Brandenburgische Technische Universität Cottbus
Institut für Informatik
Lehrstuhl Software-Systemtechnik

Masterarbeit

Analyse und Restrukturierung einer Anwendungsarchitektur

1. Gutachter : **Prof. Dr.rer.nat. Claus Lewerentz**
2. Gutachter : **Prof Dr.-Ing. habil. Ingo Schmitt**
Betreuer : **Prof. Dr.rer.nat. Claus Lewerentz**

vorgelegt von: Steffen Schulze Matrikel Nr. 2105182

Dokumentenversion: 10.08.2009
Abgabetermin: 12.08.2009

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst habe. Weiterhin versichere ich, dass ich keinen anderen als die angegebenen Quellen benutzt beziehungsweise die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Cottbus, den 12. August 2009

____

Steffen Schulze

Inhaltsverzeichnis

1 Einleitung....5
  1.1 Motivation....5
  1.2 Zielsetzung....5
  1.3 Gliederung der Arbeit....6
  1.4 Planungsmodell für grobgranulare Strukturebene....7
  1.5 Das zu betrachtende Projekt....8
2 Definitionen und Begriffsklärungen....10
  2.1 Architektur....10
  2.2 Alte-Soll-Architektur....10
  2.3 Ist-Architektur....10
  2.4 Ziel-Architektur....10
  2.5 Neue-Soll-Architektur....10
  2.6 Komponenten....10
  2.7 Konnektor....11
  2.8 System....11
  2.9 Untersystem/Subsystem....11
  2.10 Architekturstil....11
  2.11 Architekturregeln....13
  2.12 Implementierungsregeln....13
  2.13 Implementierung....13
  2.14 Ausprägung....13
  2.15 queryStatus-Aufruf....14
3 Grobgranulare Strukturebene....15
  3.1 Alte Soll-Architektur....15
    3.1.1 Übersicht....15
    3.1.2 Architekturstil und Architekturregeln....16
      3.1.2.1 Schichten....16
      3.1.2.2 Klient-Server....16
      3.1.2.3 Ereignissteuerung....17
    3.1.3 Zusammenfassung....17
  3.2 Implementierung....17
    3.2.1 Übersicht....17
    3.2.2 Implementierungsregeln und Ausprägung der Architekturstile....17
      3.2.2.1 Schichten....17
      3.2.2.2 Klient-Server....19
      3.2.2.3 Ereignisse....19
    3.2.3 Zusammenfassung....20
  3.3 Ist-Architektur....20
    3.3.1 Übersicht....20
    3.3.2 Architekturstil und Architekturregeln....20
      3.3.2.1 Schichten....20
      3.3.2.2 Klient-Server....21
      3.3.2.3 Ereignisse....22
    3.3.3 Zusammenfassung....22
  3.4 Ziel-Architektur....22
    3.4.1 Zielsetzung....22
    3.4.2 Architekturstil und Architekturregeln....23

3.4.2.1 Schichten....23
3.4.2.2 Klient-Server....27
3.4.2.3 Ereignisse....29
3.5 Neue Soll-Architektur....30
3.5.1 Nutzen gegen Aufwand....30
3.5.2 Architekturstil und Architekturregeln....30
3.5.2.1 Schichten....30
3.5.2.2 Klient-Server....30
3.5.2.3 Ereignisse....30
3.5.3 Implementierungsregeln....31
3.5.3.1 Schichten....31
3.5.3.2 Klient-Server....32
3.5.3.3 Ereignisse....33
3.6 Übergang Implementierung zur Neuen Implementierung....33
3.6.1 Schichten....33
3.6.1.1 Entkopplung von GUI und Logic-Schicht....33
3.6.1.2 Auftrennung in Daten und Logik....36
3.6.2 Klient-Server....40
3.6.3 Ereignisse....42
4 Feingranulare Strukturebene....45
4.1 Gliederung....45
4.2 QueryStatus Funktionalität....45
4.3 Parameter-Übergaben....49
4.4 Umstrukturierung von Datentypen....52
4.5 Const-Correctness durchsetzen....53
4.6 Statische Methoden....54
4.7 Plattformabhängiger Code....55
4.8 Harte Typenumwandlungen....58
4.9 Bitflags → Klassen....59
4.10 Singletons als globale Strukturen....62
4.11 Baumartige Vererbungshierarchien....64
5 Praxisbeispiel....69
6 Bewertung....72
6.1 Schlussfolgerungen....72
6.2 Relation von Architekturhierarchie zur Teamhierarchie....72
6.3 Erfahrungen in der Vermittlung....73
6.3.1 Regeln und Gruppendynamik....73
6.3.2 UML gegen Prototypen....74
7 Zusammenfassung und Ausblick....75

1 Einleitung

1.1 Motivation

Jede junge Industrie hat ein klicheebehaftetes Bild, mit dem sie in Verbindung gebracht wird. Bei der Computerspielindustrie hält sich recht hart das Vorurteil von kleinen Teams in Garagen ähnlichen Büros, welche sich ausschließlich von Pizza ernähren. Bis auf die italienische Köstlichkeit emanzipiert sich die Spielbranche jedoch zu einem unabhängigen Industriezweig. Spielserien, wie zum Beispiel das durch die Medien bekannt gewordene „Grand Theft Auto“, haben eine Jahrzehnte lange Entwicklungsphase hinter sich, in dennen sie inzwischen auf Budgets im dreistelligen Millionenbereich zurückgreifen können [WEB01]. Dabei wird viel Technologie von Drittanbietern eingekauft. Diese muss dann stimmig und flexibel in das eigene Gerüst eingearbeitet werden.

Auch das Spiel „Sacred 2 – Fallen Angel“, dessen Code der folgenden Betrachtung zu Grunde liegen wird, kann sich beim Umfang an ähnlichen Maßstäben messen. Nach 5 Jahren Entwicklung baut es auf circa 1 Million Code-Zeilen und auf etwa ein Dutzend Zusatztechnologien von Drittanbietern. Neben der Hauptplattform PC entwickelte man auch für Xbox360 und PS3. Die Codebasis wurde für diesen Teil der Serie komplett neu entwickelt und soll nun für weitere Fortsetzungen als Basis dienen.

Das Projekt „Sacred 2“ entstand iterativ inkrementell. Die Features durch das Spieldesign wurden erst während der Entwicklungsphase komplettiert. Neue Systeme wurden eingebaut, alte erweitert. Die Verantwortlichkeiten für bestimmte Programmteile wechselten nicht selten mit jeder neuen Aufgabenverteilung. Aufgrund des stetigen Zeitdrucks stand die Erfüllung einer Aufgabe über der architektonischen Sauberkeit. Das System verwilderte zusehends.

Ein gut geplantes, umgesetztes und dokumentiertes System ist bei Projekten solch einer Größe kein Zusatzziel, was man auf halber Strecke aus den Augen verlieren darf. Kleine „Workarounds“ und „Hacks“, die im Laufe der Umsetzung eingebaut werden, manifestieren sich auf Dauer zu zeitraubendem, unwartbarem Programmcode. Dieser kann nicht nur ärgerlich sein, sondern verursacht auch immense Kosten und verhindert nicht zuletzt nötige Anpassungen.

In der Entwicklung wurde auf automatisierte Laufzeittests verzichtet. Vielmehr waren manuelle Laufzeittest die einzige Instanz, um das Programm auf Fehlerfreiheit zu kontrollieren. Ebenso fanden regelmäßige Refactorings und Codereviews nicht statt.

1.2 Zielsetzung

Diese Arbeit wird begleitend zur allgemeinen Umstrukturierung der Code-Basis von „Sacred 2“ zu „Sacred 3“ geschrieben, im Laufe dessen werden wiederkehrende Architekturschwachstellen diagnostiziert und systematische Umstrukturierungen dieser erarbeitet werden. Ein klarer Architekturstil soll ebenso Ergebnis sein, wie Empfehlungen für dessen Implementierung. Die Struktur auf Klassenebene soll flexibler, klarer und wartungsfreundlicher gestaltet werden.

Dabei soll ein systematisches Vorgehen eingeführt werden, wie man iterativ die Architektur kontrollieren und überarbeiten kann.

In ersten Schritten werden Mängel analysiert und Gegenentwürfe aufgestellt. Diese werden bewertet und eine Umstrukturierung ausgearbeitet.

1.3 Gliederung der Arbeit

Kapitel 1 dient der Einführung in die Aufgabenstellung und Zielsetzung. Zudem wird das zu betrachtende Projekt Vorgestellt.

Im Kapitel 2 werden Definitionen der benötigten Begriffe zusammengestellt.

Das Kapitel 3 befasst sich mit der grobgranularen Strukturebene. Diese Betrachtung wird das Projekt aus einer architektonischen Sicht analysieren und draus Erkenntnisse ziehen.

Anschließend folgt eine feingranulare Betrachtung in Kapitel 4. Hier wird sich vor allem auf die Implementierungsebene konzentriert und dort auftretende Teilprobleme behandelt. Es wird im Gegensatz zum zweiten Kapitel eher der Programmcode untersucht, als die Architektur.

In Kapitel 5 wird ein praktisches Beispiel gegeben, was sich aus der Anwendung vorangegangener Erkenntnisse ergibt.

Kapitel 6 enthält eine zusammenfassende Bewertung der Erkenntnisse aus dieser Arbeit.

Der Anhang enthält Abbildungs-, Tabellen- und Literaturverzeichnis und einen Glossar.

1.4 Planungsmodell für grobgranulare Strukturebene

Für die Architekturbetrachtung auf grobgranularer Strukturebene wird ein Planungsmodel zu Grunde gelegt, in der eine Iterationsphase sich wie folgt definiert:

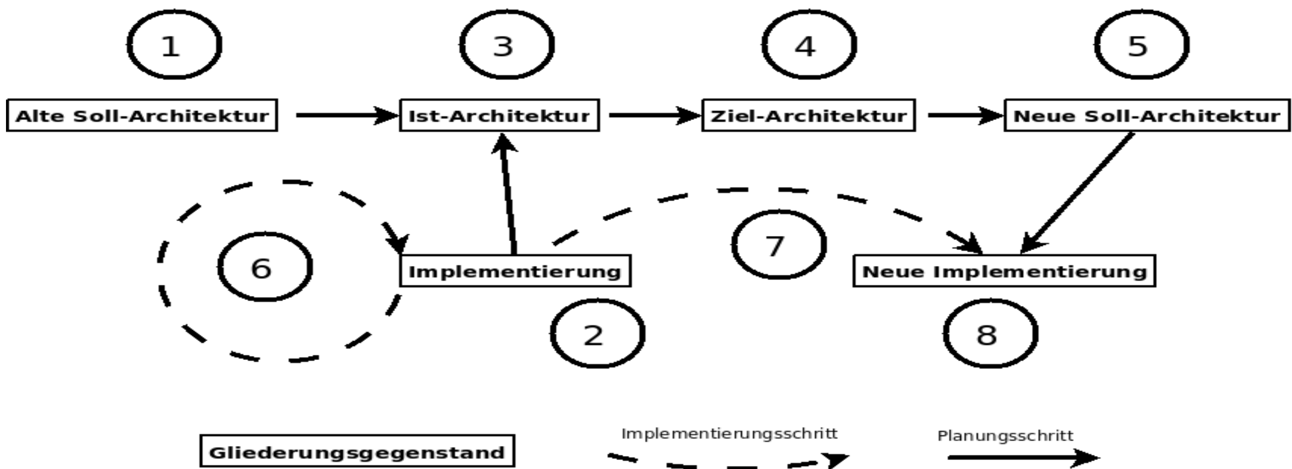

Diagramm 1: Diagramm des genutzten Planungsmodels

1. Aus den möglichen Quellen wird die **Alte Soll-Architektur** rekonstruiert. Dies ist die ursprüngliche Planung auf der die zu betrachtende Implementierung aufbaut.
2. Die **Implementierung** ist der vorhandene aktuelle Projektstand, welcher der Betrachtung zu Grunde liegt
3. Aus der vorliegenden Implementierung wird die **Ist-Architektur** zurück konstruiert. Im optimalen, aber eher unwahrscheinlicheren Fall entspricht diese der Alten-Soll-Architektur. Anhand der Ist-Architektur wird die Analyse vorgenommen und Defizite identifiziert.
4. Für diese Defizite werden Lösungen entworfen und in einer **Ziel-Architektur** festgehalten. Diese beschränkt sich auf rein architektonische Betrachtungen. Implementierungsregeln werden hier nicht definiert.
5. Die Ziel-Architektur stellt eine theoretische Wunschlösung dar und muss insbesondere anhand von Kosten-Nutzen-Abschätzungen auf einen umsetzbaren Umfang gebracht werden. Dieses Ergebnis wird im Folgenden als **Neue Soll-Architektur** bezeichnet, welche den Arbeitsauftrag für die Umstrukturierung darstellt.
6. Bevor man die Implementierung anhand der Neuen Soll-Architektur überführt, sollte die Implementierung weitestgehend aufgeräumt werden ohne die Ist-Architektur zu verändern. Dies umfasst zum Beispiel das Entfernen von handwerklichen Fehlern, Verschlanken der Implementierung (kleine Refactors) oder auch den Austausch von fremder Middleware.
Diese Aktionen können problemlos parallel zu den Punkten 4 und 5 vollzogen werden.
7. Sind Punkte 5 und 6 abgeschlossen kann die Implementierung in die **Neue Implementierung** überführt werden. Im vorliegenden Dokument werden diese so weit wie möglich als Schritt-für-Schritt-Übergänge dokumentiert, damit eine einfache und praxisnahe Umformulierung in Arbeitstasks möglich wäre.

8. Wurde die Neue Soll-Architektur in die Neue Implementierung überführt, kann eine Bewertung des Ergebnisses vorgenommen werden.

1.5 Das zu betrachtende Projekt

Das Projekt „Sacred 2“ ist in der Programmiersprache C++ geschrieben. Als Entwicklungsumgebung fungiert Visual Studio 8 unter Microsoft Windows XP und Vista. Das Projekt wurde nach 5 jähriger Entwicklungsphase im Oktober 2008 veröffentlicht und bis April 2009 weiterentwickelt und gepflegt.

Bei dem Spiel handelt es sich um ein sogenanntes Action-Rollenspiel. Der Spieler übernimmt die Kontrolle einer Spielfigur, dessen Fähigkeiten und Ausrüstung er erweitern und verändern kann. Nötiges Geld und Erfahrungspunkte erhält man durch das erfüllen von Aufgaben oder durch besiegen von Gegnern. Neben der erzählten Geschichte liegt der Fokus auf schnelle aktionsorientierte Spielweise, die dem Spieler stets Resonanz auf sein Handeln gibt.

Die Entwicklung findet an 2 Standorten statt. Die Studio 2 Software GmbH in Aachen ist für die komplette PC Version zuständig. Dort befinden sich zudem Designer, Grafiker, Producer und eine interne Qualitätssicherung. Der Ascaron Hauptsitz in Gütersloh konzentriert sich auf die Konsolenanpassung, was hauptsächlich den Renderer, die GUI und Anpassungen an die technischen Bedürfnisse der Konsolen umfasst.

Der Standort Aachen beschäftigte zum Projekthöhepunkt mehr als ein Dutzend Programmierer. Von der studentischen Hilfskraft und dem Quereinsteiger bis zum diplomierten Informatiker ist dabei jede Qualifikationsstufe vertreten.

Die Programmierer kann man in folgende Aufgabenbereiche kategorisieren:

- Logic-Programmierer
  - AI
    Künstliche Intelligenz der Figuren im Spiel. NPCs haben zum Beispiel einen festen Tagesablauf oder Gegner ein Gruppenverhalten um auf den Held zu reagieren.
  - Spells
    Zaubersprüche manipulieren die Spielwelt logisch und grafisch. Sie machen bei „Sacred 2“ einen großen Teil der möglichen Spielerinteraktionen aus.
  - Equip
    Spielfiguren und Spielgegenstände. Die Menge aller möglichen Interaktionen zwischen diesen Entitäten umfasst ein weites Spektrum. Anlegen, Handeln, Veredeln sollen hier nur beispielhaft aufgezählt sein.
- Hardware-Programmierer
  - Sound
    Auslösen des akustischen Anwendungs-Feedback
  - Graphics
    Optische Vermittlung der Spielwelt und die dafür nötigen Effekte
  - Performance und Konvertierung
    Optimieren des Codes auf Laufzeitgeschwindigkeit. Anpassen des Codes auf gewisse Plattformen (PC, Xbox360, PS3)

- Anwendungsprogrammierer
  - GUI
    Ausgabe von Spieldaten und Eingabe von Benutzerinteraktionen
- Tool-Programmierer
  - Editoren für die Spieldaten (Spielwelt, Spielgeschichte)
  - Verwaltung von Assets (Grafikmodelle, Texturen, Soundsamples)
  - Intranet (Wiki, Build-Pipeline)
- Netzwerk-Programmierer
  Synchronizität und Konsistenz von Daten und Effekten auf allen Klienten

Die architektonischen Entscheidungen kontrollierte der „Technical Director“. Dieser war allerdings durch den Zeitdruck stark als Programmierer eingebunden und konnte seine kontrollierenden und planerischen Eigenschaften daher nur bedingt wahrnehmen.

2 Definitionen und Begriffsklärungen

2.1 Architektur

Folgenden Definitionen der Architektur möchte ich mich im vorliegenden Dokument anschließen.

Definition Software-Architektur eines Systems (Struktur) [SoAr]

> *Die Software-Architektur eines Systems beschreibt dessen Software-Struktur respektive dessen Strukturen, dessen Software-Bausteine sowie deren sichtbaren Eigenschaften und Beziehungen zueinander.*

Definition Software-Architektur als Disziplin [SoAr]

> *Software-Architektur als Disziplin befasst sich mit den architektonischen Tätigkeiten und den hiermit verbundenen Entscheidungen zur Konzeption und Realisierung einer Software-Architektur(Struktur).*

2.2 Alte-Soll-Architektur

Dabei handelt es sich um die geplante Architektur für die zu analysierende Implementierung. Legt man das hier genutzte Planungsmodel zu Grunde, so ist sie die Neue-Soll-Architektur des vorangegangenen Iterationsschrittes.

2.3 Ist-Architektur

Damit wird die Architektur der zu analysierenden Implementierung bezeichnet. Diese Architektur ergibt sich direkt aus der Analyse der Implementierung. Die Differenz aus Alter-Soll-Architektur und Ist-Architektur zeigen die Teile der Ausprägung, welche nicht planmäßig umgesetzt wurden.

2.4 Ziel-Architektur

Die Ziel-Architektur entspricht dem Entwurfsschritt. In ihr wird die Wunschlösung für die gestellten Anforderungen konstruiert.

2.5 Neue-Soll-Architektur

Die Wünsche der Ziel-Architektur und die Vorgaben der Ist-Architektur müssen mit der Neuen-Soll-Architektur zusammen geführt werden. Hier fließen vor allem der Aspekt des benötigten Aufwandes für die Umstrukturierungen ein. Ist dieser zu hoch muss ein Kompromiss gefunden werden.

2.6 Komponenten

Die Definitionen der Komponente in diese Arbeit beruht auf die Aussagen von [Szyperski]:

> *Eine Komponente ist eine Kompositionseinheit mit vertraglich spezifizierten Schnittstellen, die nur explizite Abhängigkeiten zu ihrem Kontext hat. Eine Software-Komponente kann unabhängig eingesetzt werden und kann durch Dritte komponiert werden.*

Eine Komponente ist die Abstraktion eines Systems anhand all seiner öffentlichen Schnittstellen. Damit ist im Folgenden Komponente der Überbegriff für alle Architekturentitäten, ausgenommen den Konnektoren.

2.7 Konnektor

Regeln die Koordination und Kommunikation zwischen Komponenten. Dies können Funktionsaufrufe oder gemeinsame Daten sein. Es handelt sich um explizite Kopplungen.

2.8 System

Beim Systembegriff schließe ich mich der Definition aus [SoAr] an.

> *A system is an entity which maintains its existence through the mutual interaction of its parts.*

Ein System ist das Zusammenspiel von Komponenten und Konnektoren. Ein System kann selber als Komponente betrachtet werden, sofern es deren Anforderungen erfüllt.

2.9 Untersystem/Subsystem

An sich sind Untersysteme nichts anderes, als die oben definierten Systeme. Allerdings besitzen sie im Betrachtungskontext eine Zugehörigkeit zu einer übergeordneten Komponente/System.

2.10 Architekturstil

Der Begriff Stil bezeichnet eine charakteristisch ausgeprägte Art der Ausführung. Bei Architekturen sind dies Lösungsansätze für wiederkehrende Aufgabenstellungen. In [SoAr] findet man dazu folgende Definition:

> *Ein Architekturstil dokumentiert einen erprobten und erfolgreichen Weg, eine Architektur zu strukturieren. Jeder Stil besitzt bestimmte Charakteristika und dient als Vorlage für den Entwurf der eigentlichen Architektur.*

In diesem Kontext werden die oben definierten Begriffe Komponenten, (Unter-)System und Konnektor genutzt.

Da es in Informatiksystemen um den Fluss von Informationen und Kontrolle geht, können wichtige Architekturstile anhand dieser klassifiziert werden. Im Folgenden werden dazu beispielhafte Architekturstile erläutert, da diese in der kommenden Betrachtung von Interesse sein werden und um den Begriff Architekturstil zu manifestieren.

Schichten

<u>Erläuterung:</u>

- Untersysteme einer Schicht besitzen gemeinsame Merkmale und Aufgaben.
- Schichten können nur auf darunter liegende Schichten zugreifen.
- Ein striktes Schichtenmodell erlaubt nur Zugriffe auf die direkt darunter liegende Schicht.

<u>Beispiel:</u>

- TCP-Stack

Kontrollfluss:

Der Kontrollfluss wird über die Stapelung der Schichten bestimmt und beruht auf Call-and-Return Konnektoren.

Objektnetze

Erläuterung:

- Komponenten (Objekte) verknüpfen sich frei mit öffentlichen Schnittstellen anderer Komponenten.

Beispiele:

- Ein klassisches objektorientiertes Programm. Klassen und Namensräume bilden die Komponenten.

Kontrollfluss:

Der Kontrollfluss wird über die Nutzungsabhängigkeiten der Objekte bestimmt. Auch hier handelt es sich um Call-and-Return Konnektoren.

Datenflussnetze

Mittels eines Datenflusses (Konnektor) verknüpfte Verarbeitungsschritte/Filter (Komponente). Diese bilden einen gerichteten Bearbeitungsablauf, welche die Daten durchwandern.

Beispiel:

- Renderpipeline
- Compiler
- Unix-Pipes

Kontrollfluss:

Der Kontrollfluss wird über Verknüpfungen der Pipes bestimmt. Es gibt nur den Call zwischen Komponenten

Ereignissteuerung (Unabhängige Komponenten)

Erläuterung:

Komponenten registrieren sich an zentraler Stelle um bei bestimmten Ereignissen (Konnektoren) benachrichtigt zu werden.

Ereignisse können von jeder Komponente erzeugt werden.

Beispiel:

- Signal-Slot-Systeme

Kontrollfluss:

Je nachdem, welche Strukturen sich an ein Ereignis binden, verzweigt sich der Kontrollfluss. Allerdings ist kein explizites Return vorgesehen.

Interpreter oder Regelsysteme (Virtuelle Maschinen)

Erläuterung:

Konnektoren im System werden zur Laufzeit anhand von Eingabedaten interpretiert.

Beispiel:

- Scriptsprachen

Kontrollfluss:

Der Kontrollfluss wird über den zu interpretierenden Code/Werte bestimmt.

Ablagebasierte Struktur (Datenzentriert)

Erläuterung:

Komponenten betrachten eine gemeinsam genutzte Datenstruktur und reagieren auf Problemmuster, welche sie unabhängig bearbeiten.

Beispiel:

- Verteilte Systeme, welche über gemeinsamen Speicher kommunizieren

Kontrollfluss:

Komponenten werden nicht über einen Kontrollfluss gesteuert, sondern über gemeinsam genutzte Daten.

2.11 Architekturregeln

Stellt der Architekturstil das Ziel dar, so definieren die Architekturregeln den Weg. Sie definieren Orientierungspunkte und Grenzen bei der Umsetzung. Diese Regeln sind sprach- und werkzeugunabhängig.

2.12 Implementierungsregeln

Sie setzen die Umsetzung von Architekturstrukturen sprachspezifisch fest. Hier kommen Begriffe wie Klassen, Instanzvariablen oder auch Vererbung ins Spiel.

2.13 Implementierung

Ist die Umsetzung von festgelegter Architektur in einem System unter Berücksichtigung von Architektur- und Implementierungsregeln. Es ist zu unterscheiden zwischen dem hier gemeinten Vorgang der Implementierung und dem Betrachtungsgegenstand des Quellcodes, auch als Implementierung bezeichnet.

2.14 Ausprägung

Während die Implementierung die willentliche Umsetzung der Architektur darstellt, handelt es sich bei der Ausprägung um die Rückschlüsse aus dem Code auf die Architektur. Ist die Implementierung des Wunschresultat, so ist die Ausprägung das tatsächliche Ergebnis.

2.15 queryStatus-Aufruf

Dabei handelt es sich um eine projektspezifische Ausprägung. Dieser Methodenaufruf in Klassen ist mit virtuellen Funktionen zu vergleichen. Allerdings nutzt dieser untypisierte Parameter eine Kommando-ID und bringt zahlreiche Fallstricke mit sich. Die Kopplungen über diesen Aufruf gehen über die normale Vererbung hinaus, da queryStatus auch an Member oder über Events delegiert wird. An den Kopplungspunkten werden Kommando-ID und Parameter ungeprüft übergeben.

Genauer erklärt und behandelt wird queryStatus in den feingranularen Betrachtungen. Trotzdem ist die Erwähnung vor der grobgranularen Betrachtung von Bedeutung, da dieser Mechanismus das Hauptinstrument für Architekturbrüche über Schnittstellen hinweg ist und mehrfach darauf verwiesen wird.

