HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

HDLCSimulator

Diplomarbeit FS 2009

Refactoring und Ausbau des HDLC Simulators

Technischer Bericht

Status: Final

Datum: 01. Mai 2009

Student: Michael Koch (m1koch@hsr.ch)

Betreuer: Prof. Dr. Andreas Rinkel (andreas.rinkel@hsr.ch)

Inhaltsverzeichnis

1 Einleitung

Im Rahmen einer Diplomarbeit für das Institut für Internet-Technologien und -Anwendungen an der HSR wurde im Jahr 2007 ein Lernspiel entwickelt. Es soll einen Teil des OSI-Modells simulieren (1). Eine Kommunikationsseite wird dabei von der Applikation übernommen. Für die Abläufe auf der anderen Seite ist der Benutzer der Applikation zuständig. Die Simulation ist auf die Sicherungsschicht (Schicht zwei) des OSI-Modells beschränkt. Diese ist für die fehlerfreie Übertragung der Daten zuständig. Im Laufe der Simulation erhält man entweder von der eigene übergeordnete Schicht oder von seinem Kommunikationspartner verschiedene Pakete. Das Ziel des Lernspiels ist es, korrekt auf die eingehenden Pakete zu reagieren und somit eine funktionierende Kommunikation zu ermöglichen. Um dies zu erreichen kann man selber Pakete zusammenzustellen. Diese müssen anschliessend an den richtigen Ort gesendet werden. Entweder ist das Ziel die eigene übergeordnete Schicht oder der Kommunikationspartner. Die Applikation hat eine grafische Benutzeroberfläche. Es wurde bei der Erstellung darauf geachtet, dass möglichst viel Drag and Drop verwendet werden kann. Das Lernspiel ist in Level strukturiert. In jedem Level muss ein Text übertragen werden. Die Übertragung kann in beide Richtungen stattfinden. Falls der Text korrekt übermittelt wurde, wird der nächste, schwierigere Level gestartet. Das Lernspiel wurde entwickelt, um das Verständnis für die Vorgänge in der Sicherungsschicht zu verbessern und das Wissen zu festigen.

Um mit einer konkreten Implementierung arbeiten zu können, wurde HDLC als Schicht-Zwei-Protokoll verwendet. HDLC steht für High-Level Data Link Control (2). Aus diesem Grund trägt das Lernspiel den passenden Namen HDLCSimulator. Für eine genaue Beschreibung sei auf *Diplomarbeit HDLCSimulator* (3) verwiesen.

Bei der Benutzung des HDLCSimulator sind einige Probleme zu beobachten. Durch diese Probleme wird der Nutzen der ansonsten gut konzipierten Applikation erheblich beeinträchtigen. Das mit Abstand gravierendste Problem betrifft die Performance. Nach ein paar Minuten des Betriebs wird das Programm immer langsamer. Schlussendlich kommt es dann zum Stillstand und jegliche Interaktion wird verunmöglicht.

Ein weiteres Problem betrifft die Implementation des Protokolls. Der Anfang und das Ende eines HDLC-Rahmens wird jeweils mit einer bestimmten Bitfolge gekennzeichnet. Sie lautet „01111110“. Es könnte sein, dass innerhalb des Rahmens ebenfalls diese Bitfolge auftritt. Um eine Verwechslung mit einer Rahmenbegrenzung zu vermeiden, wird ein sogenanntes Bitstuffing durchgeführt. Dabei wird innerhalb des Rahmens jeweils nach fünf aufeinanderfolgenden „1“ eine „0“ eingeschoben. Auf der Gegenseite werden dann diese Nullen als erstes wieder entfernt. Im HDLCSimulator wurde das Bitstuffing mit einem Button realisiert. Drückt der Benutzer diesen Button mehrmals, führt dies zu veränderten Daten auf der Gegenseite. Dieser Umstand kann innerhalb des Levels nicht mehr korrigiert werden und ein Neustart ist nötig.

Schlussendlich bietet der HDLCSimulator noch einiges Verbesserungs-Potenzial was die Bedienung des Programms betrifft. So ist es zum Beispiel nicht möglich, die einzelnen Levels direkt anzuwählen. Man beginnt immer bei Level 1. Auch ein Neustart des aktuellen Levels ist nicht möglich. Desweiteren ist keine Hilfestellung verfügbar, falls der Benutzer mit einem Level Probleme hat und gar nicht mehr weiterkommt.

