Die Funktionalität einer Ablaufoptimierung lässt sich somit nur anhand einer bestehenden Story zeigen, die für diese Diplomarbeit nicht zur Verfügung steht.

Ein Ausblick auf die vorstellbaren Einsatzmöglichkeiten kann dennoch gegeben werden. Schon durch die grundlegenden Funktionalitäten, die der Narration-Controller aufgrund der implementierten Ablauflogik zur Verfügung stellt, lassen sich in Kombination mit XPath und der in Kapitel 7 vorgestellten DemonstrationApp kleinere Beispiele erstellen.

8.2 Änderung der Ausführungsgeschwindigkeit

Dank der neuen „inneren Uhr" des NarrationControllers ist es möglich, die Ausführungsgeschwindigkeit des Story-Ablaufs zu skalieren (vgl. Kapitel 6.3.3). Da alle an der Ausführung der Story beteiligten Komponenten die Zeitwerte von diesem skalierbaren Zeitmesser erhalten, hat eine Änderung des Skalierungsfaktors globale Auswirkung. Ein Zugriff auf diesen Faktor, z.B. über die VariableAccessor-Schnittstelle, erlaubt eine Beschleunigung oder Verlangsamung der Ausführungsgeschwindigkeit. Sollte sich ein Anwender zu viel Zeit in der bisherigen Story gelassen haben, so könnte die Geschwindigkeit also verdoppelt und somit eventuell verlorene Zeit wieder aufgeholt werden. Neben dem eigentlichen Eingriff gehört zu einer Strategie noch die Entscheidungsfindung dazu.

Natürlich könnte der Story-Autor einfach einen festen Wert angeben, ab dem eine Beschleunigung der Story durchgeführt werden soll, z.B. über das expectedTime-Attribut einer Szene (siehe Abbildung 2). Dies wäre allerdings eine zu strikte Vorgehensweise, zumal ohne vorherige Testdurchläufe keine Aussage über die Dauer der Story getroffen werden kann. Besser wäre hier die Verwendung des averageRunTime-Wertes, der von StoryHistory berechnet wird. Der Zugriff auf ICML-Attribute während der Laufzeit zur Überprüfung von Werten kann dank der XPath-Funktionalität erfolgen.

Selbst mit einer solchen Entscheidungsfindung ist die Beeinflussung der Ausführungsgeschwindigkeit eine sehr starre Strategie. Sie ist ein Kriterium, das wahrscheinlich besser in die Kontrolle des Anwenders gegeben wird. So könnte ein unerwartetes Drosseln der Geschwindigkeit zu Langeweile führen, eine

Beschleunigung den Anwender überfordern. Daher sollte der Anwender für sich selber festlegen können, in welcher Geschwindigkeit die Story abgespielt werden soll, ähnlich wie mit Videoabspielgeräten. Auch in einigen Spielen wird es dem Anwender ermöglicht, an bestimmten Stellen die Ausführungsgeschwindigkeit zu beeinflussen, vielen ist diese Art der Eingriffsmöglichkeit also vertraut. Um diese Möglichkeit zu bieten, muss jedoch die Player-Komponente eine Benutzerschnittstelle zur Verfügung stellen, die eine Geschwindigkeitsveränderung erlaubt. Eine Änderung des Anwenders muss dann an den Narration-Controller weitergeleitet werden, damit dort die Anpassung des Skalierungsfaktors erfolgen kann.

8.3 Priorisierung

Anstatt mit Geschwindigkeitsanpassungen lässt sich die Zeit auch mit der Beeinflussung des Informationsflusses kontrollieren. In einer Story verbraucht jedes eintretende Ereignis eine gewisse Zeit, genauso wie die Entscheidungen des Anwenders, Einfluss auf die Story zu nehmen. Wenn ein vorgegebenes Zeitlimit erreicht wird, sollten weniger wichtige Informationen und somit Ereignisse ausgelassen werden. Es findet also eine Priorisierung der Ereignisse statt. Eine solche Vorgehensweise muss jedoch nicht auf eine Verwendung zur Kontrolle der Zeit beschränkt sein. Eine Priorisierung könnte ebenso aufgrund einer erkannten „Gameplay Gestalt" erfolgen (z.B. um nur die zu der ermittelten „Gameplay Gestalt" passenden Ereignisse zu verwenden), oder aber es wird eine Priorisierung von Strategien vorgenommen. Strategien, die sich nur in ihrem Eingriff nicht aber in der Entscheidungsfindung unterscheiden, können dadurch in Schweregrade eingeteilt werden. So könnten zuerst subtilere Strategien angewandt werden, die nur kleine Veränderungen hervorrufen. Wenn diese zu keinen Veränderungen im Verhalten des Benutzers führt, werden Strategien verwendet, die stärkere Eingriffe in die Story vornehmen.