3 Grobgranulare Strukturebene

Im folgenden Kapitel wird anhand des beschriebenen Planungsmodells das Projekt auf einer hohen Abstraktionsebene betrachtet. Dabei wird sich auf die folgenden drei Punkte konzentriert.

- Schichten
  - Umsetzung des Schichtenmodels
- Klient-Server
  - Trennung von Klient und Serverfunktionalitäten im Programmcode
- Ereignisse
  - Implementierung des ereignisorientierten Systems im Projekt

Es gäbe viele weitere untersuchenswerte Punkte, welche jedoch den Umfang dieser Arbeit sprengen würden. Trotzdem werden hier nur einige der Vollständigkeit wegen aufgezählt.

- dynamischer Spielinhalt
  - Verbindung von Spielinhalt mit der Programmbasis über Interpreter (Lua)
- Renderpipeline
  - Wie in Datenflussnetzen aus Logik- und GFX-Daten Pixel auf dem Bildschirm werden.

Es werden die Schritte des Planungsmodells kapitelweise abgehandelt.

3.1 Alte Soll-Architektur

3.1.1 Übersicht

Die Programmarchitektur unterteilt sich in folgende Schichten:

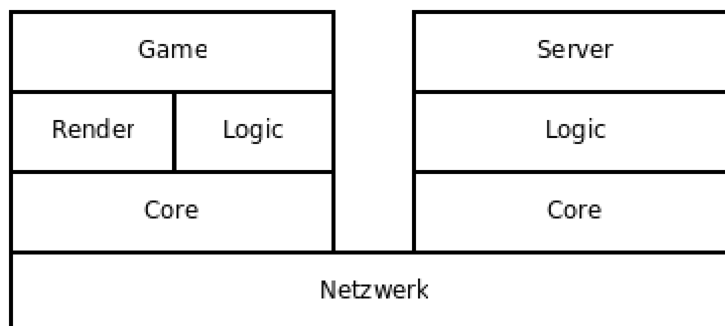

Diagramm 2: Schichtenmodell der Alten Soll-Architektur

Game

- vermittelt zwischen Nutzer und Programm
- Grafische Oberfläche und Eingabeverarbeitung

Server

- im Vergleich zu **Game** stark vereinfachte Anwendungsebene
- ermöglicht Betrieb als dedizierten Server

Render

- stellt ein dreidimensionales Abbild der aktuellen Spielumgebung dar

Logic

- enthält jegliche Funktionalität für die Spielverwaltung
- beherbergt die Dateninstanzen der Spielwelt

Core

- technische Funktionalitäten zur Kommunikation
- Hardware-Abstraktion

Netzwerk

- Kommunikation zwischen den Programminstanzen (Implementierung von Drittanbietern; wird von **Core**-Schicht angesprochen)

3.1.2 Architekturstil und Architekturregeln

Hier werden soweit wie möglich die noch nachweisbaren geplanten Architekturstile und -regeln beschrieben. Implementierungsregeln sind nicht mehr nachweisbar, werden daher erst bei der Implementierung besprochen.

In diesem und den folgenden Umstrukturierungsschritten auf der grobgranularen Betrachtungsebene wird dies exemplarisch an 3 Punkten durchgeführt. Diese wurden ausgewählt, da sie verhältnismäßig einfach zu erfassen sind, dabei aber einen breiten Überblick über das Projekt verschaffen.

3.1.2.1 Schichten

Architekturregeln:

- Komponenten kennen keine Komponenten, welche in der Schichtenhierarchie über ihnen stehen.

Erläuterung:

Es war kein striktes Schichtenmodell vorgesehen. Das heißt, eine Schicht darf auf alle Schichten zugreifen, die ihr in der Schichtenhierarchie, auch indirekt, untergeordnet sind.

3.1.2.2 Klient-Server

Architekturregeln:

- Entscheidungen der Spiellogik dürfen nur vom Server getroffen werden.

Erläuterung:

Aktionen werden zumeist auf dem Klient vom Spieler ausgelöst. Benötigen diese Aktionen eine Entscheidung der Spiellogik, so darf diese nur vom Server getroffen werden, welcher das Ergebnis an den Klient weiterleitet.

Zur synchronisierten Kommunikation zwischen den Klienten und dem Server wurde eine Ereignissteuerung vorgesehen.

3.1.2.3 Ereignissteuerung

Architekturregeln:

- Wird eine Aktion ausgelöst, so geschieht dies über ein Ereignis.

Erläuterung:

Vorgänge, welche nicht allein von der lokalen Instanz bearbeitet oder entkoppelt vom aktuellen Kontrollfluss ausgeführt werden sollen, werden über ein Ereignis ausgelöst.

3.1.3 Zusammenfassung

Die Alte Soll-Architektur wurde praktisch kaum geplant. In der Projektdokumentation fanden sich nur vereinzelte Klassendiagramme. Die oben genannten Punkte ergeben sich aus mündlich gesetzten Projektregeln.

3.2 Implementierung

3.2.1 Übersicht

Der Quelltext ist in C++ geschrieben. Einschränkend muss man jedoch feststellen, dass bei der Umsetzung eher in C gedacht wurde. Deutlichste Beispiele sind hier memset, mit denen Klassenobjekte initialisiert wurden, oder memcpy, zum Kopieren solcher. Zahlreiche void-Zeiger und daraus resultierende Typenumwandlungen sind ein weiteres Indiz, welches in der feingranularen Betrachtung genauer unter die Lupe genommen wird.

Um die Kompilierungszeit zu verringern wurde eine Technik eingeführt, welche sich Unity-Build [WEB02] nennt. Dabei werden cpp-Dateien zu größeren Paketen zusammengeführt und in einem Schritt kompiliert. Eingebundene Header-Dateien einer cpp-Datei werden dabei global im Paket sichtbar.

3.2.2 Implementierungsregeln und Ausprägung der Architekturstile

Implementierungsregeln haben sich erst während der Implementierung herauskristallisiert. Aus diesem Grunde werden sie hier ungewöhnlicherweise zusammen mit der Ausprägung der Architekturstile behandelt.

3.2.2.1 Schichten

Damit Schichten als Komponenten geschützt werden, werden sie in dynamisch gebundene Bibliotheken gegliedert. Öffentlich zugänglich soll nur sein, was nicht explizit exportiert wird. Bis auf die Verbindung GUI<->Logik wurden jedoch für keine der Schichten Schnittstellen definiert. Das führte dazu, das sich speziell in der Logikschicht fast jedes Subsystem, ob nötig oder unnötig, exportiert hat und somit global sichtbar ist.

Die hierarchische Struktur der Schichten wird „gesichert“, indem man das Benutzen von Header-Dateien übergeordneter Schichten verbietet. Dies konnte nicht eingehalten werden. Das Einbinden der Header-Dateien fand zudem meist in der eigenen Header-Datei statt, was zu einer dichten Vernetzung jener führt. Somit werden die unerlaubt bekannten Strukturen annähernd global sichtbar und dementsprechend genutzt. (Siehe auch „Importchaos“ und „Importlüge“ in [CoQualMan])

Anhand von SotoArc kann man sehr gut Architekturverletzungen erkennen. Zum Beispiel, wenn eine Schicht auf ihre übergeordnete zugreift. Schwer wird es allerdings,

wenn es Mechanismen im Code gibt, die extra dafür eingeführt wurden, um Architekturgrenzen zu überwinden. Ist eine derartige Möglichkeit einmal vorhanden, wird sie auch genutzt. Folgende Tabellen beschreiben anhand von sogenannten queryStatus-Aufrufen schichtenübergreifende Zugriffe. Aufgezeigt werden hier 4 Richtungen, die das angesetzte Schichtenmodell gar nicht erlauben dürfte. Bei den Beispielen handelt es sich um Aktionen, bei denen der Aufrufer mit einem Wissen einen Aufruf startet, das er eigentlich gar nicht haben kann.

Gesamtaufrufe	28
Varianten	11

Tabelle 1: Render -> Logik

Der Renderer ruft vor allem Daten aus der Logik ab. Dies ist unter anderem ein Resultat der fehlenden Datenschicht. Viele für das Rendern nötige Daten befinden sich in der Logik.

Gesamtaufrufe	150
Varianten	65

Tabelle 2: Logik -> Render

Aufrufe von der Logik zum Renderer zeigen sehr gut das Ausmaß an Konsequenzen, welches architekturverletzende Funktionalitäten haben können. Durch die Möglichkeit, den Renderer aus der Logik zu kontrollieren, ist ein Großteil dessen Verantwortung in die Logik gewandert. Renderobjekte werden von der Logik erzeugt und zahlreiche visuelle Effekte von dort gesteuert. Wird zum Beispiel ein Zauberspruch im Spiel aktiviert, kümmert sich die Logik neben anderen Dingen auch um die Aktivierung und das Timing der zugehörigen Effekte und Animationen.

Gesamtaufrufe	11
Varianten	11

Tabelle 3: Core -> Render

Gesamtaufrufe	35
Varianten	18

Tabelle 4: Core -> Logik

Grenzen werden dadurch immer mehr verwässert und die Vergehen manifestieren sich im System. Wird eine Architekturverletzung unglücklich mit einer anderen verbunden, kann das zu extremen Resultaten führen. So war es mit der Zeit ungewollt möglich, von einem Klienten aus Grafikeffekte auf allen anderen Klienten, welche mit dem Server verbunden sind, zu manipulieren.

Auch die Portierbarkeit hat gelitten. Auf den Konsolen wurde ein komplett anderes System für die grafische Oberfläche (folgend GUI) eingesetzt. Im Code für die PC-Version, Basis für die Konsolenportierungen, sind viele Entscheidungen, die eigentlich nur die Logik betreffen, in die GUI gewandert. Da sie die Möglichkeit hat, sehr weitreichend auf Daten der Logik zuzugreifen, hat sie diese auch anhand eigener Regeln für ihre Entscheidungsfindung genutzt, anstatt selber die Logik zu fragen. So wurden auf der Konsole Ereignisse nicht mehr ausgelöst oder es fehlten Sicherheitsbarrieren, die für PC in die GUI gewandert waren.

3.2.2.2 Klient-Server

Bis auf die Anwendungsebene mit den Schichten Game und Server, gab es in der Architektur keine weitere Trennung von Klient und Server. Diese Trennung fehlt auf grobgranularer Ebene ansonsten komplett. Das heißt, ein Server hat die gleichen Logik- und Core-Komponenten wie ein Klient.

Eine Trennung erfolgt erst auf Codeebene durch Abfrage zur Laufzeit, ob sich der Kontrollfluss gerade auf dem Server oder Klient befindet. Vergessene oder verdrehte Abfragen führen dementsprechend zu Verzögerungen in der Entwicklung. Es ist davon auszugehen, dass dieser Umstand noch zahlreiche Sicherheitslücken und Fehler im Code parat hält.

3.2.2.3 Ereignisse

Ereignisorientierte Systeme kennzeichnen sich durch die Unabhängigkeit von Auslöser und Reaktion.

In der vorliegenden Implementierung sind Ereignisse Strukturen, welche beim Auslösen mit Daten befüllt werden, und dann an die zu empfangenden Programminstanzen (zum Beispiel: „Server“, „Alle“, „Alle außer mir“) verschickt werden.

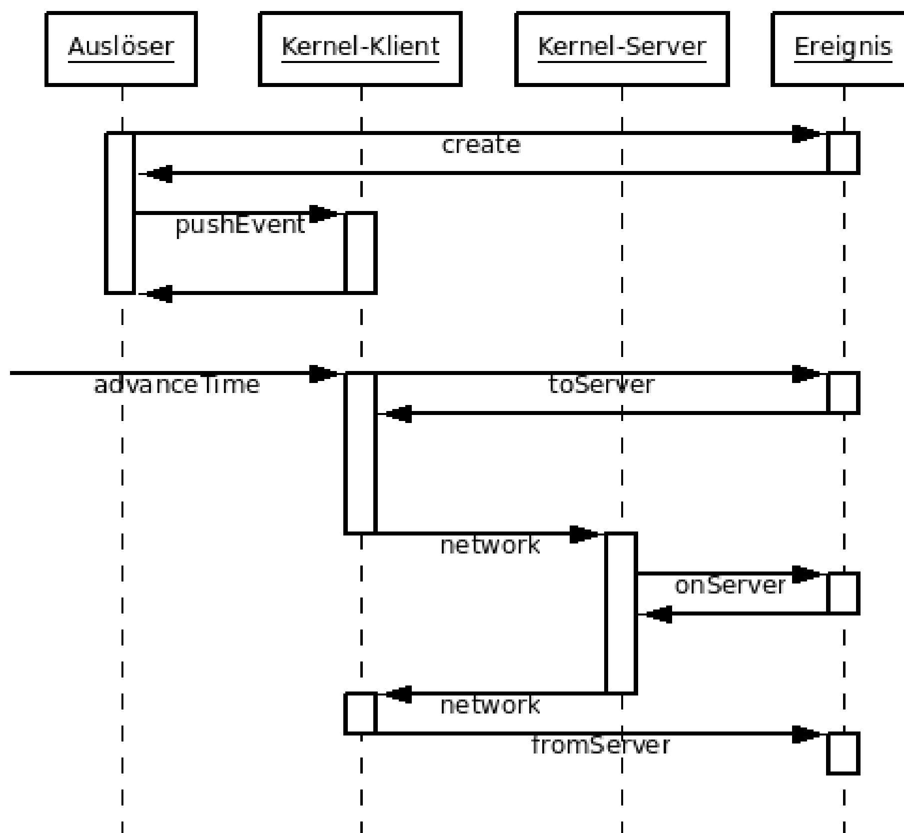

Diagramm 3: Sequenz einer Ereignisabarbeitung

Obiges Diagramm illustriert eine normale Ereignissequenz.

- Nachdem der Auslöser das Ereignis erzeugt hat, wird es dem Kernel übergeben, welcher es zwischenspeichert.
- Wird der Kernel vom System zur Abarbeitung der gepufferten Daten über advanceTime aufgerufen, findet er die Ereignisstruktur vor um auf ihr die

Methode toServer aufzurufen. Danach schickt er die Struktur über das Netzwerk an den Server.

- Dort wird der Kernel des Servers das Ereignis empfangen und dessen onServer aufrufen. Ist dies erfolgt, schickt er die Ereignisstruktur an alle vom Auslöser adressierten Empfänger.
- Deren Kernel erkennen den Server als Sender und rufen darauf fromServer auf den Ereignissen auf.

Für die grobgranulare Betrachtung werden dabei einige wichtige Dinge deutlich:

- toServer, onServer, fromServer hätte vermutlich als zentraler Punkt zur Trennung von Klient- und Server-Kontrollfluss ausgereicht. (siehe vorangegangenes Kapitel)
- Das Ereignis entscheidet darüber, welche Reaktionen getroffen werden. Damit muss es auch alle Komponenten kennen, welche es informiert. Verbreiteter ist der Ansatz, dass Komponenten auf das Eintreten eines Ereignisses warten; also die Komponenten die nötigen Ereignisse kennen müssen und nicht andersrum.
- Ereignisse mit der vorliegenden Ausprägung können die genutzten Komponenten entkoppeln, führen aber zu einer unnötigen Konzentration von Kompetenzen in einer Struktur, die nur zur Kommunikation gedacht ist.

3.2.3 Zusammenfassung

Der Implementierung ist anzumerken, dass sie auf keiner übergeordneten Planung aufbaut. Schnittstellen zwischen Komponenten entstanden meist Bedarf und ohne bei Verantwortlichkeit. Häufiger wurden diese Schnittstellen komplett umgangen und direkt auf den Daten gearbeitet.

3.3 Ist-Architektur

3.3.1 Übersicht

Aus der Implementierung und den Ausprägungen wird die Ist-Architektur konstruiert. Sie stellt die architektonische Sicht auf den gegebenen Ausgangszustand dar und dient als Basis für die Analyse um daraus dann einen Entwurf, die Ziel-Architektur, zu erstellen.

3.3.2 Architekturstil und Architekturregeln

3.3.2.1 Schichten

Der in der Alten Soll-Architektur geregelte Architekturstil der Schichten wurde nicht eingehalten. Folgendes Diagramm stellt die Kopplungen der Schichten in der Ist-Architektur dar.

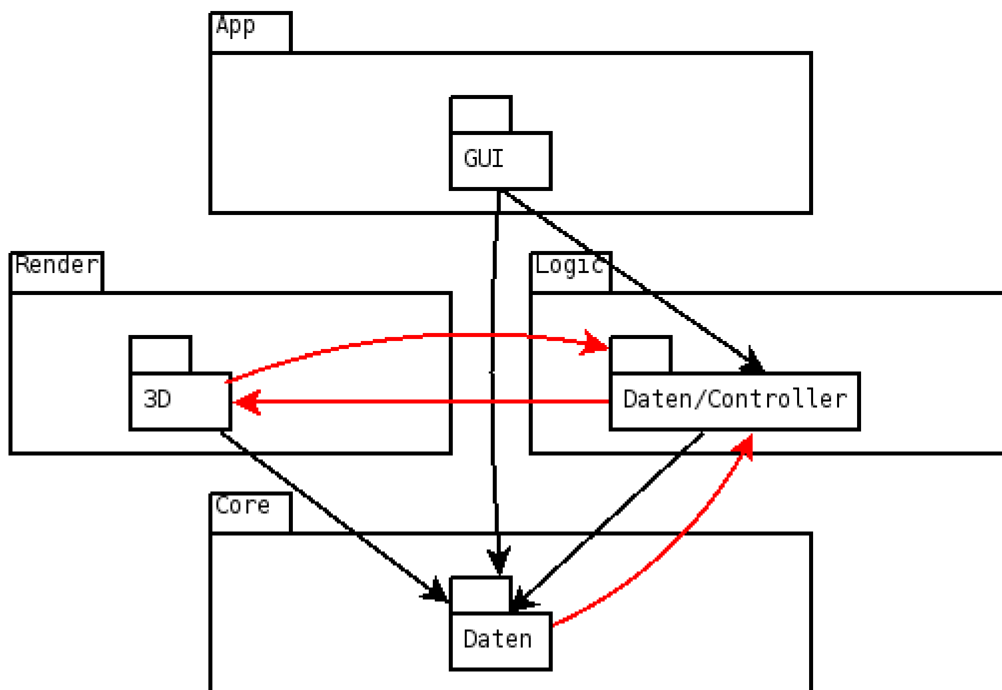

Diagramm 4: Kopplungen im Schichtenmodell in der Ist-Architektur

Kanten markieren dabei Zugriffe auf eine andere Schicht. Sind diese schwarz, so handelt es sich um erlaubte. Rote dagegen weisen auf problematische Nutzungen hin.

- GUI nutzt Daten aus Core und Logik für eigene Logikentscheidungen
- Render nutzt für die Darstellungen Daten aus Core und Logic
- Logic kontrolliert direkt Vorgänge in Render
- Daten aus Core haben Kopplungen mit Daten aus Logic
- Daten und kontrollierende Logik sind eng verwoben

Die unerlaubten Kopplungen lassen sich auf 2 Hauptursachen zurückführen. Erstere sind die mangelnden Implementierungsregeln. Diese ermöglichen das Nutzen von Strukturen, welche laut Architekturregeln gar nicht sichtbar sein dürften. Darauf lassen sich vor allem die Kopplungen von der Core- zur Logic-schicht zurückführen.

Die enge Verbindung der eigentlich getrennten Render- und Logic-schicht begründet sich durch den queryStatus-Mechanismus. Dieser wird in der feingranularen Betrachtung näher erläutert. Die vereinfachte Beschreibung dieses Mechanismus ist dem Glossar zu entnehmen.

3.3.2.2 Klient-Server

Aus den Schichtendiagrammen der vorangegangenen Kapitel kann man entnehmen, dass eine Serverinstanz keine Render-Schicht besitzt. Die bereits erwähnten Aufrufe aus den anderen Schichten in diese Schicht hinein finden trotzdem statt. Nur laufen sie ins Leere. Dies ist solange vertretbar, bis die aufrufenden Komponenten Rückgabewerte oder andere Garantien erwarten. Auch wenn es solche Abhängigkeiten nicht geben dürfte, kamen sie vor. Ursache wie so oft: weil es möglich war.

Die Trennung von Komponenten in eine Server- und eine Klienten-Komponente fand nur sehr selten statt. Eher gab es eine Komponente, welche die Vereinigungsmenge beider Funktionalitäten darstellt. Serverfunktionalität wurde maximal zur Laufzeit vor klientseitigen Zugriffen geschützt. Zum einen ist dies unschönes Design, zum anderen

nicht selten eine Sicherheitslücke. Laufzeitdaten kann jeder Klient lokal ändern und somit Aktionen auslösen, welche nur der Server starten dürfte.

3.3.2.3 Ereignisse

Bei Ereignissen in der aktuellen Architektur gibt es keine dynamischen Verbindungen zwischen dem Subjekt (Ereignisauslöser) und dem Observer (aufgerufene Managerkomponenten). Vielmehr verknüpft sich das Ereignis abhängig von der Programminstanz (Klient-Server) bereits zur Kompilierungszeit mit den Managerkomponenten. Dies erfordert das explizite Wissen über diese Komponenten und ermöglicht das Transferieren von Kontrollfluss-Entscheidungen in die Ereigniskomponente.

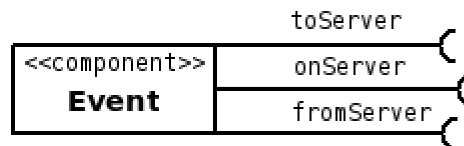

Diagramm 5: Ereigniskomponente in der Ist-Architektur

Ereignisse sind dadurch stark mit den Managerkomponenten gekoppelt. Durch das „Aufsaugen“ von Kontrollentscheidungen unterlaufen sie ihre Aufgabe als Konnektoren und bauen selber unnötig komplexe Systeme auf.

3.3.3 Zusammenfassung

Ergebnis ist ein Projekt, dessen Aufbau einem riesigen Objektnetz auf einer Ebene gleichkommt. Selbst das geplante Schichtenmodell existiert nur in der groben Planung, wird aber, wie oben dargestellt, umgangen. Ereignisse, welche normalerweise Kopplungen verringern sollen, werden falsch angewendet und erhöhen sogar noch die Kopplung und Komplexität innerhalb des Projektes.

3.4 Ziel-Architektur

3.4.1 Zielsetzung

Es sollen weiterhin die Grundzüge der Ausgangsarchitektur erhalten bleiben. Festes Ziel ist es, diese Grundpfeiler auszudefinieren und auf konkrete Architekturregeln sowie später in der Neuen Soll-Architektur praktische Implementierungsregeln zu setzen. Folgende Ziele sollen verfolgt werden.

1. Die oberste Abstraktionsebene soll durch **strikte Schichten** beschrieben werden.

2. Innerhalb der Schichten wird ein hierarchisches Objektnetz von Komponenten umgesetzt.

3. Zur Kommunikation zwischen unabhängigen Komponenten, insbesondere Schichten, werden Ereignisse eingesetzt.

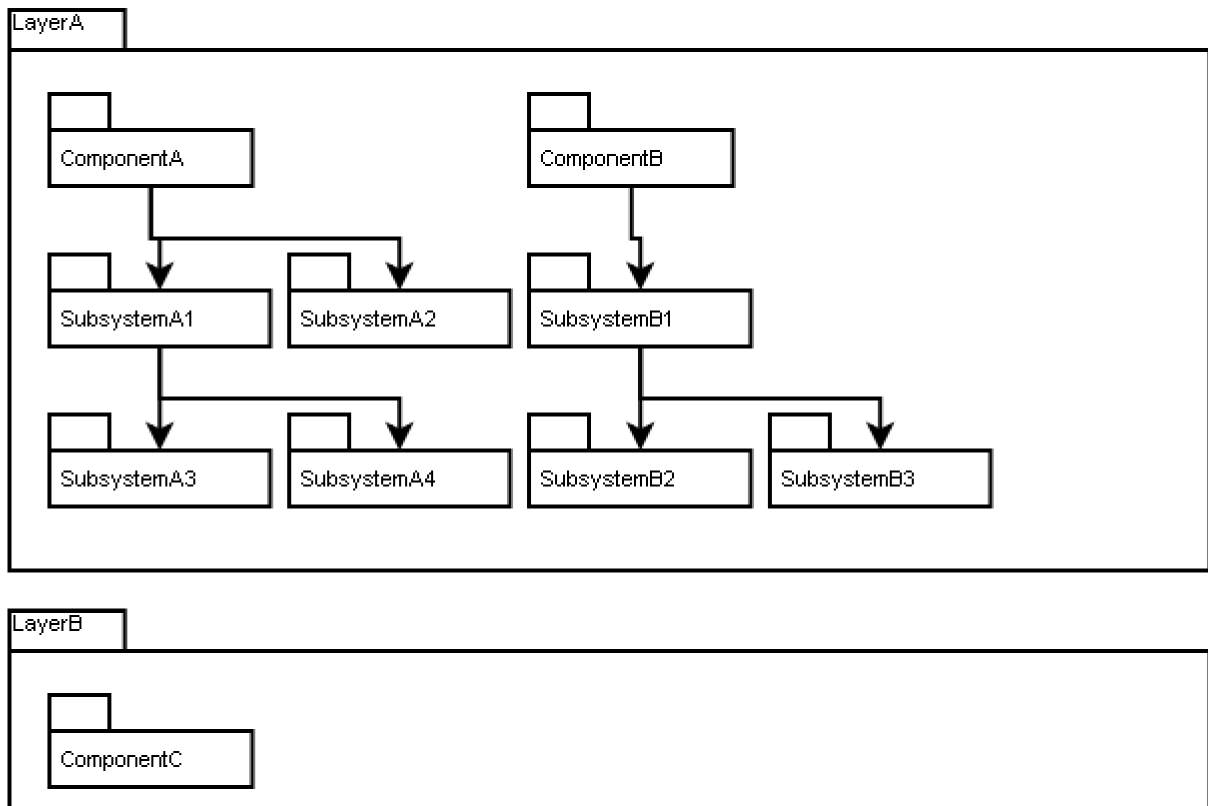

Diagramm 6: Zielsetzung für Ziel-Architektur auf oberster Abstraktionsebene

3.4.2 Architekturstil und Architekturregeln

3.4.2.1 Schichten

Angestrebt wird ein striktes Schichtenmodell. Entscheidender Antrieb für dieses Ziel ist die Plattformunabhängigkeit. Muss eine Schicht aufgrund einer plattformabhängigen Entscheidung ausgetauscht oder entfernt werden, sind komplexe Kopplungen ein Hindernis. Außerdem hat die Erfahrung gezeigt, dass die Möglichkeit, auf etwas direkt zuzugreifen, auf die einfallsreichste Art und Weise genutzt wird. Entscheidungen anhand von Daten aus der untersten Schicht werden dann auf Anwendungsebene getroffen, allein weil man es kann.