Das Ziel dieser Diplomarbeit ist es, die bestehenden Problem und Mängel zu beseitigen. Der HDLCSimulator soll nach dieser Überarbeitung dafür geeignet sein, im Unterricht als Lern-Unterstützung eingesetzt zu werden.

In dieser Diplomarbeit wird hauptsächlich Refactoring eingesetzt. Es ist wichtig, dass der Leser und der Autor dasselbe Verständnis dieser Technik haben. Damit dies sichergestellt werden kann, soll an dieser Stelle der Begriff Refactoring genau beschrieben werden.

Refactoring bezeichnet in der Informatik das Verbessern von bestehendem Programmcode. Dabei liegt der entscheidende Punkt darin, dass sich das Verhalten der Applikation für den Benutzer nicht verändert. Refactorings werden in der Regel in kleinen Schritten durchgeführt. Nach jedem Schritt ist die Applikation wieder in einem funktionierenden Zustand.

Das Ziel des Refactorings ist eine Verbesserung der Lesbarkeit, Wartbarkeit oder Erweiterbarkeit des Programmcodes. Desweiteren soll auch die Vermeidung von Redundanzen erreicht werden. Der Code wird klarer und übersichtlicher und der Zeitaufwand für die Einarbeitung verringert sich erheblich. Je besser der Programmcode geschrieben ist, umso geringer ist auch die Wahrscheinlichkeit, dass sich neue Fehler einschleichen, wenn am Code gearbeitet werden muss.

Die Lesbarkeit des Programmcodes kann dadurch verbessert werden, dass man irreführende oder einbuchstabige Variablennamen in beschreibende Namen umbenennt. Eine andere Möglichkeit wäre zum Beispiel das Extrahieren von Methoden. Falls in einer einzelnen Methode viele verschiedene Operationen implementiert sind, können diese ausgelagert werden. Dazu definiert man zuerst neu Methoden. Die einzelnen Operationen werden in diese neuen Methoden verschoben. Die ursprüngliche Methode müssen anschliessend noch mit den entsprechenden Methodeaufrufen ausgestattet werden, um die extrahierten Operationen auszuführen. Durch diese Massnahme wird neben der Lesbarkeit auch noch die Wartbarkeit erhöht. Muss der Entwickler eine Operation umschreiben, findet er die entsprechende Stelle im Programmcode viel leichter, da sie in einer eigenen Methode gekapselt ist. Ebenfalls eine Verbesserung der Wartbarkeit ist zu erreichen, indem von verschiedenen Klassen benötigt Methoden in eine separate Klasse ausgelagert werden. Somit können spätere Änderungen am Programmcode an einer einzigen Stelle getätigt werden und alle Klassen benutzen ab dann denselben Code.

Beim Durchführen eines Refactorings muss sehr sorgfältig vorgegangen werden. Der zugrundeliegende Programmcode gehört zu einer korrekt laufenden Applikation. Es ist von entscheidender Bedeutung, dass das Refactoring nicht die Ursache von neuen Fehlern bildet und das beobachtbare Verhalten der Applikation sich nicht verändert. Aus diesem Grund ist es wichtig, dass vor dem Refactoring gute Unit-Test implementiert werden. Dazu führt man Testklassen ein. Die darin enthaltenen Funktionen haben den Zweck, die Funktionalität der Applikation auf dessen Richtigkeit zu prüfen. Sind diese Tests erfolgreich, kann der erste Refactoring-Schritt ausgeführt werden. Nach Anschluss wird noch einmal der Unit-Test ausgeführt. Fällt das Resultat des Tests wieder positiv aus, hat das Refactoring die Funktionalität des Programms nicht verändert oder sogar beeinträchtigt. Dieser Vorgang wird wiederholt bis alle Refactoring-Schritte ausgeführt wurden. Voraussetzung ist natürlich, dass die Unit-Tests so programmiert sind, dass sie die gesamte Funktionalität des Programms abdecken. Ist dies nicht der Fall, könnte sich ein Fehler an einer ungeprüften Stelle im Programmcode einschleichen.

Dabei kann es darum gehen, die Performance einer Applikation zu verbessern, in dem zum Beispiel zeitaufwendige Programmabläufe effizienter programmiert werden.