Aufgrund der unterschiedlichen Möglichkeiten zur Verwendung von Priorisierung sollte ein heuristischer Wert für die Prioritätsangabe Anwendung finden. Je nach Bedürfnis kann somit eine andere Heuristik verwendet werden. Die Priorität kann hierbei entweder ein Element-Attribut oder eine Variable sein. Da Variablen in ICML eindeutige Namen besitzen müssen und diese somit nicht überladen werden können, ist ein allgemeiner Variablenzugriff nur über Umwege möglich. So kann es z.B. nicht für jede Szene eine Variable „priority" geben. Stattdessen müsste jede Variable mit einem eindeutigen Prefix versehen werden, wie z.B. „scene1_priority". Weiterhin ist es für Strategien gar nicht möglich, über eigenständige Variablen zu verfügen. Der Zugriff über Attribute ist um einiges einfacher, da hierfür die Möglichkeit der Verwendung von XPath besteht und somit beliebige Elemente mit einer Priorität versehen werden können.

Die Anwendung von XPath-Anweisungen soll im Weiteren am Beispiel einer Strategie-Priorisierung verdeutlicht werden. Auf eine Test-Story mit zwei Szenen, die mit einer Transition verbunden sind, werden hierfür drei Strategien

angewandt. Wie zuvor vorgeschlagen, unterscheiden sich die Strategien nur in der Schwere ihres Eingriffes. Als Heuristik für die Priorität werden einfache Zahlenwerte von 1-3 gewählt, die den Prioritätsstufen „unwichtig", „normal" und „wichtig" entsprechen. Nach Anwendung einer unwichtigen Strategie wird die Prioritätsstufe automatisch erhöht, so dass bei der nächsten Entscheidungsfindung ein höherer Priorität gewählt wird, die dementsprechend stärker in den Story-Verlauf eingreift.

Das Beispiel wird innerhalb der in Kapitel 7.5 beschriebenen DemonstrationApp getestet. Die zu verwendenden Strategien sollen den Anwender bei der Durchführung eines Szenenwechsels helfen, der durch einen Stimulus ausgelöst werden kann. Strategie 1 weist den Anwender nach 5 Sekunden auf die Möglichkeit der Stimulus-Benutzung hin. Nach weiteren 5 Sekunden greift Strategie 2, die dem Anwender explizit den Stimulus nennt, der den Szenenwechsel hervorruft. Wenn der Anwender nach wie vor keine Reaktion zeigt, wird nach wiederum 5 Sekunden Strategie 3 aktiviert, die den Stimulus automatisch auslöst und somit den Szenenwechsel selbstständig durchführt.

Die Realisierung dieser Strategie-Priorisierung lässt sich vollständig in ICML vornehmen. Zunächst werden die drei Strategien mit ihren Prioritäten definiert (Listing 14). Da der Zweck des Eingriffes bei allen drei Strategien derselbe ist, eignen sie sich für die Zusammenfassung zu einer Story-Policy (siehe Kapitel 6.5.6).

Die Anforderungen an die Strategien werden im Folgenden für die erste Strategie umgesetzt. Der dabei entstehende ICML-Code wird in Listing 15 veranschaulicht. Das vollständige Beispiel ist auf der beiliegenden CD einsehbar (siehe Anhang 2). Zunächst werden die Bedingungen, die zu einer Anwendung der Strategie führen sollen, definiert. Die Anforderungen besagen, dass die Strategien nur auf der für sie vorgesehenen Prioritätsstufe aktiv sein sollen. Diese muss also mit dem Prioritätswert der Strategie verglichen werden. Hierfür wurde die globale Story-Variable priorityLevel eingeführt. Der Zugriff auf das damit zu vergleichende Attribut der Strategie lässt sich wie erwartet über XPath auslesen.

```
<strategies>
    <strategy nameID="strategy1" priority="1"/>
    <strategy nameID="strategy2" priority="2"/>
    <strategy nameID="strategy3" priority="3"/>
    <policy nameID="policy1">
        <strategyRef refID="strategy1"/>
        <strategyRef refID="strategy2"/>
        <strategyRef refID="strategy3"/>
    </policy>
 </strategies>
```