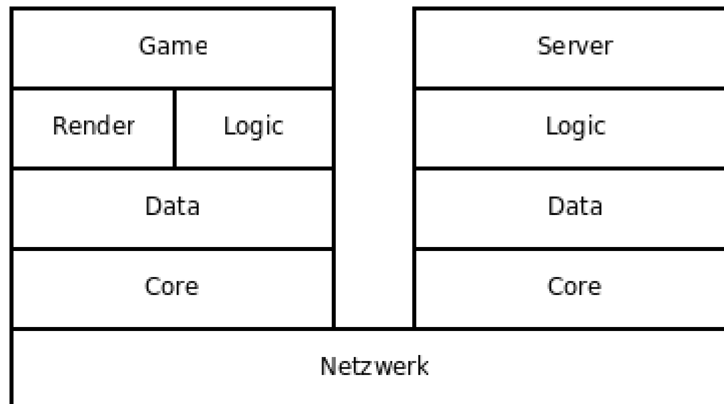

Diagramm 7: Schichtenmodell der Ziel-Architektur

Neu auf dieser Abstraktionsebene ist die Datenschicht. Diese wird eingeplant, um eine Trennung von Daten und Logik zu ermöglichen.

Besondere Aufmerksamkeit erfordert die grafische Darstellung von Daten. Dessen Anforderungen sind essentiell für die Schnittstellen zwischen den einzelnen Schichten insbesondere da es auf Serverseite die grafische Ausgabe nicht gibt.

Dies kann man in 2 Systeme auftrennen. In der Anwendungsschicht ist die Benutzeroberfläche angelegt, welche auf Klientseite eingesetzt wird. Diese nimmt Benutzerinteraktionen entgegen und stellt Daten dar. Die Darstellung in der Render-Schicht ist völlig passiv. Sie übersetzt praktisch die Daten der Spielwelt in eine grafische Darstellung. Ein Rückkanal ist nicht notwendig. Es bietet sich daher eine recht einfache Abstraktion in Model und View ohne echten Controller an.

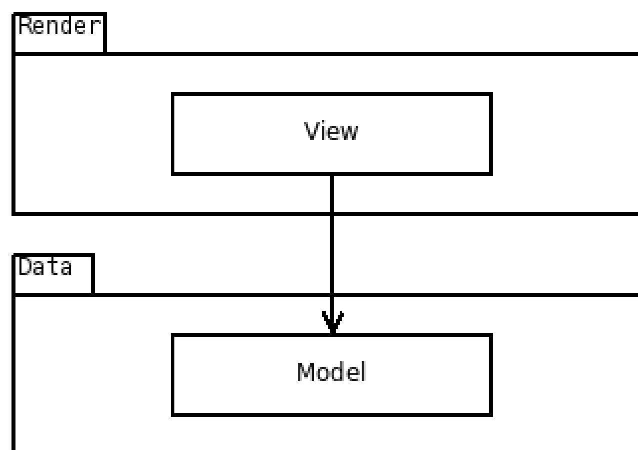

Diagramm 8: Model-View - Ansatz

Für die GUI wurde bereits ein Model-View-Presenter-Gedanke zu Grunde gelegt. Dieses System sollte strenger umgesetzt werden.

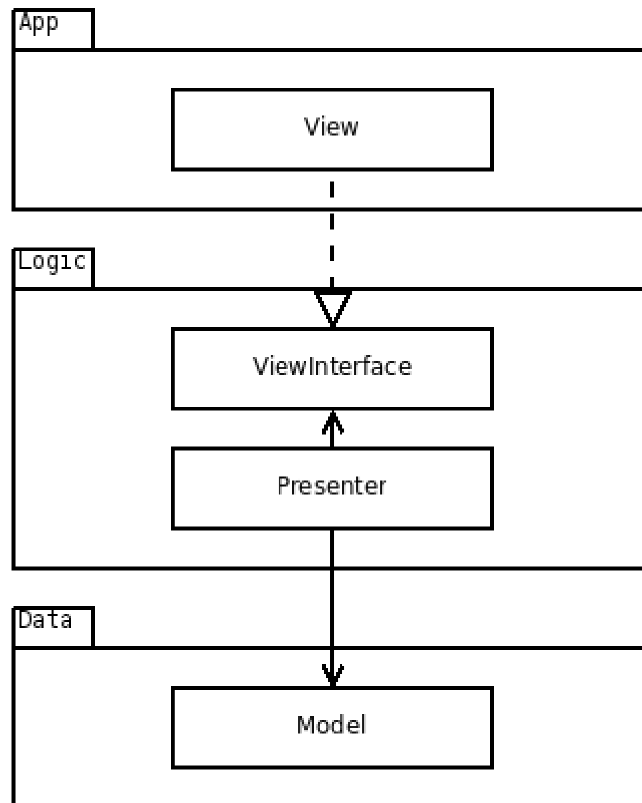

Diagramm 9: Model-View-Presenter-Ansatz

Ausgangslage:

Dass es Brüche in der aktuellen Schichtenarchitektur gibt, wurde bereits in der Analyse festgestellt. Im folgenden Bild werden diese noch einmal in einem Diagramm dargestellt.

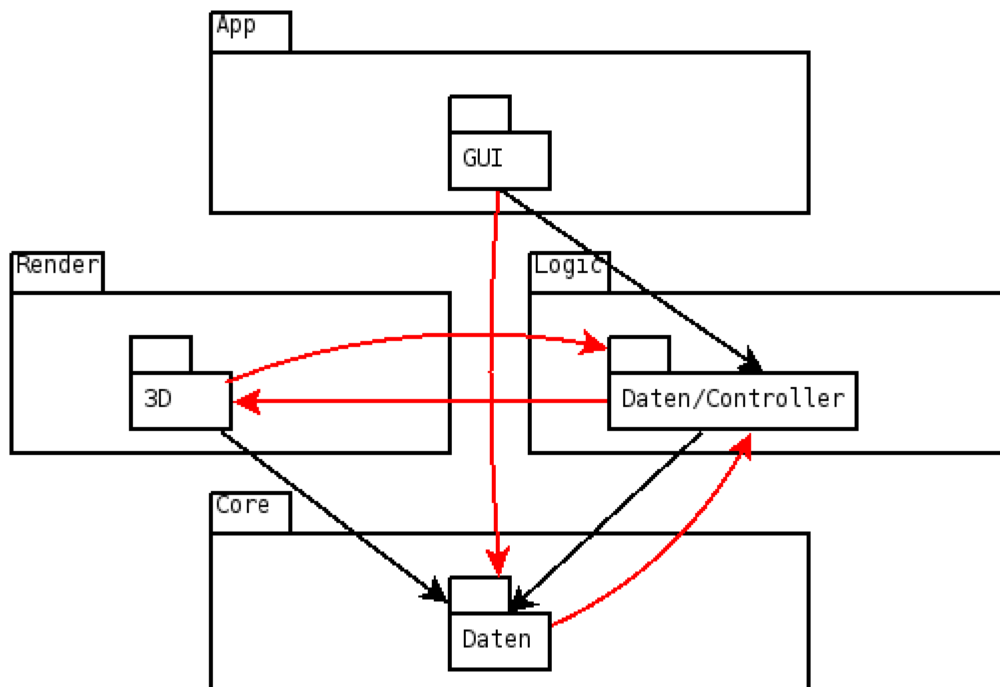

Diagramm 10: Kopplungen im Schichtenmodell zur Ziel-Architektur im ersten Iterationsschritt

Kanten markieren dabei Zugriffe auf eine andere Schicht. Sind diese schwarz, so handelt es sich um erlaubte. Rote dagegen weisen auf problematische Nutzungen hin. Konkretisierend können wir sagen:

– GUI nutzt Daten für eigene Logikentscheidungen

– 3D nutzt Daten aus Core und Logic

– Logic kontrolliert Vorgänge in Render

– Daten aus Core haben Kopplungen mit Daten aus Logic

– Daten und kontrollierende Logik sind eng verwoben

Umsetzung:

Zwei unerlaubte Kopplungen hängen mit der Lage von Daten in der Logik zusammen. Das heißt, der naheliegende Schritt ist das Zusammenziehen der Daten an einer Stelle, die der geplanten entgegenkommt. Dies wäre aktuell der Core. Allerdings sollte darauf geachtet werden, keine neuen Kopplungen zwischen Core und Daten einzuführen. Da Daten und Kontrolllogik nicht so schnell getrennt werden können, wird es anfangs dazu führen, dass man vorübergehend einen Großteil des Programmcodes aus der Logik verschieben wird. Allerdings wird dies mit dem nächsten Schritt (nächstes Kapitel) relativiert.

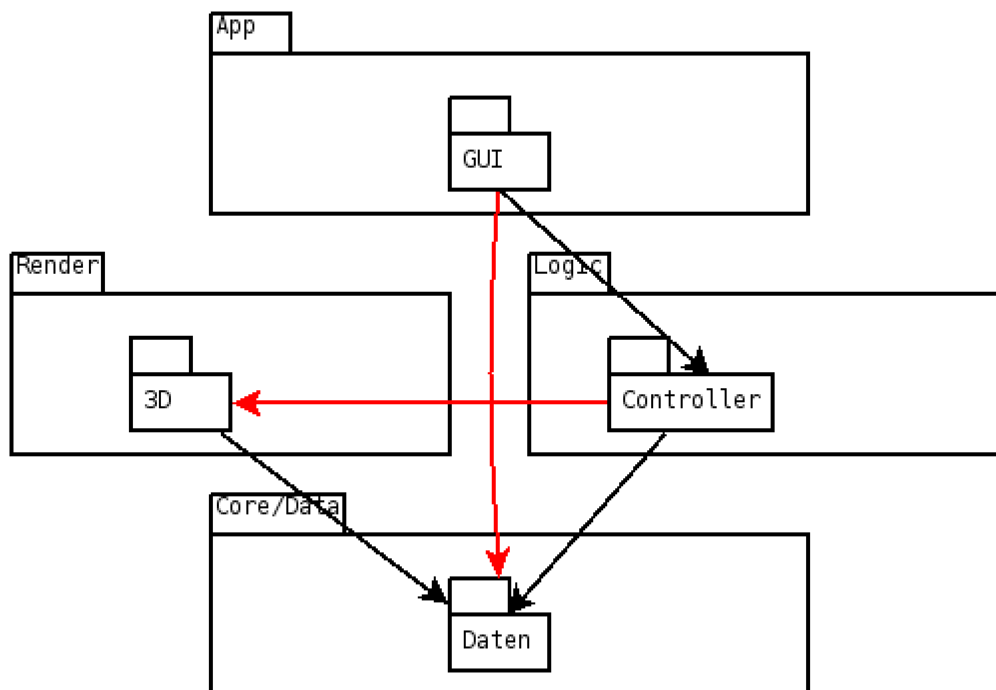

Diagramm 11: Kopplungen im Schichtenmodell zur Ziel-Architektur im zweiten Iterationsschritt

Dem Diagramm ist zu entnehmen, dass sich die Situation schon deutlich verbessern würde. Zeitmässig kann man von etwa 1 bis 2 Arbeitstagen ausgehen, um diesen Schritt abzuschließen. Allerdings würde er vermutlich den kompletten Arbeitsprozess blockieren.

Danach machen allerdings immer noch zwei rote Kanten deutlich, dass noch Umstrukturierungsbedarf vorhanden ist. Die direkte Datenabhängigkeit der GUI kann man angehen, indem man hier strikt das MVP-Konzept durchsetzt. Dafür müsste im vorliegenden Zustand eine Schnittstelle zwischen Logic und GUI eingesetzt werden,

welche diese Anorderungen erfüllen kann. Solche sind sogar bereits im System vorhanden, wurden nur nicht konsequent eingesetzt. Mit dem sogenannten UiSignaller können Zustände transparent aus der Logik abgefragt werden und bei Änderung wird die GUI über ein Slot-System benachrichtigt, vergleichbar zum Entwurfsmuster Beobachter (Observer). Auch war geplant Logikentscheidungen von der GUI in einen UiProxy abzukapseln, was in gewissem Teil auch umgesetzt wurde. Allerdings finden sich beide Strukturen in der Anwendungsschicht. Dies führt für den UiSignaller dazu, dass er nur mit der vorhandenen Logic-Schicht funktioniert, da er dort direkt auf Objekte zugreift. Die Entkopplung ist zwar mit dem Slot-System erreicht, allerdings nur innerhalb der Anwendungsschicht. Ändert man Definitionen in der Logik, muss auch der UiSignaller angepasst werden.

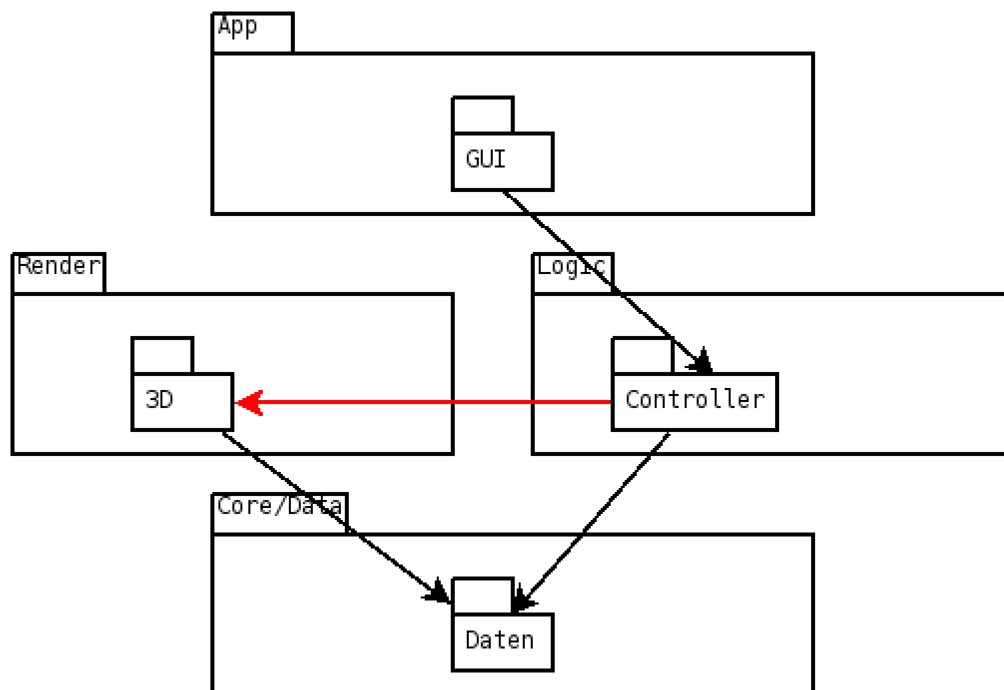

Diagramm 12: Kopplungen im Schichtenmodell zur Ziel-Architektur im dritten Iterationsschritt

3.4.2.2 Klient-Server

In der Ist-Architektur findet die Trennung von Klient und Server vornehmlich zur Laufzeit statt. Primär in der Ereignisbearbeitung, in welcher der Kernel die aktuelle Instanz und die Ereignisherkunft als Indikatoren nutzt, um eine der Abarbeitungroutinen (toServer, onServer, fromServer) auszulösen. Diese greifen dann auf Schnittstellen der Manager zurück. Da es in den Manager-Komponenten keine Trennung in Klient- und Serverteil gibt, vereinigt sich dort wieder der Kontrollfluss und diese Komponente muss dann erneut prüfen, auf welcher Instanz sie läuft.

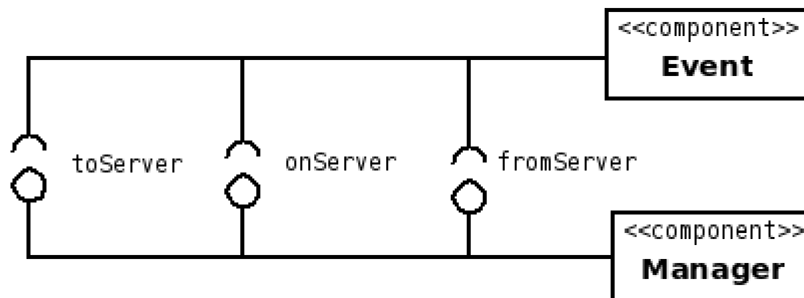

Diagramm 13: Vorliegende Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager

Durch das Vereinigen von Klient- und Servercode kreuzen sich Kontrollflüsse in einigen Methoden, was bei mangelnder Abfrage zu Fehlern führt.

Ziel 2, unterschiedliche Schnittstellen einzuführen, würde dem Problem nicht gerecht, da diese bei einem solch fluktuierenden Projekt mit mangelnden Verantwortlichkeiten schnell umgangen werden. Ziel ist daher, eine Trennung der Manager-Komponenten in Klient und Server anzugeben. Damit kann man bereits für den onServer Pfad eine eindeutige Zuordnung zu einer Komponente erreichen. Der Klient-Teil kann bis hier hin immer noch durch toServer und fromServer ausgelöst werden. Allerdings scheint toServer obsolet für weitere Betrachtungen. Es löst Aktivitäten aus, bevor das Ereignis vom Klient für den Server verpackt wird. Aktionen, die praktisch nicht das Ereignis selber tätigen muss, sondern wofür der Auslöser des Ereignisses verantwortlich ist. Daher entfällt es in der weiteren Planung. Daraus ergibt sich folgender Stand:

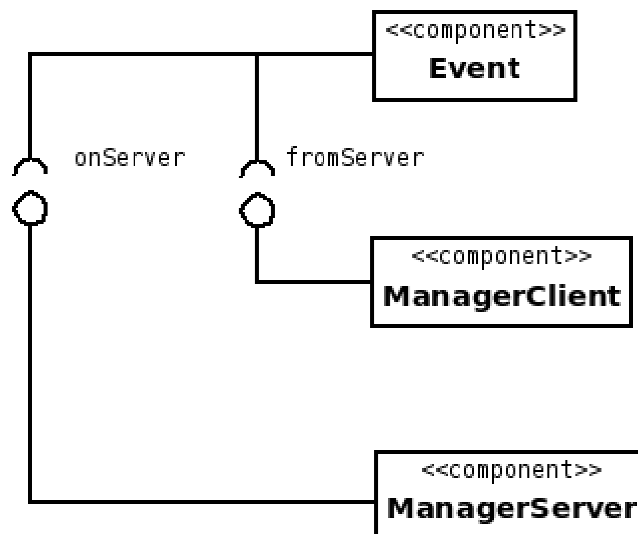

Diagramm 14: Erwünschte Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager

Als Resultat dieser Änderungen kreuzen die Kontrollflüsse sich nicht mehr in den Manager-Komponenten. Fehleranfällige Laufzeitabfragen entfallen, da sie über die Architektur abgedeckt werden.

3.4.2.3 Ereignisse

Der Ereignisbegriff ist bisher im Projekt eigenwillig. Ihn auf das allgemeingültige Verständnis zu bringen, ist möglich und erstrebenswert. Damit kann man die verbliebenen Laufzeitabfragen zwecks onServer und fromServer ersetzen.

Allgemein bindet sich eine Senke an ein Ereignis. Die Quelle löst das Ereignis aus.

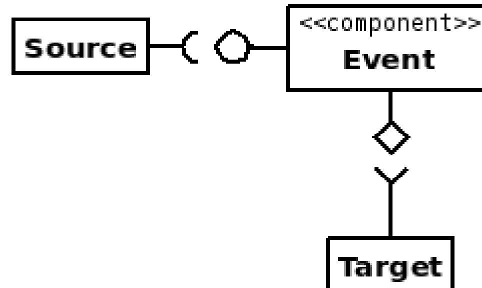

Diagramm 15: Ereignis in der Ziel-Architektur

Mit dieser Architekturregel für Ereignisse lässt sich der oben genannte Klient-Server-Ansatz für das Zusammenspiel von Ereignissen und Manager-Komponenten fortentwickeln. Die Komponenten werden dabei nicht mehr vom Event ausgewählt, sondern sie verbinden sich selber an das Ereignis. Da Klient- und Server-Komponente jeweils in einer eigenen Programminstanz laufen, kommt es dabei zu keinen Überschneidungen der Kontrollflüsse.

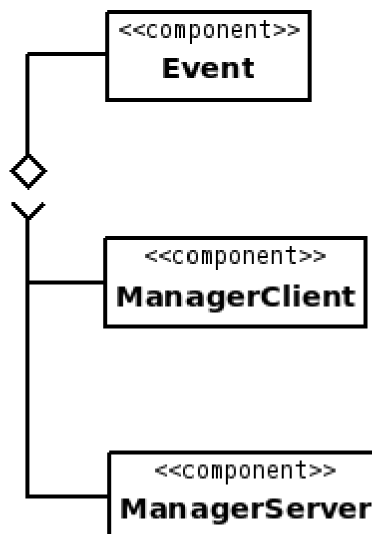

Diagramm 16: Erwünschte Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager unter Einbeziehung der Ziel-Architektur für Ereignisse

Es gäbe keine Ereignismethoden mehr, welche Aktionen in Managern starten. Damit kann man nun auch bei der Ereignisabarbeitung auf das Wissen der Manager verzichten und spart Komplexität ein. Auch entfällt der Laufzeittest des Ereignisses, in welcher Programminstanz es gerade auslöst.

Nur Kopplung der Manager mit dem Ereignis finden zum Initialisierungszeitpunkt statt (selbst dies könnte man auf die Kompilierungszeit verschieben). Alle anderen Entscheidungen können damit jedoch von der Laufzeit in die Architekturplanung verschoben werden.

3.5 Neue Soll-Architektur

3.5.1 Nutzen gegen Aufwand

Beim Übergang von der Ziel-Architektur auf die Neue Soll-Architektur muss vor allem die praktische Umsetzbarkeit geprüft werden. Oft ist die Ideallösung einfach nicht mehr mit vertretbarem Aufwand in die vorliegende Implementierung umzusetzen. Dann müssen Kompromisse eingegangen werden, die allerdings das Ergebnis nicht zu einer Flickenlösung werden lassen.

3.5.2 Architekturstil und Architekturregeln

3.5.2.1 Schichten

Bei dem Architekturstil des Schichtenmodells sind praktisch keine Abstriche nötig. Die strikte Umsetzung ist nicht nur wünschenswert, sondern sollte auch ohne große Probleme möglich sein. Dies ist vor allem auf die relativ einfache Struktur des Architekturstils zurückzuführen. Die ersten Architekturregeln sollten schon in der Alten Soll-Architektur gelten.

- Untersysteme einer Schicht besitzen gemeinsame Merkmale und Aufgaben.
- Schichten können nur auf darunter liegende Schichten zugreifen.

Neu hinzugekommen ist lediglich die Verschärfung hinsichtlich der Sichtbarkeit.

- Ein striktes Schichtenmodell erlaubt nur Zugriffe auf die direkt darunter liegende Schicht.

Dies ist auf lange Sicht notwendig, um eine Austauschbarkeit der Schichten zu gewährleisten.

3.5.2.2 Klient-Server

Bisher sind die Klient- und Server-Funktionalitäten von Manager-Komponenten im Projekt zusammen entworfen und weiterentwickelt worden. Dabei greifen sie auf gleiche Daten und Funktionen zurück. Eine ideale Trennung mit minimalen Überschneidungen würde vom Zeitaufwand mindestens die Größe erreichen, die für eine Neuentwicklung zu veranschlagen ist. Daher wird eine Lösung angestrebt, welche die Manager-Komponenten nach außen der Ziel-Architektur entsprechen lässt und intern eine schrittweise Umstrukturierung erlaubt.

Dafür wird es erlaubt, dass sich die Klient- und Server-Komponenten so weit wie nötig auf die alte Manager-Komponente berufen, also an diese kontrolliert weiterdeliegieren.

3.5.2.3 Ereignisse

Realistisch gesehen ist es nicht möglich, alle Ereignisse komplett in einem Rutsch parallel zur restlichen Entwicklung zu modifizieren. Deshalb wird es in Teilprobleme zerlegt.

Die toServer-Funktionalität lässt sich entweder vor das Auslösen des Ereignisses oder

in dessen Konstruktor verschieben. Was hierbei der beste Ort ist, muss von Fall zu Fall identifiziert werden.

Die Umstellung auf Ereignisse auf das Observer-Prinzip ist weniger trivial. Die Zahl der Ereignisimplementierungen und deren Umfang lassen einen vergleichsweise hohen Arbeitsaufwand erwarten.

Ereignisse benutzen nicht selten mehr als eine Manager-Komponente. Stellt man eine Manager-Komponente um, ist man gezwungen, alle abhängigen Ereignisimplementierungen ebenfalls anzupassen. Das heißt, die Umstellung eines Ereignisses samt seiner aufgerufenen Manager würde zu einer Lawine von Umbaumaßnahmen führen. Dies widerspricht jedoch dem Ziel der kleinen Refactoring Schritte, welches auch im Zusammenhang mit großen Softwareprojekten wichtig ist [RefInGroSoftw].