2 Analyse

2.1 Analyse der Performance

b	Exceptions	Fehler	Erweiterungen

Wird der HDLCSimulator gestartet, so dauert es etwa zehn Minuten bis der Betrieb des Programms verlangsamt wird. Nach ungefähr 15 Minuten ist keine Manipulation mehr möglich. Das Programm ist völlig blockiert. Es spielt dabei keine Rolle, ob man versucht möglichst schnell in den Level aufzusteigen, gemütlich vor sich hin klickt oder gar keine Aktionen durchführt. Dies lässt darauf schliessen, dass sich irgendwo im Programm ein Speicherleck befindet. Das bedeutet, dass verwendeter Speicher nicht wieder freigegeben wird, sobald er nicht mehr benötigt wird. Daraus resultiert, dass der Heap immer grösser wird, bis schliesslich das obere Speicherplatz-Limit erreicht wird und kein weiterer Speicher mehr frei ist, um die gewünschten Aktionen auszuführen.

Um die genaue Ursache dieses Speicherlecks herauszufinden, wurde mit dem Performance- und Memory-Analysetool JProbe (4) gearbeitet. Dieses Tool wird parallel zum zu untersuchenden Programm gestartet. Während des Programmablaufs zeichnet es unter anderem den Speicherverbrauch der verschiedenen Objekte auf und erstellt entsprechende Statistiken.

Um die problematischen Objekte zu identifizieren und somit das Speicherleck im HDLCSimulator zu lokalisieren, wurde JProbe zusammen mit dem Lernspiel gestartet. Nach Programmstart wurde vermieden, irgendwelche Aktion im Simulator auszuführen. Wie oben beschrieben war die Software nach etwa 15 Minuten blockiert. Der folgende Screenshot zeigt das Ergebnis der Analyse zu diesem Zeitpunkt.

Filter Classes * [983 / 983]

Name	Use Case Count	Heap Count	Use Case Memory	Heap Memory	Dead Count	Dead Memory
Total	--	1'854'539	--	68'084'048	--	--
simCore.simMessage	--	456'581	--	21'915'888	--	--
simCore.simTimer	--	456'580	--	21'915'840	--	--
java.util.HashMap$Entry	--	458'166	--	10'995'984	--	--
java.lang.Long	--	456'593	--	7'305'488	--	--
java.util.HashMap$Entry [ ]	--	190	--	3'693'208	--	--
java.lang.Class	--	2'288	--	1'108'904	--	--
char [ ]	--	5'838	--	351'488	--	--
java.lang.String	--	5'860	--	140'640	--	--
byte [ ]	--	48	--	110'848	--	--
int [ ]	--	223	--	92'120	--	--
java.lang.Object [ ]	--	794	--	38'992	--	--
java.util.Hashtable$Entry	--	1'607	--	38'568	--	--
java.util.Hashtable$Entry [ ]	--	222	--	36'976	--	--
javax.swing.JPanel	--	49	--	16'856	--	--
javax.swing.JRadioButton	--	24	--	11'136	--	--
java.lang.String [ ]	--	187	--	8'584	--	--
java.util.Hashtable	--	212	--	8'480	--	--
javax.swing.JLabel	--	21	--	8'232	--	--
sun.java2d.loops.Blit	--	199	--	7'960	--	--

Abb. 1 - JProbe-Analyse-Ergebnis

Wie unter Total zu sehen ist, beträgt das verbrauchte Heap Memory rund 68 MB. Dies entspricht der Grösse des gesamten Heaps. Der Heap ist also voll. Dies erklärt schon einmal, warum die Applikation blockiert und bestätigt die Vermutung, dass ein Speicherleck besteht. Der Speicherverbrauch der einzelnen Klassen ist nach absteigender Grösse sortiert. Hierbei fällt auf, dass der grösste Speicherverbrauch von den simMessage- und simTimer-Objekten stammen, die sich im simCore-package befinden. Diese Objekte sind in HashMaps abgelegt und mit Long Variablen nummeriert. Aus diesem Grund benötigen auch HashMap und Long sehr viel Speicher.

Das Package simCore gehört zur extern eingebundenen Klassenbibliothek JTOOPS-Framework (5). JTOOPS ist ein Java-Simulations-Framework. Es wurde ebenfalls an der HSR entwickelt. Mit JTOOPS lassen sich verteilte Systeme zeitlich und logisch beschreiben lassen. JTOOPS stellt Simulations-Prozesse und –Prozessoren zu verfügen. Diese laufen in eigenen Threads ab und können über Messages miteinander kommunizieren.