Listing 14: ICML-Definition von Strategien

Weiterhin soll ein Eingriff erst nach 5 Sekunden erfolgen, der NarrationController bietet für solche Zwecke eine Condition vom Typ „timeout" an. Die notwendige UND-Verknüpfung der beiden Bedingungen lässt sich wie in Kapitel 6.5.3 beschrieben durch eine Verschachtelung der Elemente erreichen. Zuletzt ist noch der Eingriff zu definieren, der hier über eine simple Textausgabe mit Hilfe der von der DemonstrationApp hinzugefügten DisplayText-Action erfolgt. Weiterhin wird die Prioritätsstufe erhöht, um bei der nächsten Überprüfung eine Strategie höherer Priorität eingreifen zu lassen.

```
<condition type="equals">
    <param type="var" name="left" value="priorityCheck"/>
    <param type="xpath" name="right"
           value="//strategies/strategy[@nameID='strategy1']/@priority" />
    <condition type="timeout">
        <param name="seconds" value="5"/>
            <action type="displayText">
                <param name="text" value="Please trigger stimulus"/>
            </action>
            <action type="setVariable" target="priorityCheck">
                <param name="inc" value="1"/>
            </action>
    </condition>
</condition>
```

Listing 15: ICML-Definition von Strategien

Die Definition der restlichen Strategien erfolgt analog. Die hier vorgestellten Strategien lassen sich nun jederzeit in einer Story anwenden und auf individuelle Bedürfnisse anpassen. Die Veränderung der Prioritätsstufe muss nicht zwingend in den Strategien selbst erfolgen. Da es sich um eine globale Story-Variable handelt, kann die Priorität jederzeit verändert und somit die Anwendung einer bestimmten Strategie erzwungen werden. Ebenso kann eine andere Prioritäts-Heuristik Anwendung finden. Eine weitere Möglichkeit wäre, den Condition-Typ „timeout" durch eine andere Entscheidungsfindung auszutauschen. Sobald die Voraussetzungen für eine Mustererkennung gegeben sind, lässt sich eine Realisierung des in Kapitel 4.4.2 vorgeschlagenen Konzeptes, Hilflosigkeit in der Verhaltensweise der Anwender zu erkennen, in Betracht ziehen.

Der vorgestellte ICML-Code lässt sich somit als Prototyp verwenden, in dem bestimmte Konzepte als Platzhalter dienen können, die zu einem späteren Zeitpunkt durch komplexere ausgetauscht werden können. So lässt sich auf Basis dieses Beispieles ebenso die in Kapitel 4.5.1.2 beschriebene Szenen-Priorisierung mit nur wenigen Änderungen umsetzen.

8.4 Inhaltsbezogene Strategien

Ohne eine konkrete Story ist es nur schwer möglich, die Auswirkung einer Ablaufoptimierung unter Verwendung inhaltsbezogener Strategien demonstrieren. Neben der Abhängigkeit von storyspezifischen Ereignissen erschwert das Streben nach möglichst subtilen Vorgehensweisen, unter Verwendung der zur Verfügung stehenden Mitteln, eine geeignete Darstellung solcher Strategien. Als Beispiel sei hier nochmal das von Chris Crafword genannte Konzept der Persönlichkeitsveränderung von computergesteuerten Charakteren genannt. Dieses besonders erfolgsversprechende Konzept sorgt für eine andere Verhaltensweise gegenüber dem Anwender, was wiederum eine Reaktion des Anwenders provoziert, die ihm wiederum neue Anstöße und Ideen geben könnte, um in der Story erfolgreich voran zu schreiten.

Um eine Verhaltensänderung in Charakteren auszulösen, ist jedoch ein Multi-Agent-System notwendig, welches die Charaktere mit einem Persönlichkeitsmodell ausstattet und damit die Verhaltensweisen und Ziele der Charaktere verwaltet und steuert. Zum Zeitpunkt dieser Diplomarbeit ist ein solches System noch nicht im Inscape-Framework realisiert. Letztendlich handelt es sich hierbei aber auch um das Verändern von Variablenwerten, so dass die notwendigen Funktionalitäten, wie z.B. ein CharacterVariableAccessor, zu einem späteren Zeitpunkt, wenn ein Multi-Agent-System zur Verfügung steht, dem NarrationController hinzugefügt werden können.