Es muss also die Möglichkeit geben alte und neue Architektur parallel unterstützen zu können. Daraus ergibt sich eine Hybridlösung.

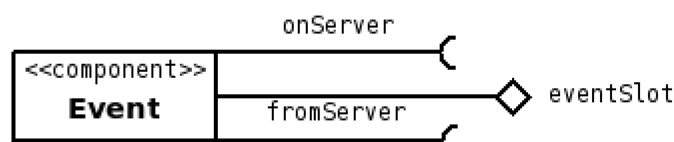

Diagramm 17: Hybridlösung der Ereignis-Komponente

So lässt sich das Projekt schrittweise umstellen, wobei neue Manager zwingend das Observer-Prinzip unterstützen müssen. Bei einer weiteren Iteration der Restrukturierung lässt sich dann überprüfen, ob man auf diese Hybridlösung komplett verzichten kann.

3.5.3 Implementierungsregeln

Systeme (Schichten, Komponenten, Untersysteme, etc.) werden nicht durch Singletons [GoF] oder andere global zugängliche Strukturen implementiert, solange dies nicht zwingend notwendig ist. Die Sichtbarkeit von Systemen wird alleine über die Architektur bestimmt und nicht über das Einbinden von Deklarationen (Headerdateien)!

Die Architekturstruktur muss sich auch in der Verzeichnisstruktur wiederfinden. Verschiebt man später ein System, sollte dies auch mit einer einfachen Kopieraktion zu bewerkstelligen sein. Zudem kann man über beschränkte „include“-Pfade so auch die Architektur weiter manifestieren. So dürfte ein System nur Subsysteme einbinden, die sich unterhalb im Pfad befinden oder globale Bibliotheken und Typendefinitionen.

3.5.3.1 Schichten

Bisher wurden Schichten als dynamisch gebundene Bibliotheken implementiert. Leider kann damit auch auf Schichten mehr als einer Ebene zugegriffen werden. Das macht es nötig, andere Mechanismen zur Zugriffskontrolle einzuführen.

Eine Schicht definiert sich als Klasse, welche seine untergeordneten Schichten als private Datenmember hält. Dadurch ist ein striktes Schichtenmodell, mit der Option möglich auf untergeordnete Schichten durchzureichen.

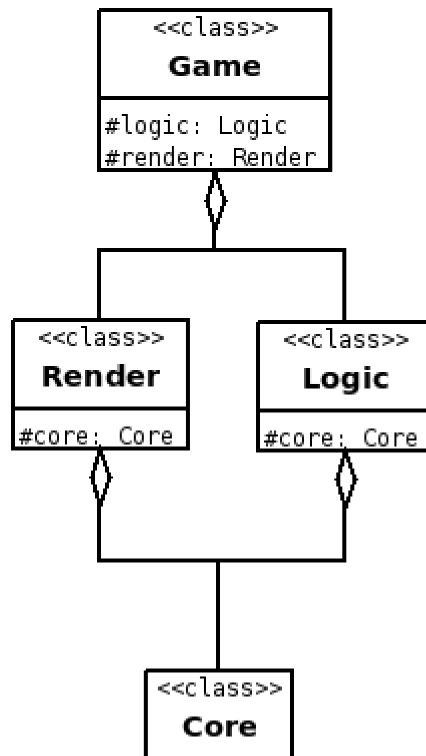

Diagramm 18: Implementierungsregel für das strikte Schichtenmodell

3.5.3.2 Klient-Server

Klient- und Server-Komponente werden sich fast immer Datenstrukturen und/oder Funktionen teilen. Damit lassen sich diese gemeinsamen Strukturen in einer geteilten Komponente zusammenfassen. Diskussionsbedarf erfordert hier einzig die Art und Weise, wie die Komponente eingebunden wird.

Komposition ist der Vererbung vorzuziehen, da sie eine geringere Kopplung darstellt. Allerdings habe ich mich in diesem Fall für eine private Vererbung entschieden. Diese entspricht bis auf ein paar Ausnahmen einer Komposition.

Die Ausnahmen wären:

- Neben den öffentlichen Elementen sind auch geschützte (protected) sichtbar
- Aufrufe der gemeinsamen Komponenten innerhalb der Tochterklasse müssen nicht delegiert werden. Sollten jedoch, der Lesbarkeit wegen, explizit angesprochen werden (z.B. ManagerCommon::foobar()).

Der einfache Wechsel zwischen öffentlicher und privater Vererbung bringt aber gerade für den Umstrukturierungsprozess deutliche Vereinfachungen. Diese werden im nächsten Abschnitt zum Übergang detailliert beschrieben.

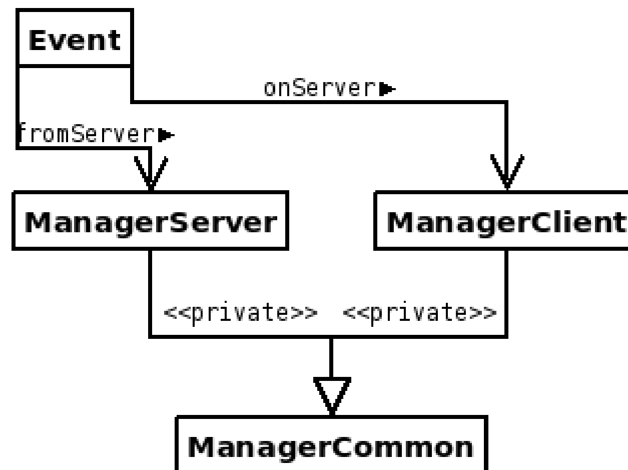

Diagramm 19: Implementierungsregel für das Zusammenspiel von Ereignis und Manager mit alter Ereignisbasis

3.5.3.3 Ereignisse

Hier bietet es sich an, für Interprozesskommunikation auf die bereits bestehende Event-Infrastruktur zurückzugreifen. Für die Kommunikation zwischen Schichten und Komponenten empfiehlt sich eine Signal-Slot Implementierung ähnlich der bereits genutzten libsigc++, die man jedoch über die Event-Schnittstelle anbietet, da sie auf der gleichen Semantik beruhen.

3.6 Übergang Implementierung zur Neuen Implementierung

3.6.1 Schichten

Folgend wird beschrieben wie man praktisch vorgehen wird, um der Neuen Soll-Architektur zu entsprechen. Dies umfasst im ersten Beispiel das Überarbeiten der Schichtenschnittstellen zwischen der Application- und Logic-Schicht, um bei einem überschaubaren Umfang zu bleiben, hier im speziellen der GUI in der Application-Schicht.

Danach wird das Vorgehen skizziert, um eine Datenschicht zwischen Logic- und Core-Schicht einzuführen. Dabei kommt es darauf an, ein geordnetes Vorgehen zum Verschieben der betroffenen Komponenten zu finden.

3.6.1.1 Entkopplung von GUI und Logic-Schicht

Analyse

In GUI-orientierten Anwendungsrahmenwerken wie zum Beispiel Qt haben sich Slots bereits bewährt. Ein äquivalentes Slotsystem wurde auch bei der GUI in „Sacred 2“ zu Grunde gelegt. Um Slots aus der Logic heraus auszulösen, wurde ein UiSignaler eingeführt. Dieser ist in der Anwendungsschicht und kann daher nur über Abfragen (Polling) Informationen einholen. Um über diese Struktur aus der Logik heraus aktiv Reaktionen auslösen zu können, wurden „Dirty-Flags“ genutzt. Parallel dazu wurden Applikation-Events eingeführt. Diese stellen die Verlängerung des Eventsystems in die Applikation hinein dar. Diese Events sind Ableitungen ihrer Logik-Äquivalente und lösen Slots aus. (siehe dazu „Interface-Bypass“ in [CoQualMan])

Die Schnittstelle für Aufrufe in die Logic-Schicht stellt der UiProxy dar. Da aber auch er sich in der Anwendungsschicht befindet, müssen alle von ihm genutzten Strukturen von dort aus sichtbar sein. Damit verliert der Proxy seine Schrankenfunktion einer „Fassade“ [GoF], da man auf die vermeintlich gekapselten Komponenten auch ohne den Proxy zugreifen kann. Dies führte unter anderem dazu, dass Logic-Events genutzt werden, um Aktionen aus der GUI heraus anzustoßen. Das sind meist auch die gleichen Stellen, an denen ungünstigerweise Daten aus unteren Schichten verknüpft wurden, um festzustellen, ob eine solche Aktion überhaupt erlaubt ist.

Da dieses Verhalten inzwischen zum praktischen Arbeiten gehörte, wurden solche Entscheidungen gekapselt. Daraus sind Strukturen wie der NetworkWrapper entstanden. Allerdings ebenfalls auf Anwendungsebene.

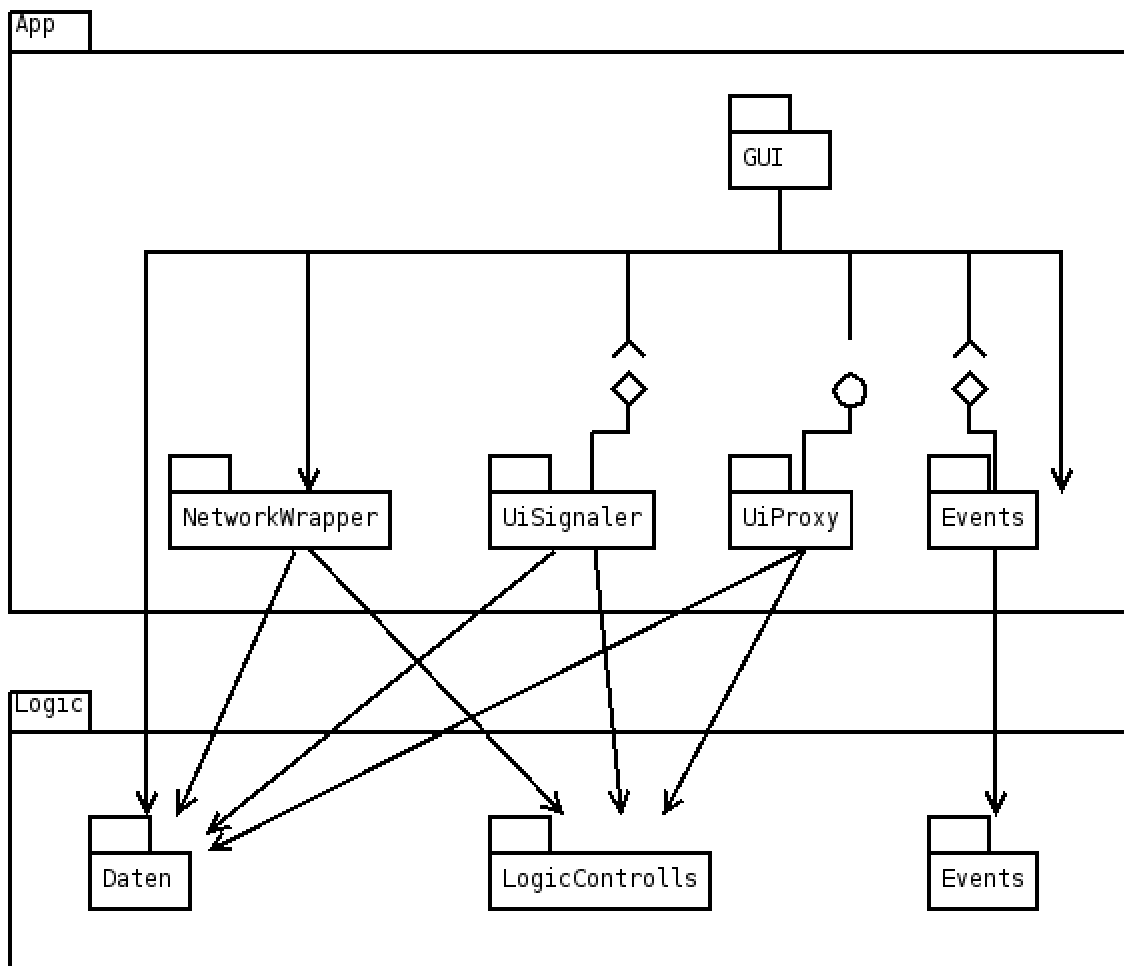

Diagramm 20: GUI relevante Kopplungen zwischen Game und Logic-Schicht in der Ist-Architektur

Durch die oben beschriebenen Entscheidungen ergeben sich zahlreiche Kopplungen zwischen den beiden Schichten, die sie praktisch nicht mehr austauschbar machen. Als besonders ungünstig hat sich die entstandene Spiellogik in der Anwendungsschicht/GUI ergeben. Entwicklungsplattform war die PC-Version, welche allerdings nicht die GUI mit den Konsolen teilt. Diese mussten die komplette Logik nachprogrammieren. Dies war nicht nur extra Arbeit, sondern auch eine riesige Fehlerquelle, da nicht offensichtlich ist, wo Logik fehlt.

Entwurf

Betrachtet man sich UiSignaler und UiProxy so kann man feststellen, dass ein großer Teil der Abhängigkeiten von ihnen ausgeht. Auf dem zweiten Blick erkennt man allerdings recht schnell, dass diese Abhängigkeiten aus einem kleinen Architekturfehler herrühren. Beide Strukturen wurden in die falsche Schicht eingeordnet. Slots („Observer“) und „Fassade“ sind eigentlich für die Verbindung zwischen den Schichten gedacht, werden allerdings nur innerhalb der Anwendungsschicht genutzt. Ein Verschieben dieser beiden Strukturen in die Logic-Schicht löst das Problem auf relativ unkomplizierte Weise. Das öffnet den Weg zu einer klar definierten Schnittstelle der Logic-Schicht über die Slots und Fassade.

Dieser Schritt erlaubt es, im Anschluss auch die gewachsenen unerlaubten Zugriffe und verlagerten Funktionen einzudämmen und in die Logic-Schicht zu schieben. Die Erweiterung des Eventsystems auf die Anwendungsschicht kann zurückgebaut werden. Die Events der Logik können das aktive Anstoßen dann über den Signaler übernehmen.

Spiellogik sollte danach für die GUI nur noch über die definierte Schnittstelle erreichbar sein.

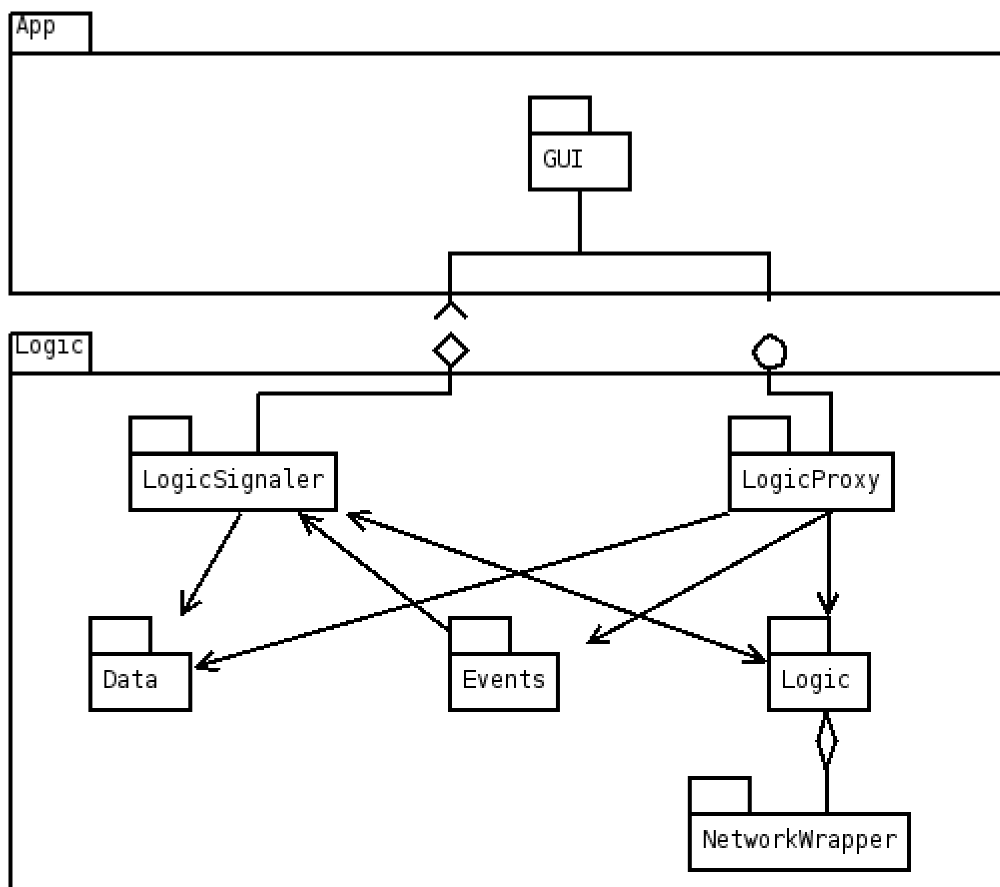

Diagramm 21: GUI relevante Kopplungen zwischen Game und Logic-Schicht in der Neuen Soll-Architektur

<u>Umsetzung</u>

1. UiSignaler in die Logic-Schicht bewegen

2. UiSignaler umbenennen in LogicSignaler

3. UiProxy in die Logic-Schicht bewegen

4. UiProxy in LogicProxy umbenennen (Es stellt praktisch keinen Proxy dar. Um jedoch Verwirrung vorzubeugen wird der Begriff beibehalten.)

5. Entfernen der Applikation-Events und deren Funktionalität in die äquivalenten Logic-Events bewegen

6. NetworkWrapper in Logic-Schicht bewegen und Aufrufe dessen über den LogicProxy umleiten

7. Funktionalität, die auf Daten zugreift, hinter den LogicProxy bewegen und Aufrufe auf diesen umleiten.

Aufwandsplanung

Schritte 1 bis 4 werden mit einem Personentag veranschlagt.

Schritt 5 wird mit ein bis zwei Personentagen veranschlagt.

Schritt 6 wird mit einem Personentag veranschlagt.

Schritt 7 wird mit vier Personentagen veranschlagt. Hier wird davon ausgegangen, das viel Codeblöcke zerlegt werden müssen, um GUI vom Logic-Code zu trennen. Dies ist wesentlich zeitaufwendiger als die vorangegangenen Schritte

3.6.1.2 Auftrennung in Daten und Logik

Definiertes Ziel:

Es wird eine Trennung von Daten und zustandsloser Logik als verbindliches Ziel festgelegt. Dies soll der Verständlichkeit des Systems dienen und Götterklassen minimieren. Kopplungen werden aufgelöst und die Kohäsion soll steigen. Ist die Trennung einmal strikt durchgesetzt, erschwert sich auch das Neudefinieren von unnötiger Logik auf Datenebene, weil schlichtweg die nötigen Abhängigkeiten nur auf Logikebene vorhanden wären. Nötig wäre jedoch ein striktes Kontrollieren, so dass keine oder nur minimale Zustandsdaten in die Logik wandern.

Ausgangslage:

Daten und Logik befinden sich teilweise im Core, aber größtenteils in der Logik. Core sind vor allem die Basisdefinitionen von cEntity. Diese sind dadurch in der Render- und Logic-Schicht sichtbar, was auch der wichtigste Grund für diese Zuordnung sein dürfte. Mit der anvisierten Datenschicht sind diese Strukturen dort besser aufgehoben, da der Core nur für elementarste Funktionalitäten zuständig sein soll, nicht jedoch für Daten.

Folgendes Diagramm dient als Abstraktion für den Ausgangszustand. Es werden bewusst nur Stellvertreter für Themenbereiche genutzt, um die Vorgänge verständlich zu halten. Die parallel zur Logic-Schicht stehende Render-Schicht wird hier außen vor gelassen. Sie soll eigentlich rein passiv arbeiten und würde die folgenden Beschreibungen nur verkomplizieren.

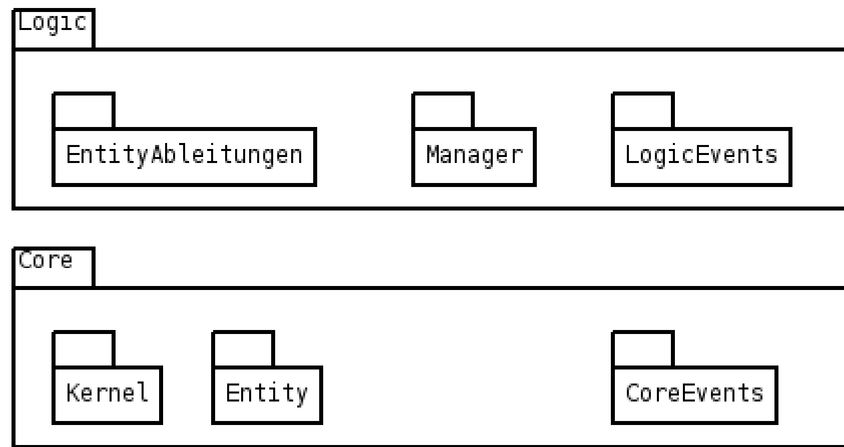

Diagramm 22: Einführung einer separaten Datenschicht - Ausgangssituation

Entity und EntityAbleitungen stehen hier stellvertretend für alle als Daten geplanten Strukturen. Diese sind meist in der Logic angesiedelt, besitzen allerdings eine Oberklasse in Core, um auch vom Renderer sichtbar zu sein. Als reine Datenstrukturen bringen sie oft extrem viel Logik inklusive Kopplungen mit.

Kernel steht für alle Strukturen, die unabhängig von Spiel und Daten funktionieren und daher ihre Position im Core einnehmen.

Manager beschreibt Strukturen in der Logik, dessen Einordnung nicht ganz eindeutig ist. Einige sind zustandslose Kontrollstrukturen, die sich rein statische Daten vorhalten. Andere sind eher dynamische Datencontainer.

<u>Übergang:</u>

Um die Philosophie der kleinen Schritte zu verfolgen, wird erstmal nur der Inhalt der Logic-Schicht über an die Stelle der geplanten Datenschicht verschoben und umbenannt. Zurück bleibt eine leere Logic-Hülle samt den Schnittstellen. Diese delegieren Aufrufe nun vorübergehend an die Datenschicht.

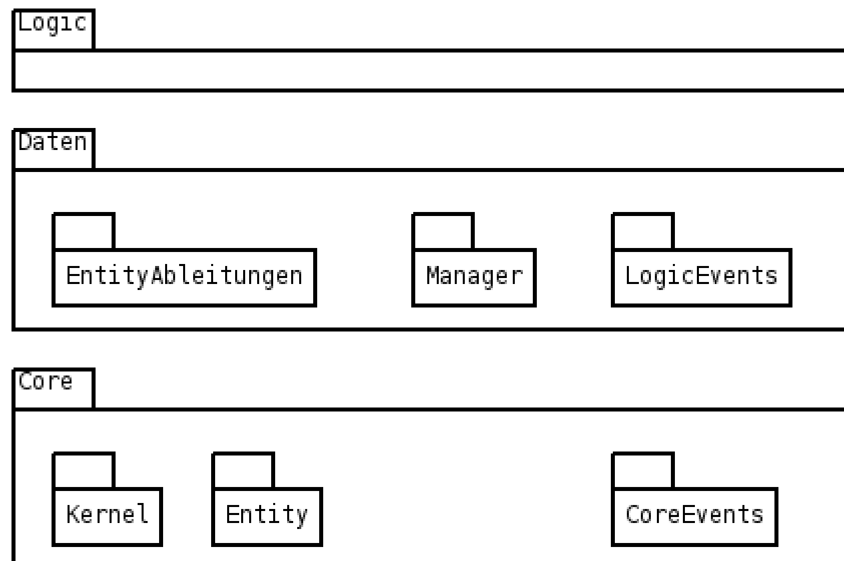

Diagramm 23: Einführung einer separaten Datenschicht – Iteration 1

Hier ist auch der richtige Zeitpunkt, eine Schnittstelle zwischen Data- und Logic-Schicht zu definieren. Diese wird anfangs um Übergangslösungen erweitert werden

müssen. Dabei sollte darauf geachtet werden, diese zu separieren, damit diese später nicht vergessen werden und die geplante Schnittstelle umgehen.

Führte der oben genannte Zwischenschritt zu einem voll lauffähigen System, folgt das Ausdefinieren der Core-Schicht. Entity und ähnliche Strukturen können nun in die Data-Schicht übernommen werden und sind dort weiter von Render- und Logic-Schicht nutzbar. Dabei muss bisher nichts weiter an den Schnittstellen zu den über Data liegenden Schichten geändert werden. Allerdings kann man nun eine verbindliche Schnittstelle für Core festlegen.

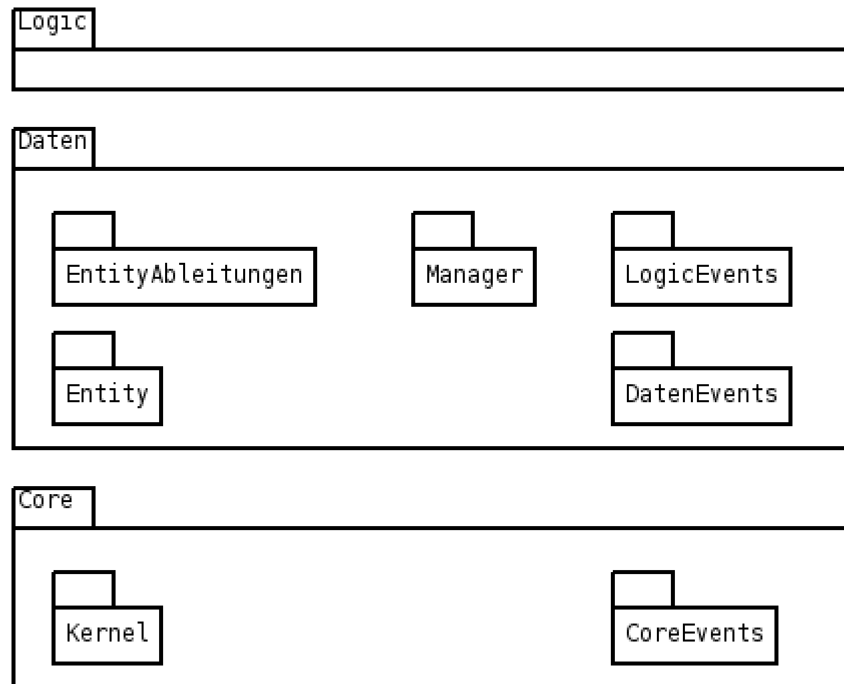

Diagramm 24: Einführung einer separaten Datenschicht – Iteration 2

Mit dem Festlegen der Core-Schnittstelle werden auch einige Events aus dem Core in die Data-Schicht wandern. Dabei sollte es sich um alle Events handeln, die auf den Daten gearbeitet haben, wie zum Beispiel die Events zum reinen Synchronisieren dieser.