Es könnte der Eindruck entstehen, dass JTOOPS für das Speicherleck verantwortlich ist, da es sich bei den identifizierten Objekten um JTOOPS-Klassen handelt. Hierbei ist aber zu bedenken, dass die Simulations-Prozesse und –Prozessoren des HDLCSimulator direkt von den entsprechenden Klassen des JTOOPS-Frameworks abgeleitet werden. Es ist viel wahrscheinlicher, dass die Abstimmung zwischen der Real-Zeit und der Modell-Zeit fehlerhaft ist. Dies hat zur Folge, dass die Modell-Prozesse viel zu oft um eine Antwort beim Benutzer anfragen. Da der Benutzter natürlich nicht so schnell wie ein Computer reagieren kann, häufen sich immer mehr Anfragen (simMessages) im Speicher an.

In einem ersten Schritt wird versucht, den Programm-Ablauf des HDLCSimulators dahingehend zu verbessern, dass auch ein stabiler Betrieb möglich ist ohne dieses Timing-Problem zu beheben. Der zweite Schritt beinhaltet die genaue Analyse des Timing-Problems und dessen Behebung.

2.1.1 Programm-Ablauf in der aktuellen Version

Wird der Simulator gestartet, so baut sich zuerst die Simulationsumgebung auf. Es werden zum Beispiel alle benötigten Simulation-Prozesse und –Prozessoren gestartet. Sie bleiben während des ganzen Programm-Ablaufs aktiv. Die Erstellung des GUIs wird ebenfalls von der Simulationsumgebung initiiert. Das folgende Klassendiagramm zeigt die für den Programmstart wichtigsten Klassen.

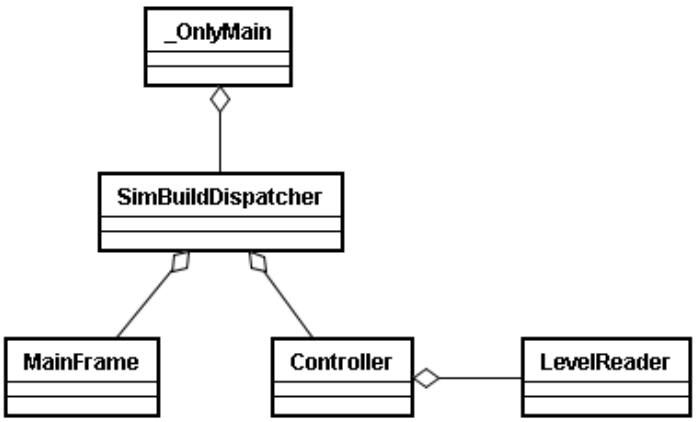

Abb. 2 - Diagramm der wichtigsten Klassen der alten Kontrollstruktur

Wird das Programm gestartet, instanziiert die _OnlyMain-Klasse den SimBuildDispatcher. Dieser bildet den Kern der Simulationsumgebung und ist dafür verantwortlich, dass alle sechs benötigten SimProzesse und SimProzessoren gestartet werden. Die einzelnen SimProzesse sind UserLayer2, HostLayer2, zweimal Layer3, Relais (alle nicht im Diagramm, da für diesen Teil nicht von Bedeutung) und der Controller-Prozess. Neben dem Erzeugen der Simulationsumgebung ist der SimBuildDispatcher ebenfalls dafür verantwortlich, dass das GUI gestartet wird. Dies passiert, indem er die Klasse MainFrame instanziiert. Diese baut dann das komplette GUI auf. Im Klassendiagramm oben ist als einziger SimProzess die Controller-Klasse zu sehen. Sie ist der zentrale Simulations-Prozess. Sie ist dafür verantwortlich, dass der richtige Level geladen wird (mit Hilfe des LevelReaders) und überprüft, wann ein Level als erfüllt gilt. Ist dies der Fall, lädt er den nächsten Level.

Damit man sich den Programm-Start besser vorstellen kann, folgt an dieser Stelle noch ein Sequenzdiagramm. Darin werden die oben beschriebenen Abläufe noch bildlich dargestellt.