8.5 Zusammenfassung

Die Flexibilität der im Verlauf der Diplomarbeit entstandenen NarrationController-Architektur wurde auf ihre Einsatzmöglichkeiten überprüft, indem versucht wurde, eine Auswahl der in Kapitel 4 genannten Konzepte als konkrete Beispiele in ICML zu realisieren. Die Software-Architektur hielt mit der Realisierung einer Strategie-Priorisierung ihrem ersten Praxistest stand. Die Grundfunktionalität einer Ablaufoptimierung kann somit als gewährleistet gelten. Schon bei diesem ersten Prototyp zeichnet sich ab, dass bei der Entwicklung von Strategien diese möglichst allgemein gehalten werden sollten, um sie mit wenigen Anpassungen in unterschiedlichen Stories wiederverwenden zu können.

9 Fazit

Diese Diplomarbeit hat sich mit dem Thema einer Ablaufoptimierung in computergestützten Interactive Storytelling-Systemen auseinander gesetzt. Hierbei wurde zunächst beleuchtet, was diesbezüglich unter einer Ablaufoptimierung zu verstehen ist und was sich von ihr erhofft wird. Im Speziellen wurde auf die Probleme, die der Einsatz einer Ablaufoptimierung in Computer-Systemen mit sich bringt, eingegangen und welche Schritte bisher unternommen wurden, um Interactive Storytelling in computergestützten Systemen zu betreiben. Hierbei stellte sich heraus, dass eine Ablaufoptimierung eine Balance zwischen den durch einen Story-Autor aufgebauten Story-Strukturen und den Bedürfnissen des Anwenders herstellen muss. Diese Bedürfnisse müssen von einem Interactive Storytelling-System erkannt werden, um anhand der Vorgaben des Story-Autors Entscheidungen zu treffen, die eine Anpassung des Story-Verlaufs vornehmen. Sinn dieser Anpassung ist es, den Anwender in einen Zustand des Flows zu versetzen und diesen solange wie möglich aufrecht zu erhalten, um dem Anwender ein möglichst optimales Erlebnis zu bieten.

Um Lösungsansätze für eine Ablaufoptimierung zu liefern, wurden mögliche Konzepte gesammelt und in Kategorien eingeteilt. Angelehnt an die notwendigen Kriterien für eine Interaktion wurde hierbei eine Unterteilung in Überwachung, Entscheidungsfindung und Eingriff vorgenommen und diese als die drei Hauptbestandteile einer Ablaufoptimierung identifiziert. Eine Ablaufoptimierung benötigt aus jeder dieser Kategorien mindestens ein Konzept um zu funktionieren. Aus dieser Erkenntnis heraus wurde ein Verfahren entwickelt, mit dem sich mehrere Konzepte aus unterschiedlichen Kategorien zu Strategien der Ablaufoptimierung zusammenfassen lassen.

Die durch das Inscape-Projekt zur Verfügung gestellte NarrationController-Komponente sollte im Weiteren um ein solches Verfahren erweitert werden, um es Story-Autoren zu ermöglichen, verschiedene Strategien zur Ablaufoptimierung zu erstellen und diese auf ihre Story anzuwenden. Hierbei stellte sich heraus, dass die benötigten Funktionalitäten einer Ablaufoptimierung keine einfache Erweiterung eines Systems darstellen, sondern teilweise tiefgreifende Veränderungen in der bestehenden Software-Architektur verursachen. Diese und weitere externe Anforderungen an die NarrationController-Architektur machten ein umfassendes Refactoring notwendig. Dieses Refactoring wurde mit Hilfe von Design-Patterns durchgeführt, indem versucht wurde, die einzelnen funktionalen Bestandteile der zu erstellenden Architektur als Design-Pattern zu identifizieren. Bei einer Übereinstimmung mit dem Verwendungszweck eines Design-Patterns wurde dieses als Basis für die Entwicklung des jeweiligen Bestandteils verwendet. Wenn sich die Möglichkeit ergab, wurde auch eine Basis-Implementation vorgenommen. Um diese in möglichst vielen Fällen anwenden zu können, wurden sie mit Hilfe generischer und auf Policies basierender Programmierung realisiert. Dadurch konnte eine Basis-Implementation vielfältig an konkrete Anwendungsfälle angepasst werden, wie am Beispiel

des Observer-Patterns ausführlich beschrieben wurde.