Hat man diesen Punkt erfolgreich mit einer lauffähigen Version erreicht, dann sind die größten Hürden genommen. Die Schnittstellen sind definiert und der aktuelle Zustand ist nicht mehr wirklich schlechter als vor der Umstrukturierung.

Die verbotenen Verbindungen zwischen Logic und Render sind jetzt Kopplungen zwischen Data und Render und somit erlaubt, solange sie abwärts gerichtet sind. Die verbliebenen unerlaubten müssen dann über bekannte Techniken aufgelöst werden.

Core wäre nun eine eigenständige Komponente und nur noch über die Schnittstelle und ihre Garantien definiert. Es können in ihr somit Umstrukturierungsmaßnahmen unabhängig vom Rest des Projekts vorgenommen werden.

Das Fundament für die Trennung von Daten und Logik ist damit gelegt. Neue Strukturen müssen sich anhand der neuen Schnittstellen definieren. Dabei ist zu beachten, dass nicht unnötige Logik in die Datenschicht wandert. Dies ist nur sehr schwer durch Architekturgrenzen zu verhindern und bedarf Verständnis aller Beteiligten.

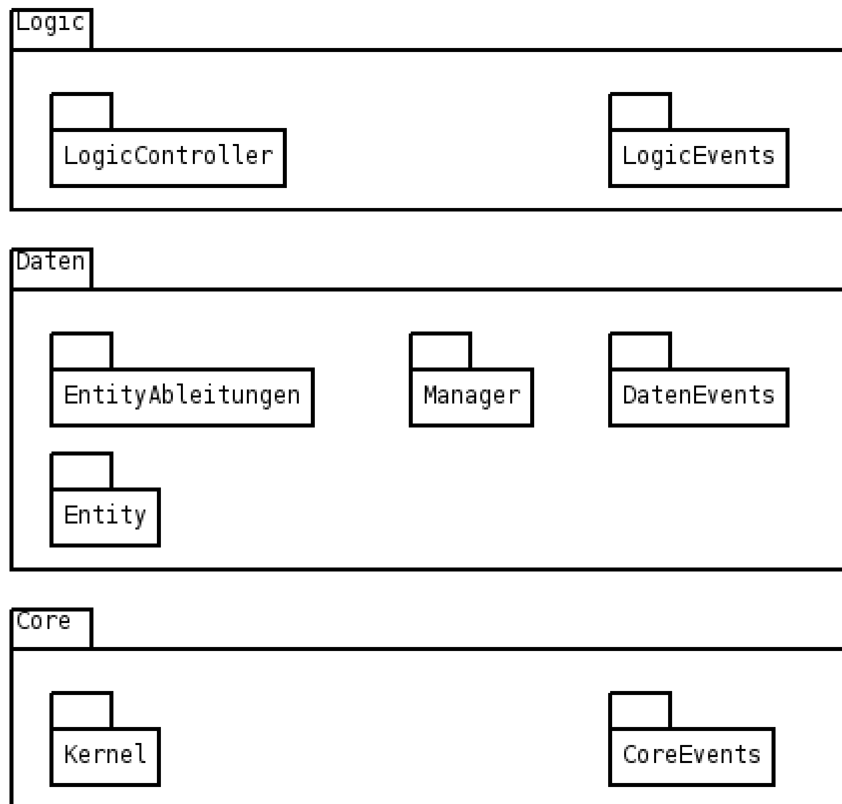

Diagramm 25: Einführung einer separaten Datenschicht – Iteration 3

Der Endzustand ist damit nur als Ziel definiert, dem man nur näher kommt, wenn man es bei jeder neuen Funktionalität oder Umstrukturierung vor Augen hat.

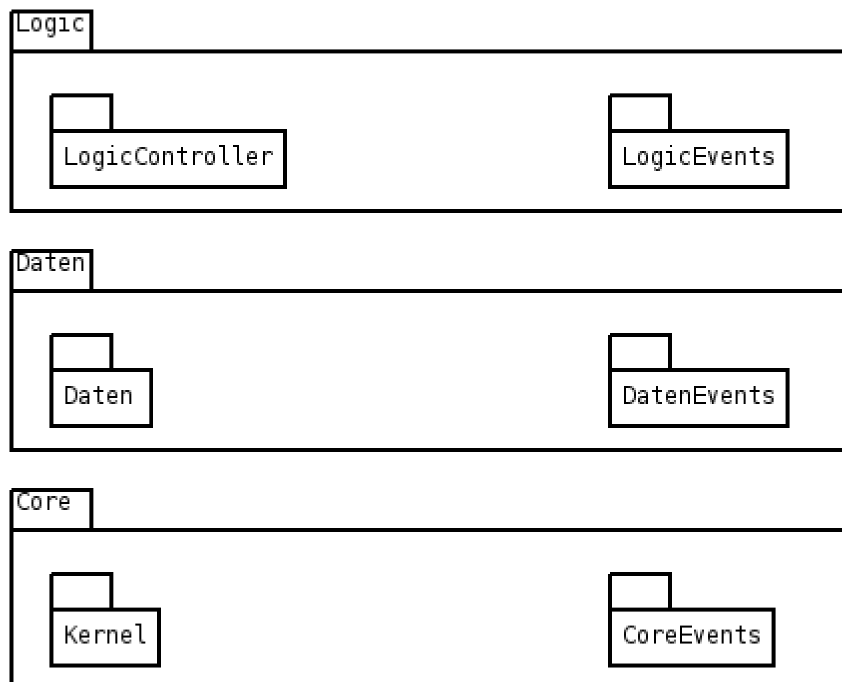

Diagramm 26: Einführung einer separaten Datenschicht – Endzustand

<u>Aufwandsplanung</u>

Dieser Umbau beeinflusst fast den kompletten in C++ geschriebenen Teil des Projektes. Die Anzahl der zu verschiebenden Komponenten ist im Vorfeld nur sehr schwer zu ermitteln, da dies ein Review aller Komponenten und ihres Kontext benötigen würde. Dementsprechend unmöglich ist eine Abschätzung der durch die Verschiebungen beeinflussten Komponenten.

Seriös wäre es, für eine erste Iteration nur bestimmte eindeutige Kandidaten in die Planung aufzunehmen. Aus der Erfahrung mit dem Verschieben dieser kann man später auch genauere Aufwandsabschätzungen für weitere Komponenten vornehmen.

3.6.2 Klient-Server

Wie beschrieben löst ein Ereignis in der Ausgangsimplementierung manuell die nötigen Funktionen im Manager aus.

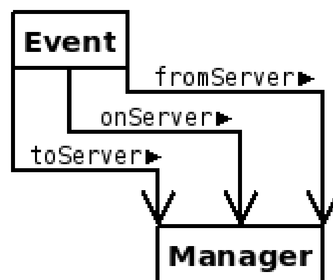

Diagramm 27: Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager in der Ausgangsimplementierung

Erster Schritt der Vereinfachung dieser Abläufe ist das Auflösen der toServer-Methode. Hier werden Aktionen ausgelöst, die vor dem Senden zum Server geschehen sollen. Diese sind aber nicht bedingt von Serverentscheidungen oder Ereignisabläufen. Daher können diese Aufrufe im Konstruktor stattfinden oder bestenfalls separat bereits vor der Ereignisauslösung. Diese Änderung sollte vor dem weiteren vorgehen abgeschlossen sein, damit keine Neubelebung dieser Funktion stattfindet, solange sie nutzbar ist. Zudem gäbe es sonst keine eindeutige Zuordnung von fromServer auf die Klient-Komponenten. Hierbei werden nur Aufrufe verschoben; es sollte dabei also keine Änderung an anderen Systemen nach sich ziehen. Das ermöglicht es auch, diesen Teilschritt parallel zur Entwicklung von weniger spezialisierten Programmierern durchführen zu lassen.

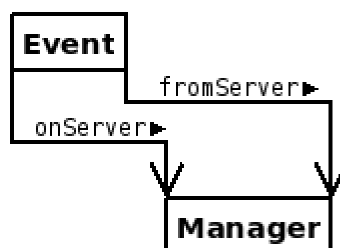

Diagramm 28: Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager nach dem ersten Umstrukturierungsschritt

Die onServer- und fromServer-Methoden sollten spätestens zu diesem Zeitpunkt auf Funktionsaufrufe von Managern reduziert werden. Bedingungen, Schleifen und anderer Programmcode sollten in diese Aufrufe verschoben werden.

Ist dieser Schritt abgeschlossen, gibt es für Klient und Server jeweils nur einen Kontrollfluss vom spezifischen Ereignis aus. Dies ermöglicht nun die Einführung von Klient- und Server-Manager-Komponenten, welche anfangs rein öffentlich vom alten Manager erben. Dadurch bieten sie von Anfang an die gleiche Funktionalität und das System bleibt voll lauffähig. Zu diesem Zeitpunkt bilden sie noch eine komplette Ist-Beziehung zum alten Manager.

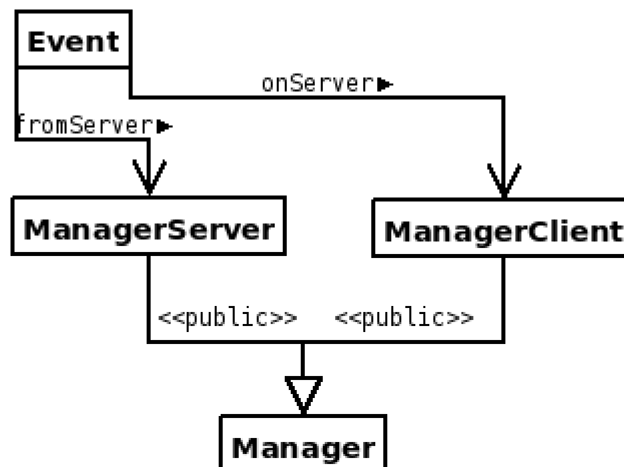

Diagramm 29: Klient-Server Trennung im Zusammenspiel von Ereignis und Manager nach dem zweiten Umstrukturierungsschritt

Mit dem Refactoring-Schritt „Hierarchie extrahieren“ [Fowler] lassen sich nun alle instanzspezifischen Methoden und Felder in die spezialisierten Klassen verschieben. Gemeinsame Methoden und Datenstrukturen verbleiben in der alten Managerklasse, welche zwecks Verständlichkeit im Namen angepasst werden sollte. Wird eine Methode von Klient und Server benötigt, so delegieren die abgeleiteten Klassen auf den gemeinsamen Teil.

Nach außen sollen dauerhaft nur die beiden instanzspezifischen Klassen sichtbar sein. Dies ist erreicht, sobald man ohne Fehler auf eine private Vererbung umsteigen kann. Die Lösung über die Sichtbarkeit der Vererbung ermöglicht es, die Ereignisse sofort umzustellen und dann schrittweise die Implementierung der Manager anzupassen. Gerade weil die Änderungen in den Ereignissen und Managerklassen dominoeffekt-artig Änderungen notwendig machen können, ist es notwendig ein unkomplizierte Zwischenlösung zu finden, welche kompatibel zur alten und neuen Implementierung ist. Die private Vererbung ist als finale Lösung vertretbar, da sie die Kopplungsstärke nach außen nicht erhöht, weil sie nur innerhalb der Klasse sichtbar ist. Dabei besitzt sie fast gleiche Eigenschaften einer Komposition über Instanzvariablen.

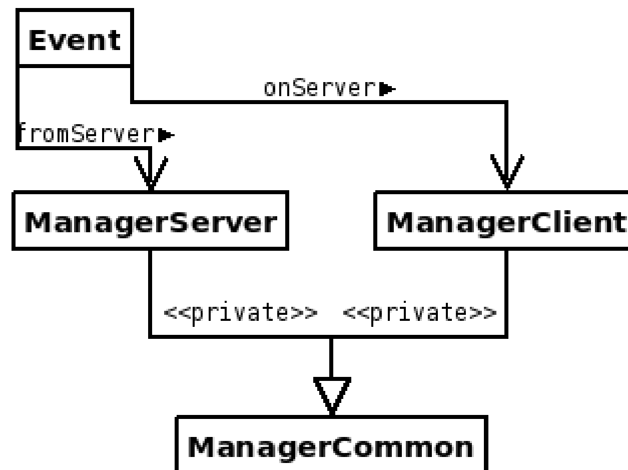

Diagramm 30: Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager im Endzustand

3.6.3 Ereignisse

Ereignisse leiten sich von einer Oberklasse ab, welche die Grundfunktionen zur Verfügung stellt. Diese definiert auch die polymorphen Methoden onServer und fromServer (toServer wird hier nicht mehr betrachtet), welche von den speziellen Ereignissen überschrieben werden.

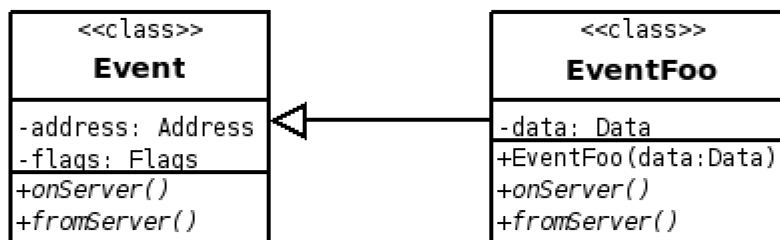

Diagramm 31: Allgemeines Ereignis und abgeleitetes spezialisiertes Ereignis

In der Ausgangsimplementierung ruft das Ereignis den Manager auf.

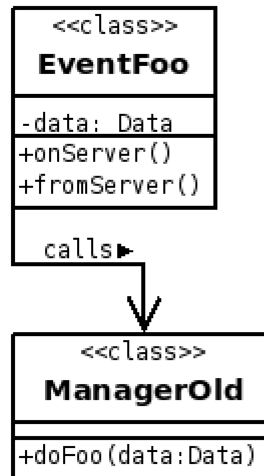

Diagramm 32: Ereignisimplementierung in der Ausgangsimplementierung

Ziel ist es, dass sich die Manager selber beim Ereignis anmelden und beim Auftreten dessen benachrichtigt werden. Dafür gibt es bereits freie Implementierungen wie libsigc++ [libsig], welche einen Producer-Observer-Mechanismus zur Verfügung stellen. Dafür wird ein Signal mit seinem Parameter-Datentyp als Member im Ereignis definiert. Die Managerklasse kann dann eine Callback-Funktion mit diesem Signal verbinden. Das ist idealerweise die bisher aus dem Ereignis manuell aufgerufene Methode. Da man nicht immer sofort alle genutzten Manager eines Ereignisses umstellen kann, wird man für den Übergangszeitraum zahlreiche Hybridlösungen als Ereignisse haben.

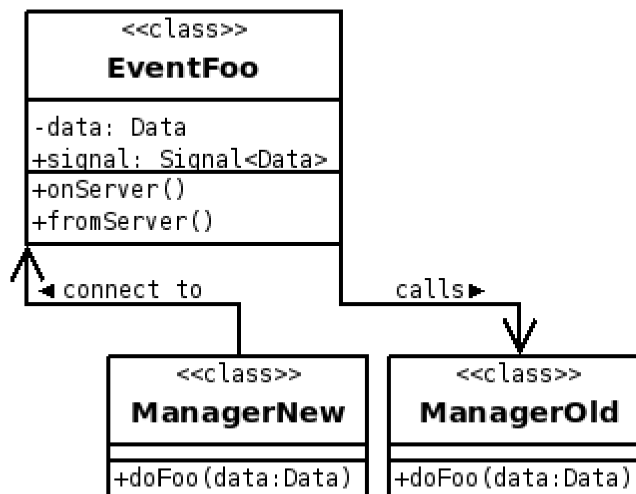

Diagramm 33: Ereignisimplementierung in der hybriden Zwischenlösung

Sind alle genutzten Manager auf das neue Prinzip umgestellt, kann man die onServer- und fromServer-Implementierungen weglassen. Das Ereignis braucht zu diesem Zeitpunkt nichts mehr über Klient oder Server zu wissen; lediglich welche Daten es transportiert, wie es diese serialisiert und welches Signal es zur Verfügung stellt.

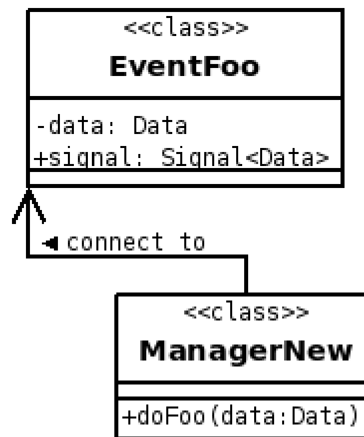

Diagramm 34: Ereignisimplementierung im Endzustand

Damit sind alle Ereignisse nur noch abhängig von den Daten, die sie transportieren. Dies ermöglicht es auch in einer späteren Umstrukturierungsiteration die Verlegung aller Ereignisse in eine gemeinsame Schicht. Bisher sind diese dort implementiert, wo sie das Wissen über die Manager haben.

4 Feingranulare Strukturebene

4.1 Gliederung

Im Folgenden werden die feingranularen Probleme anhand einer neuen Gliederung betrachtet. Dabei werden folgende Punkte abgehandelt:

Analyse

Enthält die Identifizierung des Problems anhand von Praxisbeispielen in Code oder einem Diagramm.

Kategorisierung

Einordnung der Problematik an Qualitätsindikatoren, Smells oder Vergleichbarem. Hier geht es darum, dem besprochenen Problem eine Systematik zu geben, um in weiteren Betrachtungen Metriken zur Erkennung anzuwenden und um für diese Systematik bekannte Umbaumaßnahmen (zum Beispiel Refactorings nach [Fowler]) abzuleiten.

Wenn das Thema einem Qualitätsindikator nach [CoQualMan] zugeordnet werden kann, werden immer auch gleich die Qualitätseigenschaften mit angegeben.

Entwurf

Hier wird anhand der Analyse der Neue Soll-Zustand definiert.

Umsetzung

- Schritt für Schritt Übergang vom Ausgangszustand in den Soll-Zustand

Aufwandsabschätzung

- Querabhängigkeiten
- geschätzter Personal-, Qualifikations- und Zeitaufwand

Bewertung

Hier werden Aufwand und Nutzen gegeneinander aufgewogen unter dem Aspekt, ob ein Umbau wie beschrieben möglich ist und unter welchen Vorraussetzungen.

4.2 QueryStatus Funktionalität

Analyse

Ähnlich einer abstrakten Methode kann man damit Methodenaufrufe von Unterklassen tätigen, auch wenn man nur ein Objekt der Oberklasse hat. Dafür wird der virtuellen Methode queryStatus eine MethodenID übergeben und optional drei „generische“ Parameter. Die MethodenID beginnt meist mit OCMD_, daher wird im Folgenden im Zusammenhang mit dieser Technik von OCMD gesprochen. Als Rückgabewert wird ein Unsigned verwendet. Dafür muss dann die Unterklasse queryStatus implementieren samt aller für sie wichtigen OCMDs. Kann sie das OCMD nicht bearbeiten, werden die

Parameter an die Oberklasse weitergegeben.

```
1.class cEntityAnimated
2.{
3.   virtual U32 queryStatus (eObjectCmd _cmd, U32 _param1=0, void* _generic=NULL, void*
     _generic2=NULL)
4.   {
5.        U32 ret = 0;
6.        switch (cmd)
7.        {
8.             case OCMD_GET_FOOBAR:
9.             {
10.                 return 1;
11.                 break;
12.            }
13.            default:
14.            {
15.                 ret = cEntity::queryStatus( _cmd, _param1, _generic, _generic2)
16.            }
17.       }
18.       return ret;
19.  }
20.}
```

Als augenscheinliche Vorteile werden dabei angeführt, dass man in der Oberklasse die virtuellen Funktionen nicht zu deklarieren braucht und es nicht zu Fehlern kommt, wenn das aktuelle Objekt diese Methode gar nicht implementiert.

Erkauft wird das unter anderem mit:

– Bruch der Typensicherheit (fast überall reinterpret_casts bei Parametern und Rückgabewerten; fehlende oder überflüssige Parameter werden nicht erkannt)
– komplexerem/undurchschaubarem Code
– Compiler und Analysewerkzeuge können Zusammenhänge nicht erfassen (Optimierungen, Fehlersuche)
– keine const-Korrektheit möglich

Man kann die Teilfunktionalitäten/Kommandos unterteilen in einmalig implementiert, was einer normalen Methode entspricht, und in mehrfach implementiert, was der Funktion einer virtuellen Methode gleich kommt.

Einmalig implementiert	551
Mehrfach implementiert	112
Gesamtimplementierungen	1235
Einmalig Aufgerufen	176
Mehrfach Aufgerufen	374
Gesamtaufrufe	3422
Tote OCMDs	131
Gesamtanzahl OCMDs	672

Tabelle 5: OCMD-Statistiken im Ausgangszustand

<u>Kategorisierung</u>

Als Smell nach [Fowler] sind auszumachen:

– **Switch-Befehle**: Die QueryStatus-Methode besteht aus einem riesigen Switch-Konstrukt.

Folgende Qualitätsindikatoren sind in diesem Zusammenhang von Interesse.

„Gottmethode“

Qualitätseigenschaft	**Relevanz**
Analysierbarkeit	50%
Modifizierbarkeit	50%
Stabilität	0%
Prüfbarkeit	25%
Austauschbarkeit	25%
Zeitverhalten	0%
Verbrauchsverhalten	0%
Wirtschaftlichkeitsfaktor	**Einschätzung**
Kostenzuwachs	25%
Unmittelbarkeit	50%

Tabelle 6: "Gottmethode" nach [CoQualMan]

„Labyrinthmethode“

Qualitätseigenschaft	**Relevanz**
Analysierbarkeit	100%
Modifizierbarkeit	75%
Stabilität	25%
Prüfbarkeit	100%
Austauschbarkeit	50%
Zeitverhalten	0%
Verbrauchsverhalten	0%
Wirtschaftlichkeitsfaktor	**Einschätzung**
Kostenzuwachs	75%
Unmittelbarkeit	75%

Tabelle 7: "Labyrinthmethode" nach [CoQualMan]

„tote Implementierung“

Qualitätseigenschaft	Relevanz
Analysierbarkeit	50%
Modifizierbarkeit	25%
Stabilität	25%
Prüfbarkeit	25%
Austauschbarkeit	0%
Zeitverhalten	0%
Verbrauchsverhalten	25%
Wirtschaftlichkeitsfaktor	**Einschätzung**
Kostenzuwachs	25%
Unmittelbarkeit	100%

Tabelle 8: "tote Implementierung" nach [CoQualMan]

Entwurfs

Die Umwandlung der OCMDs würde Hand in Hand mit der Umstrukturierung untypisierter Parameter gehen. Einen erheblichen Teil der Aufrufe wird man infolge dessen in normale Klassenmethoden überführen können, da der größte Teil der OCMDs nur auf einer Klassenebene implementiert ist und nichts anderes als eine Methode darstellt. Andere wird man durch virtuelle Methoden ersetzen können. Nach dem Rückbau dieser OCMD-Funktionalität sollte das System schon deutlich übersichtlicher sein und weniger Fehler noch zur Kompilierungszeit tolerieren.

Umsetzung

1 Wird das OCMD aus einer Schicht aufgerufen, welche eigentlich keinen Zugriff auf die Implementierung haben darf?

1.1 Ja - Den Aufruf mit einem TODO und Beschreibung der Situation versehen. Refactoring des OCMD abbrechen.

2 Wird der OCMD nur einmal im Code aufgerufen?

2.1 **Ja** – Die Funktionalität des OCMD an die Aufrufstelle verschieben und mit Punkt 4 fortfahren.

3 Findet bei der Verarbeitung des OCMD nur eine Delegation zu einer Methode/Funktion statt?

3.1 **Ja** - In diesem Fall kann man der Methode einen sinnvollen Namen geben; sie auf Const-Correctness hin überprüfen. Danach kann man diese Methode statt dem OCMD nutzen und mit Punkt 4 fortfahren.

3.2 **Nein** - Ist die Implementierung direkt hinter der Auflösung des OCMD, bietet sich „Methode extrahieren“ nach [Fowler] an.

4 OCMD aus der Enum-Auflistung entfernen und kontrollieren, ob der Code sich kompilieren lässt. Würde man nur die OCMD-Verarbeitung entfernen, würde dieser OCMD-Aufruf immer noch vom Compiler akzeptiert werden!

Aufwandsabschätzung

Die meisten OCMDs können zu Methoden umstrukturiert werden. Der Aufwand dafür ist gering. Es ist problemlos möglich jeden OCMD einzeln umzustrukturieren. Auch wenn der Aufwand gut teilbar ist, bleibt der Gesamtaufwand beträchtlich.

Bewertung

Eine Funktionalität wie queryStatus hat in eine produktiven Entwicklung nichts mehr zu suchen. Sie ist vollkommen durch sprachliche Mittel zu ersetzen. Der Aufwand ist nur so groß durch die Masse der OCMDs. Da diese Masse im gleichen Maße potentielle Fehlerquellen enthält ist der Umbau erfoderlich.