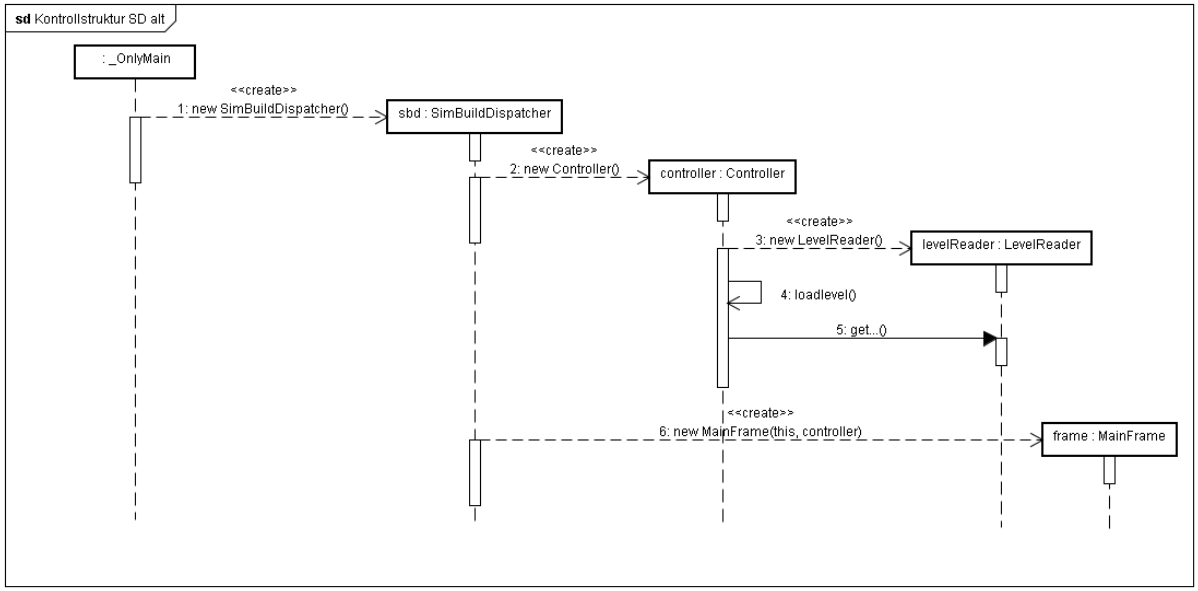

Abb. 3 - Sequenzdiagramm des aktuellen Programm-Starts

Wie in der Analyse zu Beginn des Kapitels beschrieben, gibt es ein Performance-Problem mit der aktuellen Implementation der SimProzesse. Dadurch wird der Heap stetig mit Daten gefüllt, auch wenn gar keine Aktion vorgenommen wird. Da die SimProzesse über die ganze Dauer des HDLCSimulator-Betriebs laufen, wird dieser nach ein paar Minuten zuerst langsamer und blockiert zuletzt komplett.

2.1.2 Programm-Ablauf in der neuen Version

Mit der folgenden Lösung wird der HDLCSimulator schon stabiler gemacht, noch bevor das Problem mit den simMessages und simTimer angegangen wird.

Beim Laden des neuen Levels wird jeweils die gesamte Simulations-Umgebung neugestartet. Um dies umsetzten zu können, wird die neue Klasse GameManager eingeführt. Sie befindet sich ausserhalb der Simulations-Umgebung und wird direkt von _OnlyMain instanziiert. Zum eine besteht ihre Aufgabe darin, die Simulationsumgebung aufzubauen. Zum anderen ist die GameManager-Klasse auch dafür verantwortlich, dass das GUI aufgebaut wird. Dadurch werden diese beiden Komponenten von einander entkoppelt.