Die wichtigste Refactoring-Maßnahme bestand in der Trennung von Ablauflogik und Datenmodell. Um eine bestmögliche Erweiterbarkeit und Wiederverwendbarkeit zu gewährleisten, darf das Datenmodell kein explizites Wissen über die Ablauflogik haben. Daher wurden Datenmodell und Ablauflogik in die zwei eigenständigen Bibliotheken NarrationController und IcmlGraphMaster aufgeteilt. Dies garantiert die Unabhängigkeit des Datenmodells von der Ablauflogik, erfordert aber auch die Bereitstellung neuer Zugriffsmöglichkeiten der Ablauflogik auf das Datenmodell.

Um dem NarrationController verschiedene Konzepte der Ablaufoptimierung hinzufügen zu können, wurden die Schnittstellen ActionType, ConditionEvaluator und VariableAccessor erstellt. Jede dieser Schnittstellen deckt hierbei eine der drei Hauptkategorien der Konzepte zur Ablaufoptimierung ab. Der Zugriff auf das Datenmodell wird von den Schnittstellen sowohl über C++ als auch über die Skriptsprache XPath ermöglicht, der eine besondere Bedeutung zukommt. XPath wurde entwickelt, um einen komfortablen Zugriff auf XML-Daten zu ermöglichen. Dies macht XPath zu der besten vorstellbaren Schnittstelle zwischen hardcodiertem C++-Quellcode und den auf XML basierenden ICML-Story-Strukturen. Weiterhin ist XPath auf eine ähnliche Weise wie die neue Architektur des NarrationConrollers um zusätzliche Funktionen erweiterbar. Durch Plugins lassen sich die in XPath zur Verfügung stehenden Befehle erweitern und anpassen. So lassen sich auch komplexere Algorithmen in C++ implementieren, die sich dann innerhalb einer XPath-Anweisung zusammen mit anderen XPath-Befehlen wiederverwenden lassen.

Eine solche Kombination der einzelnen Plugins ist auch für die Konzepte der Ablaufoptimierung erwünscht, um sie zu wiederverwendbaren Strategien zusammenzufassen. Hierfür wurden einige strukturelle Änderungen in ICML vorgenommen, um eine bequeme Zusammenstellung verschiedener Konzepte zu ermöglichen. Die drei vom NarrationController angebotenen Schnittstellen haben dabei ihre jeweilige Entsprechung in den ICML-Elementen Action, Condition und Variable gefunden. Die Kombination der einzelnen Erweiterungsmöglichkeiten über die Plugin-Schnittstellen von XPath und NarrationController machen diesen somit zu einem mächtigen Framework. Die Erweiterungen können je nach Bedürfnis sowohl storyspezifische als auch allgemeine Zusatzfunktionalitäten enthalten, die sich sogar mit anderen Erweiterungen kombinieren und somit wiederverwenden lassen. Damit lässt sich der NarrationController in einer Vielzahl unterschiedlicher Einsatzgebiete anwenden. Eine derartige Flexibilität setzt aber voraus, dass für eine Verwendung des NarrationControllers immer eine Anpassung auf den jeweiligen Anwendungsbereich erfolgen muss. Diese Aufgabe kommt der Player-Komponente zu, die die Schnittstelle zwischen Anwender und der durchführbaren Story zur Verfügung stellt. Im Laufe der Konzeptentwicklung hat sich herausgestellt, dass nicht nur der im Hintergrund laufende NarrationController zur Ablaufoptimierung beiträgt, sondern auch die Player-Komponente, indem sie eine für den Anwender

angemessene und erwartungskonforme Benutzerschnittstelle anbietet. So sehen es die meisten Konzepte zur zeitgesteuerten Ablaufoptimierung vor, dass der Anwender selbst eine gewisse Kontrolle über die Zeit während der Ausführung bekommt. Neben den notwendigen Anpassungen des NarrationControllers durch die Player-Komponente spielt die eigene Erweiterbarkeit und Anpassungsfähigkeit der Player-Komponente somit auch eine wichtige Rolle für die Realisierung einer zufriedenstellenden Ablaufoptimierung.