4.3 Parameter-Übergaben

Analyse

In großen Teilen des Projektes, speziell in der Logic-Schicht, hat es sich durchgesetzt, dass Parameter als möglichst unspezialisierte Datentypen übergeben werden. Erst in der Funktion wird die Instanz wieder differenziert, um dann an die nächste Stufe wieder als Basistyp übergeben zu werden.

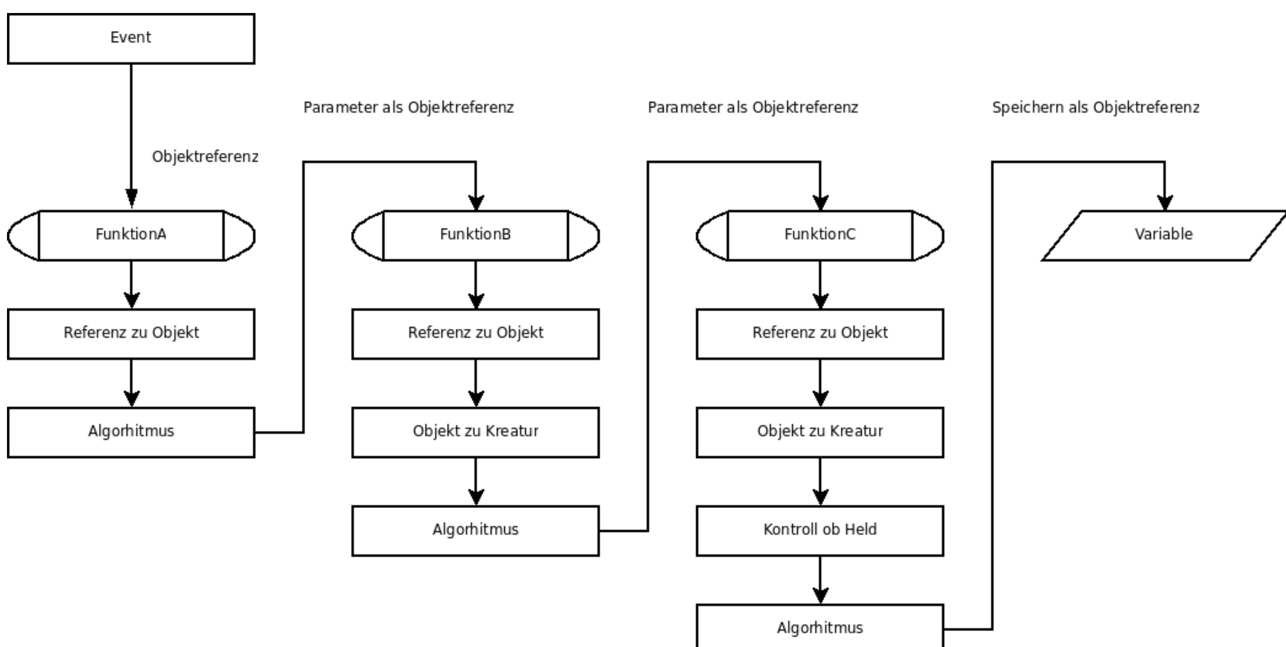

Diagramm 35: Untypisierte Parameterübergaben im Beispiel

Wie in der Abbildung 3.1 zu erkennen ist, entstehen so Aufrufkaskaden, welche stets die Eingabe auf die benötigte Unterklasse spezifizieren. Interessant ist eine solche Entwicklung auch für die Fehleranfälligkeit, da oft einfach davon ausgegangen wird, dass die Referenz (nicht zu verwechseln mit Zeigern, Referenz ist hier ein Indexwert, der die Entität im System identifiziert) eine bestimmte Objektinstanz beschreibt. Da harte Typenumwandlungen im Projekt keiner Regel unterliegen, wird dabei oft die Referenz direkt in eine spezielle Unterklasse (z.B.: Kreatur) umgewandelt. Da es zur Kompilierungszeit keine Überprüfungsmöglichkeit gibt, schleichen sich so schwer zu findende Fehler ein. Falsch spezialisierte Instanzen, welche versuchen, virtuelle Funktionen zu nutzen, die es gar nicht gibt sind nur ein mögliches Symptom. Verschärft wird das Problem durch die queryStatus-Aufrufe.

Dieses Vorgehen verletzt das „Don't-Repeat-Yourself“-Prinzip. Auf jeder Ebene wird die Instanz erneut identifiziert. Bei der Erstellung von Objekten muss deren Typ bekannt

sein. Diese Information zu vergessen, um sie später mehrmals auf komplexem Weg wieder herauszufinden, ist eine unnötige Redundanz.

Kategorisierung

Wird der übergebene Parameter direkt auf eine Form spezialisiert, so kann man den Qualitätsindikator **„allgemeine Parameter“** annehmen.

Qualitätseigenschaft	Relevanz
Analysierbarkeit	75%
Modifizierbarkeit	75%
Stabilität	50%
Prüfbarkeit	25%
Austauschbarkeit	75%
Zeitverhalten	25%
Verbrauchsverhalten	0%
Wirtschaftlichkeitsfaktor	**Einschätzung**
Kostenzuwachs	50%
Unmittelbarkeit	75%

Tabelle 9: "allgemeine Parameter" nach [CoQualMan]

Wird anhand der Typen-ID eine abhängige Spezialisierung vorgenommen, ist sich am Qualitätsindikator **„simulierte Polymorphie“** zu orientieren.

Qualitätseigenschaft	Relevanz
Analysierbarkeit	100%
Modifizierbarkeit	100%
Stabilität	100%
Prüfbarkeit	25%
Austauschbarkeit	75%
Zeitverhalten	50%
Verbrauchsverhalten	0%
Wirtschaftlichkeitsfaktor	**Einschätzung**
Kostenzuwachs	100%
Unmittelbarkeit	75%

Tabelle 10: "simulierte Polymorphie" nach [CoQualMan]

Entwurf

Eine Funktion sollte immer genau die Datentypen als Parameter haben, die sie benötigt und auf welche sie keinen Zugriff hat. Die eingehenden Daten sollten dabei

möglichst spezialisiert sein, um an der Übergabestelle schon zur Kompilierungszeit die Korrektheit der Daten garantieren zu können. Damit sollte bereits aus dem Funktionskopf hervorgehen, welche Daten als Eingabe wirklich benötigt werden. Enthält eine Funktion im Hauptpfad eine Spezialisierung eines Eingabeparameters, so ist davon auszugehen, dass sie auch nur speziell für diesen Datentyp funktioniert. Damit kann man festlegen, dass sie auch nur diesen Datentyp als Parameter akzeptiert.

Neben einer Anpassung der Parameterübergaben ist eine Typisierung der Objektreferenz zu überprüfen, da es immer noch zahlreiche Stellen geben wird, an denen eine Repräsentation über eine Referenz sinnvoll ist. Dies sind vor allem Ereignisse und Variablen die entkoppelt benutzt werden. Bei beiden Sachverhalten kann es vorkommen, dass ein Objekt zwischen Schreib- und Leseprozess zerstört wird.

Jedes Objekt hält neben seiner Referenz noch einen Typ. Es spricht nichts dagegen, diese beiden Werte in einem Datentyp zu vereinen, welcher an Stelle der bisherigen Referenz genutzt wird.

Umsetzung

1. Identifizieren der Stellen, an denen eine Referenz in den synchronen Programmablauf „injiziert“ wird. (Events, Auslesen von Variablen)

2. Alle genutzten Schnittstellen an dieser Stelle auf die höchstmögliche Spezialisierung überprüfen und anpassen. Rekursiv dessen genutzte Schnittstellen anpassen.

Diese Umsetzung wird nur schwer zu realisieren sein. Realistischer könnte folgender Ansatz sein:

1. Wird bei der praktischen Arbeit eine Schnittstelle entdeckt, welche Referenzen benutzt und diese in Instanzen umwandelt, so sollte die Schnittstelle angepasst werden und alle damit verbundenen Aufrufe. Wandert die Instanziierung dadurch in den Aufruf, ist dies hinzunehmen. Diese sollte bei der nächsten Iteration dann weiter in der Aufrufhierarchie nach vorne wandern, bis sie das Ziel (Ereignis oder Lesevorgang der Variablen) erreicht hat.

Aufwandsabschätzung

Praktisch ist eine reine Verschiebung von Code eine Hierarchieebene höher. Aufgerufene Funktionen müssen nicht zwingend mit umstrukturiert werden. Hier würde es reichen, die Parameter wieder auf die benutzte Form zu verallgemeinern. Es ist eine Umstrukturierung, die sehr gut zerstückelt und somit immer vorgenommen werden kann, wenn ein bestimmter Programmteil im Fokus liegt.

Die Umstrukturierung kann gut in die Aufgabenplanung eingearbeitet werden und ist daher in der laufenden Entwicklung möglich.

Bewertung

Der Code ist nach der Umstrukturierung weniger komplex, trifft weniger Laufzeitentscheidungen und verliert zahlreiche Redundanzen. Der Aufwand ist hoch, jedoch gut zu stückeln.

Diese Umstrukturierung ist zu empfehlen.

4.4 Umstrukturierung von Datentypen

Analyse

Nur was als Klasse implementiert wurde, hat seinen eigenen Datentyp. Andere Daten werden in Basisdatentypen gespeichert oder zumindest bei Parameterübergaben oder dem Serialisieren in einen solchen umgewandelt. Kommt es später zu einer Änderung des Datentyps; ist es dann notwendig, an allen Stellen im Code dies anzupassen. Nicht weniger aufwendig wird, das durch die vorhandenen harten Typenumwandlungen. Durch sie kann der Compiler keinen Hinweis bei der Umstellung liefern, da sich ab dem Cast die Spur des neuen Datentyps verliert.

Ähnlich verhält es sich bei der Nutzung „fremder“ Datentypen, deren Implementierung nicht änderbar ist. Container der STL beispielsweise sind im Grunde ein gutes Instrument, um Daten generisch zu speichern. Allerdings werden sie nicht selten zu blauäugig als Standardcontainer für einen Datentyp festgelegt. Die Übergabe in Subsysteme und Benutzung mit Iteratoren benötigt dann direkte Abhängigkeiten zu dieser einen Containerklasse.

```
21.std::vector<Entity> m_entities;
22.
23.getNearObjects( std::vector<Entity> _nearObjects )
24.{
25.  _nearEntities.clear();
26.  for( std::vector<Entity>::iterator it = m_entities.begin(); it != m_entities.end();
     it++)
27.  {
28.       _nearEntities.push_back( *it );
29.  }
30.}
31.
```

Die Daten werden offensichtlich abhängig von 2 Datentypen gespeichert. Dem std::vector und der eigenen Entityklasse. Erfüllt die Funktion von std::vector allerdings nicht mehr die Anforderungen gibt, es nur die Möglichkeit, den Container auszutauschen oder mit einer eigenen Klasse zu überladen. Spätestens hier wird man merken, an wie vielen Codestellen man Hand anlegen muss.

Kategorisierung

Als Smell nach [Fowler] sind auszumachen:

- **Neigung zu elementaren Datentypen**: Nutzung von nichtssagenden und nicht erweiterbaren elementaren Datentypen
- **Datenklumpen**: Typendeklarationen, die gehäuft zusammen auftreten oder direkte Abhängigkeiten besitzen
- **Schrottkugeln herausoperieren**: Würde man im Beispiel der STL-Container die Containerart von std::vector auf std::list ändern, muss dies an allen Punkten passieren, wo diese Deklaration genutzt wird.
-

Entwurf

Für jeden Wert, der etwas spezielleres ausdrückt als ein Basisdatentyp repräsentiert, sollte ein eigener Datentyp zur Verfügung gestellt werden. Dies kann im einfachsten Fall ein typedef auf einem Basisdatentyp sein. Allerdings ermöglicht dies schon eine rudimentäre Typensicherheit und erhöht zudem die Flexibilität für spätere

Änderungen. So ist es recht einfach, den typedef durch eine Klassendefinition zu ersetzen und somit dem Datentyp eine eigene Funktionalität mitzugeben. Daraus folgt aber auch eine einheitliche Notation. Es ist offensichtlich, dass bei der Deklaration einer Variable zwischen Klassen anderer Datentypen schnell gewechselt werden kann. Alle Namen von Datentypen sollten daher ohne Präfix definiert werden. Ansonsten kann es passieren, dass nach vielen Refactoring-Iterationen eine Variable mit CFoobar deklariert ist, obwohl diese nicht mehr als eine Typendefinition auf einen Integer darstellt.

Umsetzung

1. Feststellen, was für Daten eine Variable enthält. Kann man nicht anhand des bisher genutzten Datentyps eindeutig auf den Inhalt und den Wertebereich schließen?

2. Wird die Variable an Subsysteme weitergegeben?

3. Stellen mehrere Variablen im System diese Art von Daten dar?

4. Wurde zumindest einer der vorhergehenden Punkte mit „Ja“ beantwortet, dann ist eine Typisierung sinnvoll. Ansonsten ist davon auszugehen, dass der Variablentyp nur lokalen Einfluss hat (z.B. Zählvariablen). Dann kann die Refactorisierung hier abgebrochen werden.

Wurde Punkt 1 mit „Nein“ beantwortet, reicht eine Typendefinition um spätere Refactorisierungen am Datentyp (z.B. Kapselung in Klasse) transparent zu halten.

Aufwandsabschätzung

Eine Komplettüberarbeitung aller Datentypen wäre sicher eine Mammutaufgabe, da das komplette Projekt untersucht und angepasst werden müsste. Überschaubarer ist hier die Untersuchung von Komponenten. In ihnen kann man Variablen überprüfen und in der Theorie bis zu den Schnittstellen der Komponenten anpassen. Nur wurde schon besprochen, dass die Schnittstellen im Projekt nicht wirklich strikt umgesetzt sind. Auf Membervariablen wird sehr oft von außen zugegriffen. Damit wird oft ein Dominoeffekt losgetreten.

Erfahrungen mit dieser Umstrukturierung haben gezeigt das die Anpassung eines einzigen Variablentyps den Entwickler schon quer durch alle Schichten und zahlreiche Systeme führen kann. Andere Variablen lassen sich dagegen recht schnell umdeklarieren.

Bewertung

Eine Anpassung ist auf lange Sicht unerlässlich. Allerdings ist der Aufwand oft unberechenbar.

Empfohlen wird eine strikte Typisierung von neuen Variablen, welche an Komponentenschnittstellen genutzt werden. Eine Umdeklarierung alter Variablen sollte geschehen, wenn sie Teil einer Schnittstelle ist und der Aufwand abschätzbar bleibt.

4.5 Const-Correctness durchsetzen

Analyse

Die Const-Correctness erlaubt es dem Compiler zu warnen, wenn Schnittstellen auf unerlaubte Weise genutzt oder implementiert werden. In der Implementierung wird sie nur eingeschränkt genutzt. Eine weitreichende Nutzung wird allein schon durch die

queryStatus-Funktionalität verhindert. Diese unterstützt keine Const-Correctness und unterbricht deren Kette, wodurch alle benutzenden Funktionalitäten nicht mehr const sein können.

Kategorisierung

Hierbei handelt es sich um eine handwerkliche Voraussetzung. Const ist Sprachmittel, um Ein- und Ausgabeeigenschaften von Parametern festzulegen.

Entwurf

Getreu der Planung sollten Schnittstellen so umgesetzt werden, wie sie definiert wurden. Wenn ein Parameter nur lesenden Zugriff erlaubt, sollte dies mit const deklariert werden.

Umsetzung

Hier kann man das Problem nur von ganz hinten aufrollen. Da const-Methoden keine nicht const-Methoden aufrufen dürfen, muss am Ende der Aufrufkette mit der Umstrukturierung begonnen werden. Hier helfen Analysewerkzeuge, welche solche Aufrufketten schematisch darstellen können. Blätter in einem solchen Diagramm sind der erste Ansatzpunkt.

Aufwandsabschätzung

Mit passenden Analysewerkzeugen ist es möglich, die Umstrukturierung in kleine Etappen zu zerlegen. Dann ist dies auch parallel zur Entwicklung möglich. Fehlen diese Werkzeuge, wird man nur vereinzelte Methoden finden, welche am Ende einer Aufrufkette sind. Viel eher wird man sich eine Methode herauspicken, welche man recht unabhängig vermutet, und sich dann mit den Abhängigkeiten auseinander setzen müssen. Dabei wird man einen eigenen Codezweig erstellen müssen, um den Produktionsbetrieb nicht zu gefährden.

Bewertung

Für die Klarheit der Implementierung ist diese Umstrukturierung zu empfehlen. Ohne Analysewerkzeuge sollte jedoch genau abgewägt werden ob man ihr nicht besser eine geringere Priorität gibt.

4.6 Statische Methoden

Analyse

Das Deklarieren statischer Methoden möge auf den ersten Blick nur aus Leistungsgründen interessant sein (Compiler kann solche Methoden einfach und effizient inlinen). Auch muss kein this-Zeiger mehr übergeben werden.

Weitergehend sind sie jedoch ein wichtiger Indikator, wie man eine Methode in der Architektur einzuordnen hat. Von vielen Programmierern werden statische Methoden nie benutzt und globale Funktionen sind verpönt. Dadurch wird jede erdenkliche Funktion als Methode einem Objekt zugeordnet. Dies erhöht die Komplexität und je nach Parameter auch die Kopplung der Klassen.

Kategorisierung

Hierbei handelt es sich um eine handwerkliche Voraussetzung. Static definiert im Zusammenhang mit Methoden eine klassenweite Sichtbarkeit.

Entwurf

Eine statische Methode ist ein deutliches Indiz, dass diese bei geringer Abhängigkeit zur zugehörigen Klasse oder Namensraum an einen anderen Ort gehören kann.

Klassenmethoden können bei der Analyse wie eine normale globale Funktion behandelt werden. Der einzige Unterschied ist die Sichtbarkeit welche direkt von der Klasse bedingt ist.

Wird einer solchen Methode zum Beispiel ein anderes Objekt als Parameter übergeben, so gehört diese Methode wohl eher zu diesem Objekt.

Umsetzung

1. Methoden identifizieren, die keine Klassen-Member benutzen
2. Methoden als statisch deklarieren

Aufwandsabschätzung

Mit Hilfe von Analysewerkzeugen lassen sich solche Methoden effektiv aufspüren. Manuell wird dies nur im Rahmen eines kompletten Reviews möglich sein. Die Umstrukturierung an sich birgt kaum Risiken. Nur die Methode muss neu deklariert und alle Aufrufe angepasst werden. Querabhängigkeiten gibt es nicht. Eine Umstrukturierung parallel zur Entwicklung ist daher problemlos möglich.

Bewertung

Wenn man alle Methoden überprüfen will, entspricht der Aufwand sicher nicht dem Nutzen. Sinnvoll ist es beim Review einer Komponente, auch die Methoden auf die Richtigkeit ihrer Deklaration zu überprüfen.

4.7 Plattformabhängiger Code

Analyse

„Sacred 2“ wird für drei sehr verschiedene Plattformen umgesetzt. Windows/PC, Xbox360 und PS3. Low-Level-Code wurde über ein plattformspezifisches SDK ausgelagert. Es kapselt Operationen für zum Beispiel Dateisystemzugriffe, mathematische Optimierungen oder Schnittstellen für den Renderer. Dies ist größtenteils Code, der sich allein schon über die Hardwareabhängigkeit als plattformspezifisch qualifiziert.
Daneben gibt es auch Code, der auf den Plattformen ein unterschiedliches Verhalten der Anwendung vorgibt. Solche Unterschiede wurden oft „vor Ort“ mit if-then-else oder weit häufiger mit Precompiler-Anweisungen getrennt.

```
1.#ifdef CONSOLE_IMPL
2. // Consolen Code
3.#else
4. // PC Code
5.#endif
```

Solche Blöcke erstrecken sich in vielen Fällen über mehrere Bildschirmseiten und der nichtaktive Codeteil ist ausgegraut oder gar ausgeblendet. Wurde ein Plattformteil von einem Bug befreit oder um eine Funktionalität ergänzt, so vergaß man gleiches zu oft in den anderen Plattform-Abschnitten.
Bei der praktischen Umsetzung hat sich das Problem der auseinander gedrifteten Programmversionen besonders deutlich gezeigt. Die über den Preprocecessor gesteuerten Alternativpfade für Konsole und PC unterscheiden sich zum Beginn der Umstrukturierung in einem solchen Grad, dass kaum mehr für alle Pfade transparente Änderungen vorgenommen werden können.
Zusätzlich arbeiteten Konsole und PC circa fünf Monate auf verschiedenen Zweigen, welche deutlich auseinander liefen. Ein Zusammenführen der Versionen war praktisch kaum mehr möglich. Also wurde sich dafür entschieden, den Konsolenzweig als Basis für die Umstrukturierungen zu nehmen. Da das Konsolenteam zu diesem Zeitpunkt noch mit der Fertigstellung von „Sacred 2“ beschäftigt war, fehlte wichtiges Wissen bei der Integration der Code-Pfade.
Insgesamt kommt man auf 3179 mal "#ifdef CONSOLE_IMPL" im Projekt. Die Probleme und Kosten, welche dieser Umstand mit sich bringt sind kaum mehr zu beziffern.

Kategorisierung

Als Smell nach [Fowler] sind auszumachen:

- **Duplizierter Code**
- **Divergierende Änderungen**

Entwurf

Aus diesen Erkenntnissen ergibt sich die ohnehin notwendige Entscheidung, plattformspezifische Abschnitte so elementar wie möglich zu gestalten. Dies würde allerdings zu extrem verschachteltem Code führen, würde man dies inline lösen. Übersichtlicher, aber trotzdem elementar, lässt sich dies über Interfaces lösen, welche die nötigen Funktionaltäten für alle Plattformen transparent nach aussen darstellen.

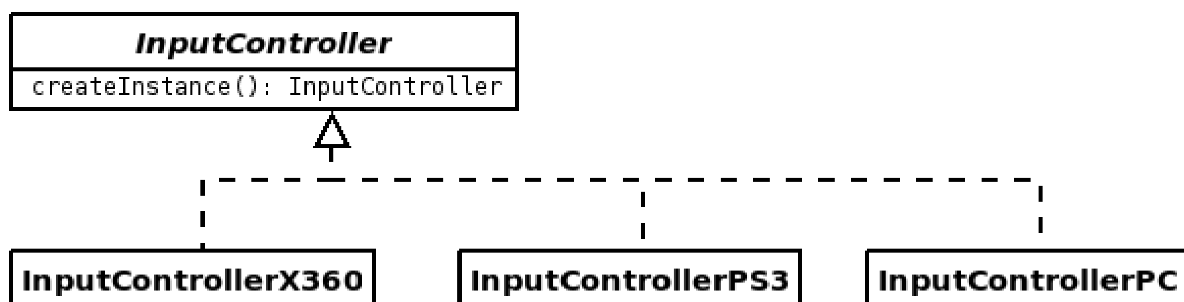

Diagramm 36: Beispiel Schnittstelle für plattformabhängigen Code

Was wie in folgendem Beispiel umgesetzt werden kann:

```
6.class InputController; // the abstract interface
7.class InputControllerX360 : public InputController; // implementation for X360
8.class InputControllerPS3 : public InputController; // implementation for PS3
9.class InputControllerPC : public InputController; // implementation for PC
10.
11.InputController*
12.InputController::createInstance()
13.{
14. switch( global.getPlatform() )
```

```
15. {
16.  case X360 : return new InputControllerX360();
17.  case PC :   return new InputControllerPC();
18. }
19. //PANIC!!
20. log.sendError( “missing implementation of InputController” );
21.}
22.
23.InputController* inputController = InputController::createInstance();
```

Am Nutzungsort gibt es danach nur eine Kopplung zum Interface. Dass dieses auf Plattformen spezialisiert ist, bleibt verborgen und kann auch später problemlos rückgebaut werden. Auch können Überschneidungen der Plattformanforderungen übersichtlicher umgesetzt werden.
Der Fakt, dass immer nur eine Implementierung benötigt wird, legt eine Umsetzung mit generativen Techniken (Template-Programmierung) nahe. Eine solche Lösung ermöglicht es dem Compiler, ungenutzen plattformabhängigen Code bereits zur Kompilierungszeit zu verwerfen. Dieser Ansatz ist sicher der technisch naheliegendere, allerdings erfordert er auch tieferes Verständnis mit Templates, da er bei falscher Anwendung recht destruktiv wirken kann (explodierende Kompilierzeiten und Codegröße).
Die Interface-Technik wurde bereits in Teilen der Konsolenimplementierung verwendet, allerdings nicht konsequent im ganzen Projekt durchgesetzt. Da mit ihr bereits Erfahrungen gesammelt wurden, und die Arbeit mit Klassen breiter bekannt ist, würde ich diese der Template-Programmierung vorziehen. Diese kann später eine funktionierende Interface-Implementierung ergänzen/ersetzen.

Umsetzung

1. Plattformspezifische Codeblöcke als “Methode extrahieren”.

2. Verwandte Methoden in einer Klasse bündeln

3. Für die extrahierten Blöcke “Bedingten Ausdruck durch Polymorphismus ersetzen” anwenden, um dadurch eine abstrakte Schnittstelle mit den plattformspezifischen Implementierungen zu erhalten.

Aufwandsabschätzung

Bei einigen Komponenten wird die Umstrukturierung einer Neuimplementierung gleichkommen, da dort plattformspezifischer Programmcode so durcheinander steht. Teilweise unterscheiden sich die Plattformen sogar in den Schnittstellen der Komponenten.

Die Umstrukturierung braucht alleine schon wegen den unterschiedlichen Entwicklungsumgebungen der Plattformen zusätzlich Zeit und wird nur in einem separaten Entwicklungszweig zu bewerkstelligen sein.