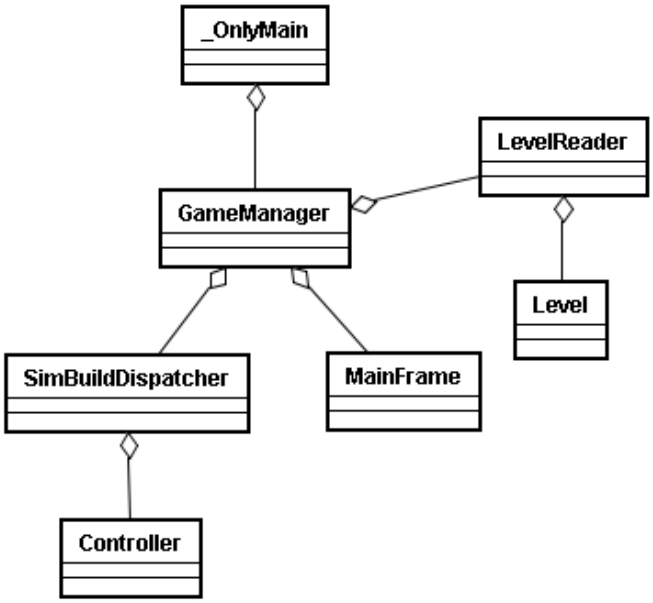

Abb. 4 - Diagramm der wichtigsten Klassen der neuen Kontrollstruktur

Im Klassendiagramm ist zu sehen, dass der LevelReader in dieser Version ebenfalls vom GameManager instanziiert wird. Bis anhin war der Controller-Prozess für das Laden der Levels zuständig. Wie oben erwähnt wird aber die Simulationsumgebung mit jedem Level neugestartet. Somit muss dem Controller-Prozess diese Funktion entzogen werden, da sonst die Level-Informationen jedes Mal verloren gehen würden. Diese Arbeit übernimmt nun ebenfalls die neue GameManager-Klasse. Um das Einspielen neuer Level im Programmcode übersichtlicher zu gestalten, wird zusätzlich die neue Klasse Level eingeführt. Somit können alle Level-Informationen als Level-Objekt an die Simulations-Umgebung weitergegeben werden. Bis anhin wurden die Parameter mit vielen get()-Methoden einzeln aus dem LevelReader-Objekt ausgelesen.

Im den folgenden Zwei Abbildungen ist der neue Ablauf bei Programm-Start sowie das Laden eines neuen Levels als Sequenzdiagramme dargestellt.