Im Zeitrahmen der Diplomarbeit war es leider nicht mehr möglich, die vom Inscape-Framework verwendete Player-Komponente und die damit verbundenen Darstellungsmöglichkeiten auf die neue NarrationController-Architektur anpassen zu lassen. Daher wurde eine Eigenimplementation eines Players auf Textausgabe-Basis vorgenommen. Hiermit konnten im begrenzten Rahmen Tests der zuvor entworfenen Konzepte zur Ablaufoptimierung durchgeführt werden. Aufgrund einer fehlenden Beispiel-Story und den begrenzten Darstellungsmöglichkeiten der textbasierten Player-Komponente lässt sich die Wirkung einer Ablaufoptimierung jedoch nicht zufriedenstellend nachvollziehen.

Die Grundlagen zur Verwendung einer Ablaufoptimierung in computergestützten Interactive Storytelling-Systemen wurden mit dieser Diplomarbeit geschaffen. Basierend auf den Ergebnissen dieser Diplomarbeit können umfassende Tests erfolgen, anhand denen Anpassungen und Erweiterungen des NarrationControllers vorgenommen werden können. Die Durchführung einer Usability-Untersuchung in einem Test-Labor mit Probanden dürfte sich wahrscheinlich als die beste Möglichkeit erweisen, um verschiedene Strategien zur Ablaufoptimierung zu realisieren und auf ihre Gebrauchstauglichkeit zu überprüfen. Durch die Aufzeichnungen, die der NarrationController für jeden erfolgreichen Story-Durchlauf vornimmt, lassen sich Statistiken aufstellen. Zusammen mit einer Befragung der Probanden lassen sich Rückschlüsse auf die Qualität des Anwendererlebnisses ziehen und somit auch die Wirkung verschiedener Strategien feststellen.

Hierbei werden storyspezifische Strategien wahrscheinlich ein besseres Ergebnis liefern. Der Nachteil ist, dass sie für nur eine Story verwendet werden können, was den Aufwand der Erstellung möglicherweise nicht rechtfertigt. Für die Wiederverwendbarkeit von Strategien sollten allgemeinere Konzepte bevorzugt werden. Wie realistisch es ist, allgemeine Konzepte für eine Vielzahl von Stories zu finden ist jedoch schwer abzuschätzen. Die Möglichkeiten der Parametrisierung von ICML und der flexiblen Erweiterbarkeit des NarrationControllers stellen in dieser Hinsicht eine Erleichterung für das Finden und Verwenden solcher Konzepte und Strategien dar.

Glossar

Design Patterns

Dieser aus dem Gebiet der Architektur stammende Begriff bezeichnet ein Muster, bzw. eine Schablone zur Lösung eines bekannten Design-Problems. Seit Anfang der 90er Jahre hat der Begriff auch Einzug in den Bereich der Software-Entwicklung gefunden, da man auch dort auf Entwurfsproblemen stieß. Im Folgenden sind die in dieser Diplomarbeit verwendeten Design Pattern beschrieben.

Adapter

„The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.“ [31]

Composite

„The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.“ [31]

Decorator

„The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.“ [31]

Facade

„The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.“ [31]

Factory Method

„The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclass.“ [31]

Abstract Factory

„The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.“ [31]

Flyweight

„A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context – it's indistinguishable from an instance of the object thats not shared. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between intrinsic and extrinsic state. Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it shareable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.“ [30]

Observer

„The Observer Pattern defines a one-to-many dependeny between objects so that when one object changes state, all of its dependents are notified and updated automatically.“ [31]

Visitor

„With the Visitor pattern, you define two class hierarchies: one for the elements being operated on (the Node hierarchy) and one for the visitors that define operations on the elements (the Node Visitor hierarchy). You create a new operation by adding a new subclass to the visitor class hierarchy. As long as the grammar that the compiler accepts doesn't change (that is, we don't have to add new Node subclasses), we can add new functionality simply by defining new Node Visitor subclasses.“ [31]

Immersion

Dieser Begriff bezeichnet das Eintauchen in eine virtuelle Welt. Im Falle von Interactive Storytelling findet das aufgehen in dieser fiktiven Welt auf einem viel intensiveren Level statt, durch die Möglichkeit, direkt mit der Umgebung interagieren zu können.