Bewertung

Der Aufwand ist hoch, aber die Umstrukturierung zwingend notwendig. Mit der Hardwareabstraktionsschicht "ORC" besitzt das Projekt bereits einen Lösungsansatz, auf den dafuer zurückgegriffen werden sollte.

4.8 Harte Typenumwandlungen

Analyse

Zahlreiche Umstände, vor allem queryStatus, benötigen für ihre Funktion harte Typenumwandlungen. Diese finden sich als reinterpret_cast, static_cast und C-Style-Casts zuhauf im Projekt.

reinterpret_cast	457
static_cast	4091
C-Style Cast	unbekannt

Tabelle 11: Anzahl von Typenumwandlungen im Ausgangszustand des Projektes

Ein gewisser Teil lässt sich sicher auf den Umstand der hardwarenahen Programmierung zurückführen. Der Großteil beruht allerdings eher auf vermeidbaren Designentscheidungen (siehe Parameter-Übergaben). Aus diesem Grund werden sehr häufig Typenumwandlungen durchgeführt, um vom übergebenen Basistyp auf den eigentlich benötigten Typ zu kommen. Dynamische Typenumwandlungen von C++ wurden aus Leistungsgründen deaktiviert (MSVC implementiert dynamischen Typenvergleich über kaskadierende Zeichenketten-Vergleiche). Statt dessen werden an diesen Stellen harte Typenumwandlungen verwendet, die auf dem unumstößlichen Vertrauen in fehlerfreies Design fußen. Ergebnis sind nicht selten schwer zu findende Abstürze, die sich meist als pur virtuelle Aufrufe äußern.

Kategorisierung

Auch hierbei handelt sich sich um eine handwerkliche Voraussetzung, die nicht erfüllt wurde. Harte Typenumwandlungen auf dieser Ebene, weg von der Hardware, sind kein probates Mittel der Implementierung.

Entwurf

C-Style-Casts sollten komplett durch C++ artige Casts ersetzt werden. Zum einen, da diese leichter analysierbar sind und eine gewisse Qualifizierung erlauben (reinterpret, static, dynamic, const) und einen gewissen Teil an Fehlerquellen erkennbar machen [CAST02]. Zum anderen da sie selbst bei richtiger Benutzung zu Laufzeitfehlern führen können [CAST01].

Von einer bereits in Betracht gezogenen eigenen projektweiten RTTI-Implementierung ist abzuraten. Diese ist nicht transparent umsetzbar und erfordert bei jeder Klassendefinition, oder Änderung dieser, Anpassungen aller abhängigen Klassen. Unachtsamkeiten führen dabei zu einem undefinierbaren Verhalten.

Mit dem Ziel, Typenumwandlungen möglichst komplett zu vermeiden, aber der Gewissheit, dass dies ein sehr umfangreiches Unterfangen ist, bleibt nur ein Kompromiss als vorläufige Lösung. Da RTTI in der Debug-Version angeschaltet ist, sollte zumindest dort die Laufzeitüberprüfung per dynamic_cast angeschaltet sein. Auch in der Release-Version ist die RTTI anzuraten. Im Master lässt sich dies aber nicht durchsetzen, da dort das massive Benutzen von dynamischen Typenumwandlungen zu nicht vertretbaren Leistungseinbußen führen würde.

Dafür bietet sich die folgende in [CAST01] beschriebene Zwischenlösung an:

```
1.template<class To, class From>
```

```
2.To checked_cast( From* from )
3.{
4. assert( dynamic_cast<To>(from) == static_cast<To>(from) && "checked_cast failed" );
5. return static_cast<To>( from);
6.}
```

Auf lange Sicht ist die Benutzung aller Arten von Typenumwandlungen minimal zu halten und/oder zumindest so zu kapseln und abzusichern, dass eine Falschverwendung möglichst schwer gemacht wird (vergleiche dazu auch [CAST02]).

Umsetzung

1. C-Style-Casts ausnahmslos durch entsprechende C++ artige Casts ersetzen. Ausnahme sind dynamic_casts, welche durch checked_cast ersetzt werden können.

2. Verbliebene Casts so weit wie möglich zurückbauen und Neuimplementierungen ohne derartige Typenumwandlungen umsetzen.

3.

Aufwandsabschätzung

Der Aufwand ist praktisch nicht abzuschätzen, da man automatisiert vermutlich erst auf Compiler-Ebene zuverlässige Aussagen treffen kann, was wirklich ein Cast ist. C-Style-Casts sind nur sehr schwer ohne den Kontext identifizierbar.

Von Vorteil ist die geringe Abhängigkeit. Da es in dieser Umstrukturierung nur um die Typenumwandlung und nicht um die Typendeklaration geht (siehe oben), sind kaum Abhängigkeiten vorhanden. Ist eine Stelle identifiziert, so kann man sie nach dem Feststellen des richtigen Cast-Typs problemlos ändern. Damit ist die Umstrukturierung problemlos parallel zur Entwicklung durchzuführen.

Bewertung

Eine Lösung für das Cast-Problem ist essentiel für eine funktionierende Architektur. „Nur ein entfernter Cast ist ein guter Cast“. Damit die Typenumwandlungen kontrollierbar bleiben, ist der Verzicht auf C-Style-Casts unumgänglich.

Ein kompletter Umbau des Projektes auf C++ artige Casts ist praktisch nicht durchführbar. Daher wird man auch dieses Problem nur im Zusammenhang mit dem Review einer Komponente angehen können, damit die Problemstellung überschaubar bleibt.

4.9 Bitflags → Klassen

Analyse

Flags werden fast ausschließlich in Bits von Unsigneds gespeichert. Für den Zugriff werden Bitoperationen genutzt. Diese sind fehleranfällig und schwer zu lesen. Compiler und Analysewerkzeuge tun sich ebenso schwer mit Bit-Operationen. Auch kam es schon vor, dass solche Variablen mit 8Bit serialisiert wurden, weil dies bis dahin die nötige Flag-Anzahl abgedeckt hat. Interessant wird so etwas dann erst, wenn die Anzahl der Flags die 8 übersteigt. Programmieren mit Bitoperationen verlangt bei der Benutzung immer Hintergrundwissen vom Programmierer, welches ihm die Datenstruktur nicht geben kann. Ohne dieses Hintergrundwissen kann man die Datenstruktur nicht nutzen. Zudem sollte klar sein, dass bei einem plattformübergreifenden Projekt solche Erweiterungen auf Bit-Ebene nur dort

angewendet werden dürfen, wo sie nicht die Programminstanz verlassen, andernfalls ist eine Serialisierung einzuplanen.

Kategorisierung

Als Smell nach [Fowler] sind auszumachen:

- **Neigung zu elementaren Datentypen**: Nutzung von nichtssagenden und nicht erweiterbaren elementaren Datentypen

Entwurf

C++ bietet in seinem Sprachkonzept ausreichende Möglichkeiten, um Flags für den Compiler erkennbar zu implementieren.

```
1.class BitField
2.{
3.   public:
4.        bool flagA : 1;
5.        bool flagB : 1;
6.} bitField;
7.
8.bitField.flagA = true
9.write( &bitField, sizeof( bitField ) );
```

Diese erlauben e,s ohne Hintergrundwissen an jeder Stelle mit dem Code zu arbeiten. Will man ganz korrekt sein, kann man bereits an dieser Stelle die Klassenmember kapseln. Dabei wird in den meisten Fällen allerdings der Aufwand den späteren Nutzen deutlich überschreiten. Wird dies später notwendig, kann man dies bei korrekter Benutzung ohne großen Mehraufwand über „Eigenes Feld kapseln“ nachholen. Eine Lösung mit Bitfields aus der STL wäre ebenfalls denkbar, jedoch nicht portabler. Um plattformunabhängig zu bleiben, empfiehlt es sich, Serialisierungsmethoden zu implementieren. Diese verschieben hardwaerenahe Programmierung per Bitoperatoren an eine definierte Stelle.

Die Bitfieldoperatoren sind nur ein Hinweis für den Compiler, wie er die Speicherbelegung optimieren kann. Dies geschieht besonders beim Mix von verschiedenen Datentypen nicht optimal. Ist jedoch eine optimale Belegung zwingend, ist anzuraten, die Klasse mit den oben erwähnten Zugriffsmethoden auszustatten und die Speicherung per Bitoperationen in dieser Klasse zu kapseln. Da es sich um einen „Plain Old Datatype“ handelt, wird dies genauso viel Speicherplatz belegen, als wenn man die Daten weiterhin ungekapselt in einem Unsigned hält und bearbeitet.

Umsetzung

Die Maskendefinition wird meistens über eine Enumeration gelöst. Diese kann direkt durch eine Klassendefinition ersetzt werden.

```
10.enum eBitmask
11.{
12.  EBM_FLAGA = 1 << 0,
13.  EBM_FLAGB = 1 << 1
14.};
```

wird direkt überfuhrt in:

```
15.class BitField
16.{
```

```
17.  public:
18.       bool flagA : 1;
19.       bool flagB : 1;
20.  public:
21.       void write( FileMem _dest ) const;
22.       void read( const FileMem _source );
23.};
```

Nächster Schritt ist das Umdeklarieren der Variable, die bisher die Flags gehalten hat, in den neu definierten Klassentyp. Ist die Variable gefunden, wird sie nicht einfach direkt umdeklariert, da wir uns auf den Compiler als „TODO“- Unterstützung verlassen wollen. Da dieser Bitoperationen auf der Klasse allerdings meist nicht als typverletzend erkennt, müssen wir anders vorgehen. Neben der Umdeklaration wird auch der Variablenname bei der Deklaration umbenannt (Dies kann ein Übergangsname sein, den man später mit Hilfe der IDE in den Ursprungsnamen refactorisieren kann).

Das Arbeiten mit Bitflags lässt sich eigentlich in den meisten Fällen direkt auf die neuen Bitfield Zugriffe abbilden.

```
24.oldBitField &= EBM_FLAGA; // wird zu
25.newBitField.flagA = true;
26.
27.oldBitField ~= EBM_FLAGA; // wird zu
28.newBitField.flagB = false;
29.
30.if( newBitField.flagA ) // wird zu
31.if( oldBitField & EBM_FLAGA )
```

Da Bitfelder im Speicher compilerspezifisch abgelegt werden dürfen, sollte bei plattformübergreifender Programmierung nicht vergessen werden, eine eigene Serialisierung einzuführen.

```
32.dest.write32( oldBitField ); // wird zu
33.newBitField.write( dest );
```

Aufwandsabschätzung

Der Vorgang enthält das Anpassen einer Typendeklaration. Der Aufwand dafür wurde im Kapitel zur Typisierung abgeschätzt. Zusätzlich kommt der Implementierungsaufwand dazu. Dieser hängt stark vom Umfang und der Benutzung des Bitfeldes ab. Jede Bitoperation muss durch einen passenden Methodenaufruf ersetzt werden. Bei komplexen und kryptischen Operationen ist Vorsicht geboten. Detailierte Tests der neuen Strukturen sollten danach nicht vernachlässigt werden.

Bewertung

Stabile und gut getestete Systeme müssen nicht angepasst werden. Sind die Flags jedoch nicht gekapselt und Operationen können und werden extern direkt auf der Flag-Variable vorgenommen, sollte man direkt die Kapselung in einem Bitfield angehen.

4.10 Singletons als globale Strukturen

Analyse

Das Entwurfsmuster „Singleton“ als globale Struktur zu missbrauchen ist eine Gefahr, auf welche stets hingewiesen wird.

Anzahl Singletonableitungen	134

Tabelle 12: Anzahl Singletonableitungen im Ausgangszustand

Betrachtet man die obige Zahl, so liegt die Vermutung nahe, dass genau dies im zu betrachtenden Projekt passiert ist.

Bei der Betrachtung der Singletons fällt die unterschiedliche Aufgabe der Strukturen auf. Einige sind gemischte Daten und Funktionsstrukturen, die über das Singleton einzigartig und global gemacht werden (siehe TeamMgr), weiterhin finden sich reine Funktionscontainer, bei denen das Singleton genutzt wurde, um global sichtbar zu sein (siehe LokaMgr). Letztere sind besser als statische Funktionen in Klassen oder Namensräume zu kapseln.

Dies erspart eine Indirektionsstufe und vereinfacht später weitere Umstrukturierungen.

Kategorisierung

Singletons dienen hier als Ersatz für globale Variablen. Durch die stetige Zugreifbarkeit kommt es zu einer hohen Kopplung verschiedenster Systeme. Die Einfachheit, auf Singletons zuzugreifen, motivierte die Entwickler dazu, Funkionalitäten gleich dort zu implementieren und führt zu Götterklassen.

Der globale Zugriff ermuntert weiterhin zum Umgehen von Schnittstellen.

Entwurf

Folgend einige Beispiele im Code, die dies unterstreichen, jeweils mit einem prototypischen Alternativvorschlag.

```
1.Var = cTypeMgr::Instance().isCreature( ent->getType() );
```

Die Entity-Typ wird darauf ueberprueft ob er eine Kreatur ist. So wie man die Situation im vorangehenden Satz formuliert, koennte man sie auch implementieren. Die Kopplung zum TypeMgr wird dadurch an dieser Stelle ueberfluessig und reduziert sich auf die "Type"-Implementierung.

```
2.var = ent->getType().isCreature();
```

```
3.cKernel::Instance().pushEvent( &evt );
```

Ein Event wird dem Kernel zur Abarbeitung übergeben. Wenn es nur einen Kernel gibt, findet der Event sicherlich auch selber dorthin. Der Kernel muss bei der Event-Benutzung nicht bekannt sein.

```
4.evt.pushEvent();
```

```
5.var = cLokaMgr::Instance().getText("QUEST_MSG_ITEMS_RECEIVED");
```

Anhand einer ID wird ein lokalisierter statischer Text abgefragt. Ob hier wirklich ein Singleton benötigt wird ist zumindest diskussionswürdig. Eine statische Implementierung ist zumindest gleichwertig.

```
6.var = cLokaMgr::getText("QUEST_MSG_ITEMS_RECEIVED");
```

```
7.U32 ref;
8.Creature* creature = (Creature*)cObjectMgr::Instance().getEntity( ref );
9.if( creature )
```

Eine solche Kapselung kann oft auch in Kombination mit einer strikteren Typisierung erfolgen. Würde obiges Beispiel zu einem undefinierteu Verhalten führen, falls die Instanz nicht wirklich eine Kreatur ist, könnte dies mit folgendem expliziten Syntax erkannt werden. Das Ergebnis ist zudem deutlich besser zu lesen und benötigt keine Kenntnis des ObjectManagers.

```
10.TypedRef ref;
11.Creature* creature = ref.getCreature();
12.if( creature )
```

<u>Umsetzung</u>

Sobald ein Kriterium mit einer angegebenen Antwort zutrifft, kann die Iteration beendet werden und zum nächsten Schritt übergegangen werden.

Für jede Methode im Singleton folgende 2 Kriterien kontrollieren:

1. Bekommt der Singleton-Aufruf Parameter übergeben?

Ja – Es bietet sich an, die Nutzung des Singleton in die Klasse des übergebenen Parameterobjektes zu verschieben. Damit beschränkt man die Kopplung des Singleton mit dieser Klasse. („Methode verschieben“ [Fowler])

2. Ist die aufgerufene Methode abhängig von Daten der Singleton-Klasse?

Nein – Methode auf statisch setzen. Gegebenenfalls an eine bessere, geeignete Stelle bewegen. (siehe Kapitel zu statischen Methoden)

Nach den Methoden die Kriterien für den Singleton überprüfen:

3. Hält der Singleton Instanzvariablen?

Nein – Die Struktur ist damit ein reiner Funktionscontainer. Damit können alle

enthaltenen Funktionen unter Einbezug der Architektur frei bewegt werden. („Methode verschieben" [Fowler])

4. Sind die vom Singleton gehaltenen Instanzvariablen dynamisch?

Nein – Der Singleton enthält keine Daten oder nur Daten, die einmal initialisiert und danach nicht mehr geändert werden. Dann lässt sich der Singleton einfach in eine statische Klasse überführen. Klassenmember und Methoden werden als statisch definiert und zu einem global einmaligen Punkt, wie es bei der vorliegenden Singleton-Verwendung bereits der Fall ist, initialisiert. Die Ableitung vom Singleton entfällt logischerweise.

Aufwandsabschätzung

Die Umstrukturierung lässt sich gut in einfache Teilaktionen, wie oben beschrieben, zerlegen. Nach der korrekten Diagnose können diese Änderungen auch von weniger erfahrenen Entwicklern vorgenommen werden. Durch die gute Teilbarkeit ist dies auch gut parallel zur Entwicklung möglich.

Bewertung

Singletons in einer derartigen Form sind nicht nötig, sondern destabilisieren die Architektur. Ihre Umstellung ist vergleichsweise abschätzbar und sollte daher durchgeführt werden. Man erhält ein System mit deutlich klareren Schnittstellen.

4.11 Baumartige Vererbungshierarchien

Analyse

Gemeinsame Schnittstellen werden über die Superklasse geerbt. Da so gut wie immer nur von einer einzigen Klasse geerbt wird, entsteht eine Baumstruktur in der Vererbung. Gemeinsame Schnittstellen wandern so in die erste gemeinsame Oberklasse. Diese halten dann gemeinsame Funktionalität und Daten. Dies ergibt unüberschaubare Götterklassen und Daten beanspruchen auch Speicher, bei Objekten, welche nicht diese Schnittstellen benötigen.

Alle im Spiel befindlichen dynamischen Objekte leiten sich direkt von cEntity ab. Dabei ist eine streng baumartige Struktur entstanden. Interface-Klassen wurden praktisch gar nicht genutzt.

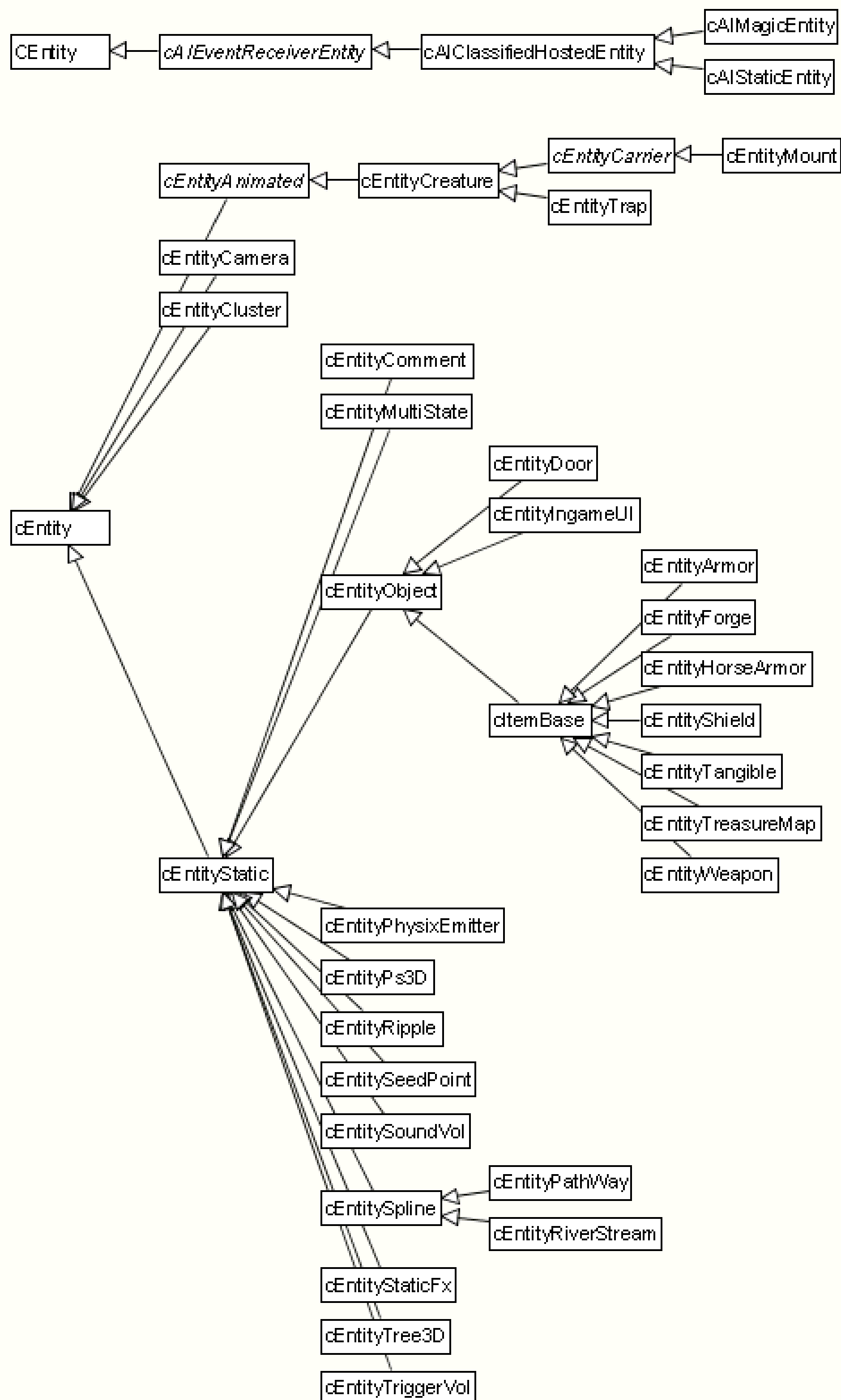

Diagramm 37: Vererbungsbaum für cEntity in der Ausgangsarchitektur

Als Ergebnis dessen sind gemeinsam genutzte Attribute und Methoden in die erste gemeinsame Oberklasse gewandert. Am besten lässt sich dies an den beiden Knoten cEntity und cEntityCreature beschreiben.

cEntity

Methoden	121
Öffentliche Methoden	113
Öffentliche Methoden (Get/Set ausgenommen)	28
Ungenutzte Methoden	92
Attribute	21
Ungenutzte Attribute	11
Eingehende Referenzen auf die Klasse	175
Ausgehende Referenzen der Klasse	16

Tabelle 13: Ausgewählte Metriken für cEntity im Ausgangszustand

Methoden	323
Oeffentliche Methoden	144
Oeffentliche Methoden (Get/Set ausgenommen)	50
Ungenutzte Methoden	307
Attribute	57
Ungenutzte Attribute	36
Eingehende Referenzen auf die Klasse	89
Ausgehende Referenzen der Klasse	65

Tabelle 14: Ausgewählte Metriken für cEntityCreature im Ausgangszustand

Betrachtet man die Werte für ungenutzte Methoden denkt, man eher, die Klassen bestehen zum Großteil aus totem Code. Der Umstand ist allerdings recht schnell erklärt. Private Aufrufe der Methoden werden nicht gezählt. Aufrufe über queryStatus (siehe OCMD) werden nicht als solche erkannt. Diese werden erst in der Klasse zu einem analysierbaren Methodenaufruf umgelenkt. Was einen erst aufatmen lässt, macht die Situation bei genauer Betrachtung nur noch schärfer. Viel Funktionalität, die eigentlich einer Methode gleich kommt, wurde in OCMDs verpackt. Genaue Zahlen würde man vermutlich nur bekommen, wenn man manuell den Code analysiert. Per Textanalyse kann man jedoch von circa 300 genutzten OCMDs ausgehen. Wieviele davon die „ungenutzten Methoden“ aufrufen oder nur aus eigenem Code bestehen, ist auch eine Sache, die sich nur manuell überprüfen lässt. Auch die anderen angeführten Werte sind daher nur ein erstes Indiz.

Ungeachtet dieser Tatsachen ist allein schon die Größe der analysierten Methoden ein deutliches Zeichen. Legt man die in [RefGro] zitierte 30er-Daumenregel zu Grunde, ist diese bei den Methoden in cEntityCreature schon um mehr als das Zehnfache überschritten.

Kategorisierung

Als Smell nach [Fowler] sind auszumachen:

- **Große Klasse**

Klassen wie cEntity und cEntityCreature sammeln enorme Funktionalitäten, welche eine Gemeinsamkeit ihrer Unterklassen sind.

Qualitätsindikator **„Gottklasse (Methode)“**

Qualitätseigenschaft	Relevanz
Analysierbarkeit	75%
Modifizierbarkeit	50%
Stabilität	0%
Prüfbarkeit	75%
Austauschbarkeit	75%
Zeitverhalten	0%
Verbrauchsverhalten	0%
Wirtschaftlichkeitsfaktor	**Einschätzung**
Kostenzuwachs	75%
Unmittelbarkeit	75%

Tabelle 15: "Gottklasse (Methode)" nach [CoQualMan]

Entwurf

Klassen erben geteilte Schnittstellen nicht allein über ihre Superklasse, sondern teilen sich Schnittstellen über pur abstrakte Interfaces, von denen sie direkt erben.

Umsetzung

Gemeinsame Anforderungen von Objekten identifizieren. Diese aus den Oberklassen entfernen und in Schnittstellen ziehen oder über Komposition einbinden.

Aufwandsabschätzung

Das Neumodellieren einer solch großen Vererbungshierarchie, welche zudem hohe Kopplungen zum restlichen System hat, ist dem Aufwand einer kompletten Neuplanung gleichzusetzen, wenn er einen solchen nicht sogar übertrifft. Es ist damit zu kalkulieren, jede Klasse in dieser Vererbungshierarchie einem kompletten Review zu unterziehen. Dies impliziert Änderungen an allen abhängigen Systemen.

Bewertung

Der Aufwand lässt eine komplette Änderung in einem Anlauf nicht zu.

Es ist zu empfehlen, für die Klassenhierarchie eine Neuplanung vorzunehmen, ohne

dabei ins Detail zu gehen.

Diese Neuplanung sollte bei allen Umbaumaßnahmen und Neuplanung als Referenz dienen. Ziel ist es, das System an einer weiteren „Degenerierung“ zu hindern und schrittweise in eine Lage zu bringen, welche einen Umbau ganzer Teilsysteme und schlussendlich des ganzen Systems erlaubt.