Refactoring

„A refactoring is a „behaviour-preserving transformation“ or, as Martin Fowler defines it, „a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.“ [29], [32]

Literaturverzeichnis

[1] Crawford, C.: "On Interactive Storytelling", New Riders, 2005

[2] Aarseth, E. J.: "Cybertext: Perspectives on Ergodic Literature", The Johns Hoplins University Press, 1997

[3] "Interactive fiction" http://en.wikipedia.org/wiki/Interactive_fiction, Wikipedia, Stand: Juni 2006

[4] Crawford, C.: "Flawed Methods for Interactive Storytelling", http://www.erasmatazz.com/library/JCGD_Volume_7/Flawed_Methods.html

[5] Glassner, A.: "Interactive Storytelling. Techniques for 21st Century Fiction", AK Peters, Ltd., 2004

[6] Mateas, M., Stern, A.: "Façade: An Experiment in Building a Fully-Realized Interactive Drama", 2003
http://www.interactivestory.net/papers/MateasSternGDC03.pdf

[7] Begg, M., Ellaway, R., Dewhurst, D., Macleod, H.: "Virtual Patients: considerations of narrative and game play", aus Digital Game Based Learning, Universitätsverlag Karlsruhe, 2005

[8] Spierling, U.: "Learning with Digital Agents - Integration of Simulation, Games, and Storytelling", aus Digital Game Based Learning, Universitätsverlag Karlsruhe, 2005

[9] Hoffman, A., Iurgel, I., Becker, F.: "Interactive Drama and Learning Experiences", aus Digital Game Based Learning, Universitätsverlag Karlsruhe, 2005

[10] Peinado, F., Gervás, P.: "Transfering Game Mastering Laws to Interactive Digital Storytelling", aus Technologies for Interactive Digital Storytelling and Entertainment, Springer, 2004

[11] Csikszentmihalyi, M.: "Flow: The Psychology of Optimal Experience", Harper Perennial, 1991

[12] "Flow" http://en.wikipedia.org/wiki/Flow_%28psychology%29, Wikipedia, Stand: 17.Oktober 2006

[13] "Flow" http://de.wikipedia.org/wiki/Flow_%28Psychologie%29, Wikipedia, Stand: 17.Oktober 2006

[14] Propp, W.: "Morphologie des Märchens", Carl Hanser Verlag, 1972

[15] Magerko, B.: "Mediating the Tension between Plot and Interaction", 2005
http://www.eecs.umich.edu/~magerko/research/magerko.imagina.2005.pdf

[16] Lindley, C.A.: "The Gameplay Gestalt, Narrative, and Interactive Storytelling", Computer Games and Digital Cultures Conference, 2002

[17] Lindley, C.A.: "Narrative, Game Play and Alternative Time Structures for Virtual Environments", aus Technologies for Interactive Digital Storytelling and Entertainment, Springer, 2004

[18] Falstein, N.: "Introduction", aus Game Design: Theory & Practice, 2005

[19] Eladhari, M.: "Object Oriented Story Construction in Story Driven Computer Games", Stockholm University, 2002

[20] Lindley, C.A.: "Story and Narrative Structures in Computer Games", aus Developing Interactive Narrative Content, sagas/sagasnet reader, 2005

[21] Böttcher, R.A.: "Flow in Computerspielen", Otto-von-Guericke-Universität Magdeburg, 2005

[22] Pfeifer, B.: "Narrative Combat: Using AI to Enhance Tension in an Action Game", aus Game Programming Gems 4, Charles River Media, 2004

[23] Inscape Consortium "Inscape Project Overview" http://www.inscapers.com/INSCAPE.Flyer.A4.project.overview.v1.2.pdf, , Stand: 2006

[24] Magerko, B., Laird, J.E., Assanie, M., Kerfoot, A., Stokes, D.: "AI Characters and Directors for Interacive Computer Games", 2004 http://www.eecs.umich.edu/~magerko/research/IAAI04MagerkoB.pdf

[25] Russel, S.J., Norvig, J.: "Artificial Intelligence: A Modern Approach", Prentice Hall, 2002

[26] Schwab, B.: "Advanced AI Engine Techniques", aus AI Game Engine Programming, Charles River Media, 2004

[27] "Ghost car" http://en.wikipedia.org/wiki/Ghost_car, Wikipedia, Stand: Juli 2006

[28] Skonnard, A., Gudgin, M.: "Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More", Addison-Wesley, 2001

[29] Kerievsky, J.: "Refactoring to Patterns", Addison-Wesley, 2005

[30] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995

[31] Freeman, E., Freeman, E.: "Head First Design Patterns", O'Reilly, 2004

[32] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: "Refactoring:

Improving the Design of Existing Code", Addison-Wesley, 1999

[33] "Altova XMLSpy Home Edition Benutzerhandbuch" http://www.altova.com/de/manual2006/XMLSpy/spyhome/, Altova, Stand: Oktober 2006

[34] "Altova UModel Benutzerhandbuch" http://origin.altova.com/de/manual2006/UModel/, Altova, Stand: Oktober 2006

[35] Alexandrescu, A.: "Modern C++ Design: Generic Programming and Design Patterns Applied", Addison-Wesley, 2001

[36] Smaragdakis, Y., Batory, D.: "Mixin-Based Programming C++", 2000 http://www.old.netobjectdays.org/pdf/00/papers/gcse/batorysmar.pdf

[37] Alexandrescu, A.: Prying Eyes: A Policy-Based Observer (I), C/C++ Users Journal, 2005 http://erdani.org/publications/cuj-2005-04.pdf

[38] Alexandrescu, A.: Prying Eyes: A Policy-Based Observer (II), C/C++ Users Journal, 2005 http://erdani.org/publications/cuj-2005-06.pdf

[39] Llopis, N.: "The Clock: Keeping Your Finger on the Pulse of the Game", aus Game Programming Gems 4, Charles River Media, 2004

[40] "Qt Reference Documentation" http://doc.trolltech.com/4.2/index.html, Trolltech, Stand: Oktober 2006

[41] "Xerces-C++ Documentation" http://xml.apache.org/xerces-c/apiDocs/index.html, Apache, Stand: Oktober 2006

[42] "Xalan-C++ API Documentation" http://xml.apache.org/xalan-c/apiDocs/index.html, Apache, Stand: Oktober 2006

[43] "Loki Modules" http://loki-lib.sourceforge.net/html/modules.html, Loki, Stand: Oktober 2006

[44] "Boost Libraries and Documentation" http://www.boost.org/libs/libraries.htm, Boost, Stand: Oktober 2006

[45] "CppUnit Documentation" http://cppunit.sourceforge.net/doc/lastest/index.html, CppUnit, Stand: Oktober 2006

Anhang 1: Verwendete Tools

- Programmierumgebung

  NarrationController und IcmlGraphMaster wurden unter Zuhilfenahme der Entwicklungsumgebungen Visual Studio 7.1/2003 und Visual Studio 8/2005 erstellt. Die Express-Version von Visual Studio 8 ist frei unter Folgendem Link herunterladbar:

  http://www.microsoft.com/germany/msdn/vstudio/express/download.mspx

- UML-Diagramme

  Die UML-Komponentendiagramme wurden mit Gentleware Poseidon UML erstellt (http://www.gentleware.com).

  Die UML-Klassendiagramme und -Sequenzdiagramme wurden mit Altova Umodel erstellt (http://www.altova.com).

- XML Schema

  Die XML Schemata und deren graphische Abbildungen wurden mit Altova XMLSpy erstellt (http://www.altova.com).

- Versionskontrolle

  Für die Versionierung und Verwaltung von Quellcode und Diplomarbeit wurde ein Subversion-Repository verwendet (http://subversion.tigris.org/).

- Erstellung der Diplomarbeit

  Die Diplomarbeit wurde mit OppenOffice 2.0 erstellt.

  (http://de.openoffice.org/)

  Einige der Abbildungen wurden mit Adobe (ehemals Macromedia) Fireworks MX 2004 bearbeitet (http://www.adobe.com/products/fireworks/).

Anhang 2: CD

Die Ergebnisse der Diplomarbeit wurden auch auf eine CD gebrannt. Der Inhalt setzt sich wie folgt zusammen:

- Die Diplomarbeit in elektronischer Form
- API-Dokumentation der NarrationController-Komponente
- API-Dokumentation der IcmlGraphMaster-Komponente
- XML Schema-Dokumentation der ursprünglichen ICML-Version
- XML Schema-Dokumentation der nach der Diplomarbeit vorliegenden ICML-Version
- UnitTest und DemonstrationApp in ausführbarer Form und als Quellcode

Anhang 3: UML-Klassendiagramm des NarrationControllers

Zum besseren Verständnis des Zusammenspiels und der Abhängigkeiten der in Kapitel 6 einzeln vorgestellte Bestandteile der NarrationController-Komponente befindet sich auf der Folgenden Seite ein UML-Klassendiagramm mit allen wichtigen Klassen und Schnittstellen, die im NarrationController verwendet werden.