5 Praxisbeispiel

Analyse:

ObjectMgr und TypeMgr wurden bereits in anderen Zusammenhängen erwähnt. Beim ObjectMgr handelt es sich um eine Kontrollstruktur für dynamische Objekte, welche diese erstellen, zerstören und anhand einer ID referenzieren können. Der TypeMgr gibt anhand einer übergebenen Typ-ID Informationen zurück. Folgend zwei Anwendungsfälle für diese beiden Strukturen.

Ablauf Typenüberprüfung:

1. Zur Laufzeit initialisiert der TypeMgr alle möglichen Informationen und Relationen anhand der Typ-ID in Datenstrukturen. Dies erfolgt einmal beim Programmstart.
2. Es entsteht der Bedarf, eine Instanz anhand des Typs zu identifizieren.
3. Die Typ-ID führt jede Objektinstanz als Instanzvariable mit sich und wird gegebenenfalls ausgelesen.
4. Einer Prüfmethode wird die ID übergeben, welche daraus die gewünschten Schlüsse zurückgibt. Am häufigsten wird überprüft ob eine Instanz einem bestimmten Typ angehört (isItem, isCreature etc).

Ablauf Instanzerstellung:

1. Zur Laufzeit sammelt der ObjectMgr alle ihm bekannten Factorys in einer List.
2. Es entsteht der Bedarf, eine Instanz von einem bestimmten Typ zu erstellen.
3. ObjectMgr::create() wird die Typ-ID als U32 übergeben.
4. ObjectMgr sucht in seiner List die verantwortliche Factory.
5. Die Factory erzeugt das Objekt und setzt die Typ-ID in einer Instanzvariable des Objektes.

Ergänzend dazu ein Klassendiagramm der erläuterten Zusammenhänge:

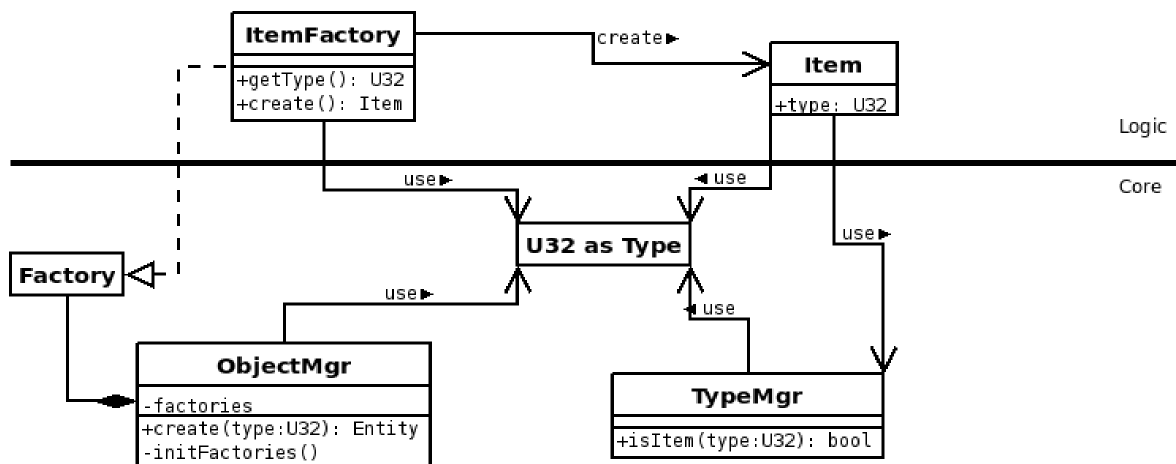

Diagramm 38: Ausgangszustand des Beispielsystems

Bereits anhand dieser beiden Fallbeispiele und des Diagramms lassen sich folgende Erkenntnisse erzielen:

– Zwischen den Komponenten bestehen unnötige Kopplungen.

- Die Factory muss vom Programmierer explizit beim ObjectMgr angemeldet werden.
- Überall, wo mit Typen gearbeitet wird, muss der TypeMgr bekannt sein. Dieser bringt seinerseits eine Kette von Abhängigkeiten mit (nicht im Diagramm vermerkt).

- Typen sind mit U32 nicht ausreichend typisiert.
  - Analysewerkzeuge erkennen keine Kopplungen zwischen den Systemen.
  - mangelnde Typensicherheit, Wartbarkeit und Erweiterbarkeit
- geringe Kohärenz für die betrachteten Fallbeispiele
  - ItemFactory, TypeMgr::isItem sind offensichtlich Item-spezifische Funktionen
  - TypeMgr stellt Funktionalitäten, die allein mit der Typ-ID verwendet werden.

Entwurf:

- Definition eines Datentypen Type
  - enthält Typ-ID
- Funktionalitäten der Factory werden als Klassenmethoden in das Objekt gezogen.
  - Objekt enthält als Klassenvariable eine Instanz der RegisterFactory. Diese wird mit Typ-ID und Referenz auf die create-Methode initialisiert. Dabei meldet sie sich implizit beim ObjektMgr an.
- Objektspezifische Methoden des TypeMgr werden als Klassenmethoden in das Objekt gezogen.

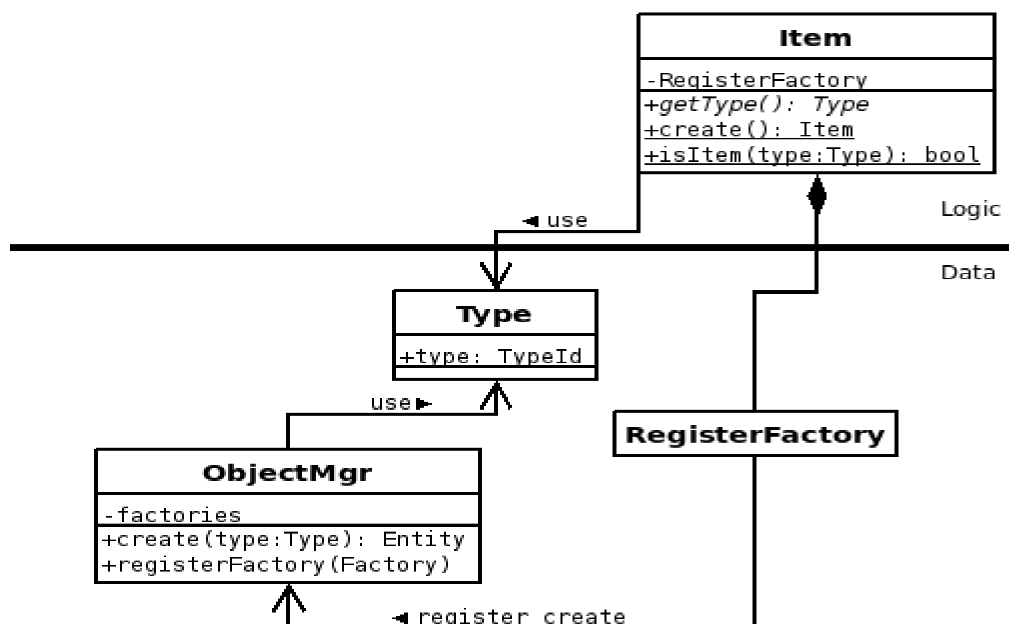

Diagramm 39: Soll-Zustand des Beispielsystems

Umsetzung:

1. Einführung von Type
2. TypeMgr-Funktionalität verschieben.
    1. Typspezifische in die Klasse „Type“ und objektspezifische in ObjektMgr
        1. siehe Kapitel zur Umstrukturierung von Singletons
3. ObjektMgr durch registerFactory Funktionalität ergänzen
    1. RegsiterFactory Hilfsklasse einführen
4. ObjektFactory-Funktionalität in Objektklasse (hier „Item“) verschieben
    1. „Klasse integrieren“ [Fowler]

Aufwand:

Der Aufwand umfasst die Typisierung von „Type“, welche im entsprechenden Kapitel bereits besprochen wurde. Alle weiteren Komponentenschnittstellen bleiben gleich oder werden nur verschoben.

Bewertung:

Factories als eigenständige Komponenten verschwinden, der TypeMgr wird um seinen eigentlichen Existenzgrund gebracht und ist so offen für eine komplette Entfernung. Als Ergebnis erhält man ein überschaubares System.

6 Bewertung

6.1 Schlussfolgerungen

Fast allevVorgeschlagenen Umstrukturierungen sind von der Komplexität recht simpel. Erst die Masse der jeweiligen Verstöße treibt den Aufwand in die Höhe. Diese hätte man verhindern können, wenn man die bekannten Probleme in einer früheren Phase ausgearbeitet hätte. So jedoch hatten sie Zeit sich zu einem Dogma bei den Programmierern zu entwickeln und sich tief und zahlreich in das System einzugraben.

Problemesituationen, wie zum Beispiel Singletons gehen schlichtweg auf das Missverständnis der Prinzipien zurück. Singletons wurden als Entwurfsmuster propagiert und hatten damit den Ruf, dass man etwas Überlegtes und Bewährtes benutzt. Die Anforderungen wurden nicht mehr überprüft und das Pattern wurde zu einem Antipattern.

Leider waren diese Problemmuster in der Analyse schwerer zu finden als erwartet, da Analysewerkzeuge mit der handwerklichen Umsetzung des Programmcodes nur sehr schlecht klar kamen. Funktionalitäten, wie queryStatus sind in diesem Zusammenhang besonders hervorzuheben.

Die als Lösungen erarbeiteten Entwürfe fordern nie einen kompletten Neuaufbau der Systeme, jedoch ist der Aufwand diesem teilweise ebenbürtig. Dieser war nur zu vertreten mit den verhinderten Fehlerquellen, die einige Problemmuster mit sich bringen.

6.2 Relation von Architekturhierarchie zur Teamhierarchie

Teamhierarchie und Softwarearchitektur sind nur konsequent durchzusetzen, wenn sie Hand in Hand gehen. Eine durchdachte Architektur ist schnell verwaschen, wenn es keine Verantwortlichkeiten gibt und jeder überall was entscheiden/ändern darf. Ebenso kann eine eingespielte Teamstruktur schnell darüber stolpern, dass es keine Zielvorgabe für die Architektur gibt. Mammutaufgaben landen dann bei Personen, welche eigentlich nur für eine Teilaufgabe zuständig sind. Teamleads dagegen bekommen von essentiellen Änderungen nichts mit, da sie versteckt von einem Subsystem aus, in kontrollierende Teile der Architektur eingreifen.

Mit den angedachten Architekturbausteinen wäre folgende Relation zwischen Teamstruktur und Architektur denkbar:

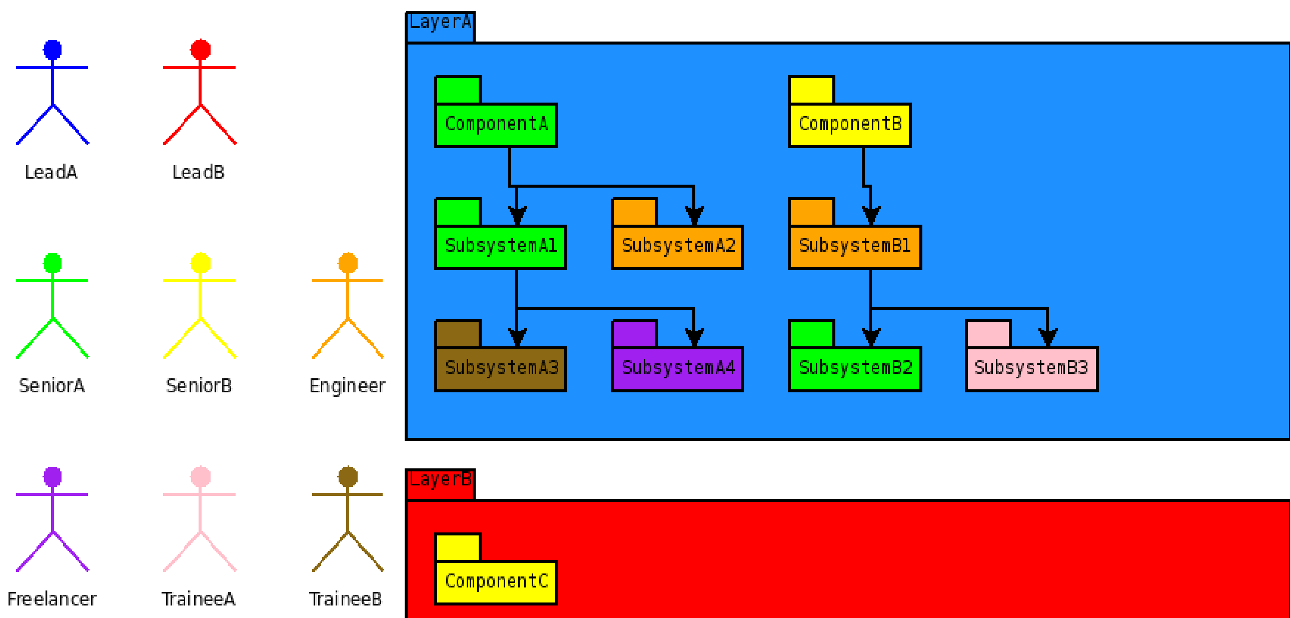

Diagramm 40: Architekturhierarchie bildet sich in der Teamhierarchie ab

Dargestellt ist ein Beispielteam und eine schematische Architektur. Die Farbcodes kennzeichnen mögliche Verantwortlichkeiten. Diese Orientieren sich an der Erfahrung und dem Wissen eines Teammitgliedes. Verantwortlichkeit schließt nicht aus, das man andere Teilsysteme kennt oder dessen Verantwortlichkeit teilt. Dies ermöglicht auch die einfache Integration von freien Mitarbeitern und Neulingen.

Abgegrenzte Module erlauben auch das unabhängige Arbeiten an diesen Systemen. Bisher bekam ein Programmierer die Aufgabe, eine Anwendungseigenschaft zu implementieren und musste dabei mehr oder minder an alle Systeme des Projektes. Sinnvoller wäre es, Aufgaben auf Schnittstellen und Garantien zu begrenzen, für das System, dessen Verantwortung man hat. Benötigt man von Subsystemen erweiterte Funktionalität, gibt man diese Anforderungen an den Verantwortlichen.

Verantwortlichkeiten können dabei jederzeit neu verteilt werden. Betroffene können dabei genauer abschätzen, wie hoch die Einarbeitungszeit ist, da neue Systemteile bekannt und eingrenzbar sind.

6.3 Erfahrungen in der Vermittlung

6.3.1 Regeln und Gruppendynamik

Es hat sich gezeigt, dass Regeln nur akzeptiert werden, wenn ihre Funktion und die Begründung auch verstanden werden. Ist dies nicht der Fall, kommt es immer wieder zum Rückfall in die alten Vorgehensweisen („Weil wir das schon immer so machen.“).

Um einen Meinungs- und vor allem Wissensaustausch anzuregen, wurde als Kommunikationsplattform ein Intranet gestütztes Forum eingeführt. Dies war nötig, da es zwei Standorte gab, welche bisher nur im engen Rahmen kommunizierten (E-Mails und Telefonate, vor allem zwischen Teamleads). Dieses Forum wurden sehr gut angenommen. Zahlreiche Probleme wurden angesprochen und rege diskutiert. Oft bildete sich eine an sich konsensfähige Lösung heraus. Diese wurden allerdings nie in verbindliche Regeln umgesetzt. Die Technische Leitung hielt dies nicht für möglich, da man immer irgendjemanden verstimmt hätte. So lies sich die oben erwähnte Regelung

für Casts nicht durchsetzen, da wenigen der Aufwand, C++ artige Typenumwandlungen zu schreiben, zu hoch war. Es waren emotionale Gründe, die über die Qualität des Produktes gestellt wurden.

6.3.2 UML gegen Prototypen

UML ist eine bewährte Methode, um Architekturen und Abläufe darzustellen. Allerdings ist in der Praxis heute immer noch nicht jeder in der Lage, UML korrekt zu interpretieren. Selbst dann noch ist nicht jeder Entwickler in der Lage eine so umschriebene Architektur korrekt zu implementieren oder die Abläufe in einem solchen System zu verstehen. Das hat nicht unbedingt etwas mit mangelndem Wissen zu tun.

Prototypen sind hier ein mögliches Werkzeug der Vermittlung. Dieser Prototyp wird von manchen Vorgehensmodellen in der Softwareentwicklung sowieso verlangt und bedarf daher keinen Mehraufwand. Anhand dessen kann Programmierern in ihrer „Muttersprache“ das Funktionieren eines Systems erklärt werden. Auch kann man Implementierungsregeln am praktischen Beispiel erläutern.

7 Zusammenfassung und Ausblick

Rückblickend kann man feststellen, dass einige Probleme ihre Brisanz erst mit der Zeit bekommen haben. Über einen langen Zeitraum wurden Praktiken eingesetzt, obwohl immer klarer wurde, welche Nachteile sie mit sich bringen. Jedoch wurde bis zum Ende, und gerade unter dem herrschenden Zeitdruck, das Motto ausgegeben: „Augen zu und durch“.

Als Konsequenz muss man feststellen, dass unter anderem diese Einstellung das Projekt und damit die Firma ins Wanken gebracht hat. Das Spiel „Sacred 2“ konnte im Herbst 2008 für den PC veröffentlicht werden. Gefolgt von zahlreichen Patches, welche das Produkt einigermaßen stabil machen konnten. Die Version für die Konsolen verzögerte sich um mehr als ein halbes Jahr. Kämpfte man zu Beginn nur mit der Inkompatibilität zum PC-Quelltext, kam später noch die Qualitätssicherung von Sony und Microsoft dazu.

Mangels Struktur und Konventionen in der Architektur war das Debuggen von Fehlern immer wieder ein Start bei Null. Änderungen brachten alte Fehler wieder hervor, das Vertrauen in die eigene Arbeit sank mit jeder Woche.

Sein Finale fand das Stück in der Insolvenz und Abwicklung der Firma. Die Verzögerungen hatten die Kosten derartig in die Höhe getrieben, dass es keine fortführende Finanzierung mehr gab. Es ist Spekulation zu behaupten, dass mit einer besseren Planung und Struktur der Software alles anders hätte laufen können. Allerdings darf man in einer Industrie mit solchen Investitionssummen nicht den aktuellen Wissensstand ignorieren.

Wie in jeder jungen Industrie wird auch die Computerspieleindustrie mit der Zeit Erfahrungen und Erkenntnisse aus anderen Feldern aufnehmen und einbinden. Auch sie wird irgendwann erst mit dem „Hausbau“ beginnen, wenn Bauplan und Statik abgesegnet wurden. Auch wird nicht jeder Arbeiter den Grundriss ändern dürfen, so wie er es aus seinen alten Projekten her kennt.

Der Architekt wird die Konstruktion ausarbeiten und der Bauleiter die Resultate abnehmen.

Anhang:

Glossar

GFX - Graphical Effects – Grafische Effekte

Grafische Spielereien, welche die optische Umsetzung sehenswerter machen. Zum Beispiel: Partikel, Unschärfe, Glanz und Transparenzen.

GUI – Graphical User Interfaces

Die grafische Benutzeroberfläche. Dies umfasst Eingabemethoden und Anzeigen von Daten im Spiel und im Menü. Separat davon wird die Darstellung der Spielwelt betrachtet, welche nicht Teil der GUI ist.

MSVC – Microsoft Visual C

Die C/C++ Entwicklungsumgebung von Microsoft. Im Zusammenhang ist meist der Compiler gemeint.

NPC – Non Player Character

Von der künstlichen Intelligenz kontrollierte Spielfiguren. Dies können Gegner, Händler oder Figuren der erzählten Geschichte sein.

OCMD

Bezeichnet in diesem Dokument exemplarisch die Technik, welche durch queryStatus umgesetzt wurde. Eine Art Funktor mit untypisierten Parametern, welche manuell in einem switch-Block aufgelöst oder abgebildet werden.

Damit lassen sich über das Basisobjekt cEntity, welches in der Core-Schicht definiert ist, sämtliche Schnittstellen umgehen.

Dieser Mechanismus wurde zu einem Hauptinstrument bei der Implementierung.

POD – Plain Old Datatype

Datentypen, dessen interne Darstellung rein die definierten Daten enthält. Sie entsprechen elementaren Variablentypen oder structs aus C Zeiten. Sie enthalten zum Beispiel keinen Zeiger auf eine vTable. Oft impliziert POD auch die Öffentlichkeit aller enthaltenen Instanzvariablen.

QueryStatus

siehe OCMD

SotoArc

Ist ein kommerzielles Werkzeug zur statischen Code-Analyse von Softwarearchitektur. Es visualisiert die statische Struktur eines in Java, C# oder C++ geschriebenen Codes. Es konzentriert sich dabei auf die Grobgranulare Strukturebene. Es ist Teil der SotoGraph Produktfamilie.

Abbildungsverzeichnis

Diagramm des genutzten Planungsmodels....7
Schichtenmodell der Alten Soll-Architektur....15
Sequenz einer Ereignisabarbeitung....19
Kopplungen im Schichtenmodell in der Ist-Architektur....21
Ereigniskomponente in der Ist-Architektur....22
Zielsetzung für Ziel-Architektur auf oberster Abstraktionsebene....23
Schichtenmodell der Ziel-Architektur....24
Model-View - Ansatz....24
Model-View-Presenter-Ansatz....25
Kopplungen im Schichtenmodell zur Ziel-Architektur im ersten Iterationsschritt 25
Kopplungen im Schichtenmodell zur Ziel-Architektur im zweiten Iterationsschritt 26
Kopplungen im Schichtenmodell zur Ziel-Architektur im dritten Iterationsschritt 27
Vorliegende Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager....28
Erwünschte Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager....28
Ereignis in der Ziel-Architektur....29
Erwünschte Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager unter Einbeziehung der Ziel-Architektur für Ereignisse....29
Hybridlösung der Ereignis-Komponente....31
Implementierungsregel für das strikte Schichtenmodell....32
Implementierungsregel für das Zusammenspiel von Ereignis und Manager mit alter Ereignisbasis....33
GUI relevante Kopplungen zwischen Game und Logic-Schicht in der Ist-Architektur....34
GUI relevante Kopplungen zwischen Game und Logic-Schicht in der Neuen Soll-Architektur....35
Einführung einer separaten Datenschicht - Ausgangssituation....37
Einführung einer separaten Datenschicht – Iteration 1....37
Einführung einer separaten Datenschicht – Iteration 2....38
Einführung einer separaten Datenschicht – Iteration 3....39
Einführung einer separaten Datenschicht – Endzustand....39
Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager in der Ausgangsimplementierung....40
Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager nach dem ersten Umstrukturierungsschritt....40
Klient-Server Trennung im Zusammenspiel von Ereignis und Manager nach dem zweiten Umstrukturierungsschritt....41
Klient-Server-Trennung im Zusammenspiel von Ereignis und Manager im Endzustand....42
Allgemeines Ereignis und abgeleitetes spezialisiertes Ereignis....42
Ereignisimplementierung in der Ausgangsimplementierung....43
Ereignisimplementierung in der hybriden Zwischenlösung....43
Ereignisimplementierung im Endzustand....44

Untypisierte Parameterübergaben im Beispiel..49
Beispiel Schnittstelle für plattformabhängigen Code.......................................56
Vererbungsbaum für cEntity in der Ausgangsarchitektur.................................65
Ausgangszustand des Beispielsystems..69
Soll-Zustand des Beispielsystems..70
Architekturhierarchie bildet sich in der Teamhierarchie ab..............................73

Tabellenverzeichnis

Tabelle 1: Render -> Logik..18
Tabelle 2: Logik -> Render..18
Tabelle 3: Core -> Render..18
Tabelle 4: Core -> Logik...18
Tabelle 5: OCMD-Statistiken im Ausgangszustand...46
Tabelle 6: "Gottmethode" nach [CoQualMan]...47
Tabelle 7: "Labyrinthmethode" nach [CoQualMan]..47
Tabelle 8: "tote Implementierung" nach [CoQualMan].....................................48
Tabelle 9: "allgemeine Parameter" nach [CoQualMan].....................................50
Tabelle 10: "simulierte Polymorphie" nach [CoQualMan].................................50
Tabelle 11: Anzahl von Typenumwandlungen im Ausgangszustand des Projektes 58
Tabelle 12: Anzahl Singletonableitungen im Ausgangszustand........................62
Tabelle 13: Ausgewählte Metriken für cEntity im Ausgangszustand.................66
Tabelle 14: Ausgewählte Metriken für cEntityCreature im Ausgangszustand....66
Tabelle 15: "Gottklasse (Methode)" nach [CoQualMan]...................................67

Literaturverzeichnis

[WEB01] Mark Androvich: GTA IV: Most expensive game ever developed?; 2008,

[SoAr] Vogel, Arnold, Chugthai, Ihler, Mehlig, Neumann, Voelter, Zdun: ;, 2005,3-8274-1534-9

[Szyperski] Szyperski, Clemens: ;Addison-Wesley, 1998,

[WEB02] Oliver Reeves: The Magic of Unity Builds; 2007, http://buffered.io/2007/12/10/the-magic-of-unity-builds/

[CoQualMan] Frank Simon, Olaf Seng, Thomas Mohaupt: Code-Quality-Management;dpunkt.verlag, 2006,3-89864-388-3

[RefInGroSoftw] Stefan Roock, Martin Lippert: Refactorings in großen Softwareprojekten;dpunkt.verlag, 2004,3-89864-207-0

[GoF] Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software;Addison-Wesley, 1995,

[Fowler] Martin Fowler: Refactoring;Addison/Wesley, 2005,978-3-8273-2278-4

[libsig] : libsigc++ Homepage; , http://libsigc.sourceforge.net/

[CAST02] Bjarne Stroustrup: Bjarne Stroustrup's C++ Style and Technique FAQ; 2008, http://www.research.att.com/~bs/bs_faq2.html#static-cast

[CAST01] Herb Sutter, Andrei Alexandrescu: C++ Coding Standards; 2007,