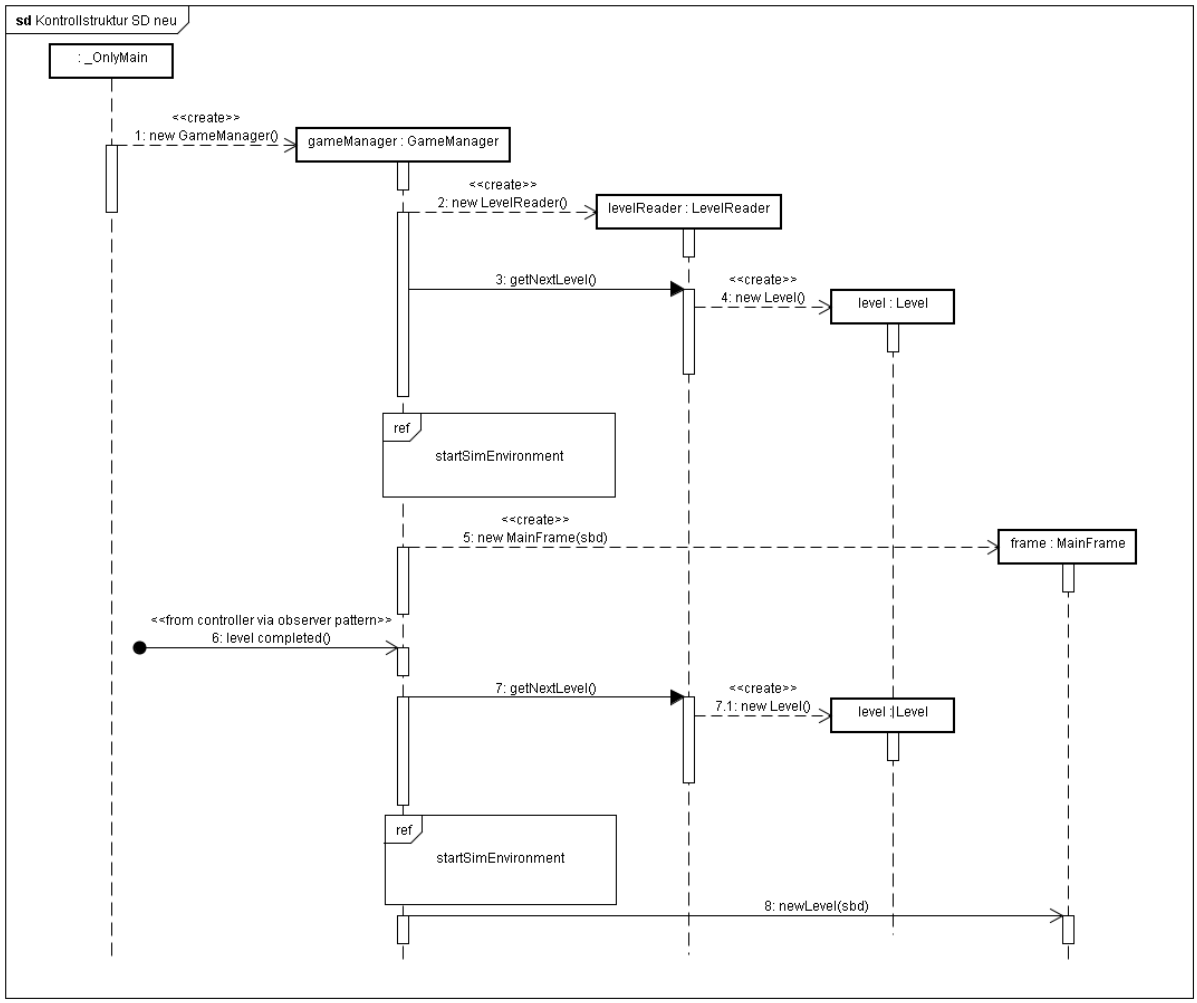

Abb. 5 - Sequenzdiagramm des neuen Programm-Starts

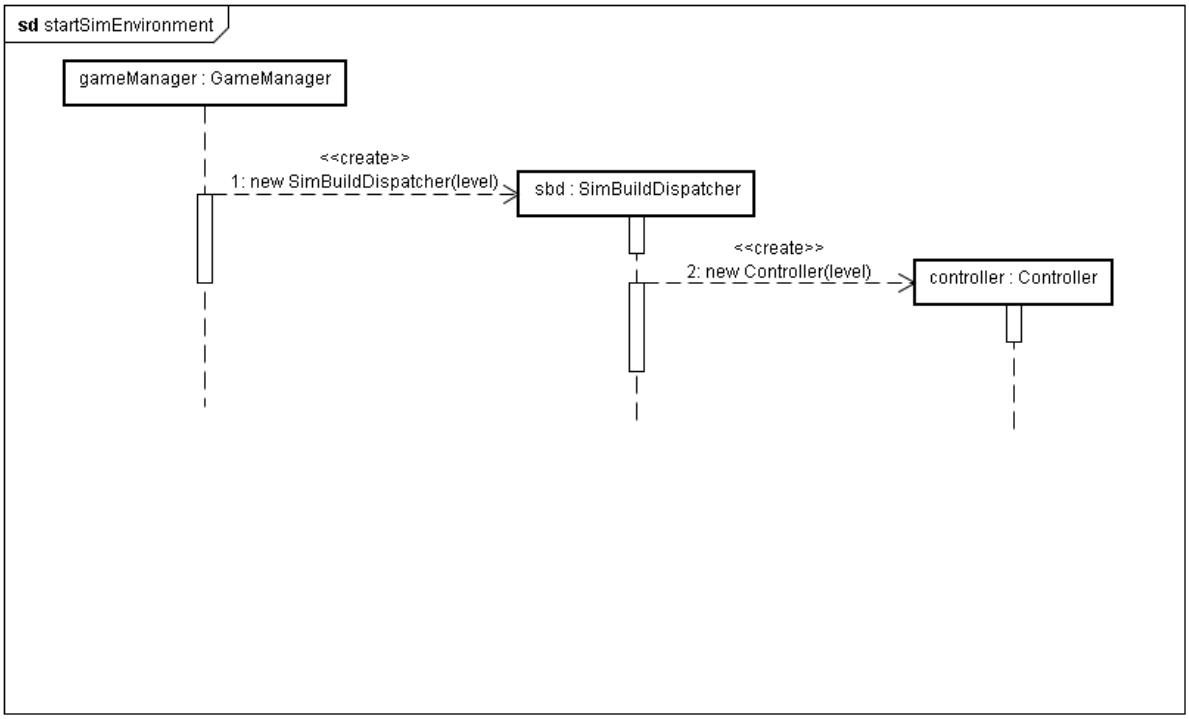

Abb. 6 - Sequenzdiagramm der Referenz startSimEnvironment

Um das Diagramm übersichtlicher zu gestalten wird das Starten der Simulationsumgebung als eigenes Diagramm mit dem Namen startSimEnvironment modelliert. Im Hauptdiagramm wird dann an zwei Stellen darauf referenziert (mittels UML-Struktur „ref“ und dem Namen).

Damit der GameManager weiss, wann ein Level erfüllt wurde, wird das Observer-Pattern verwendet. Der Controller überwacht nach wie vor die Korrektheit der übertragenen Daten. Entscheidet er, dass ein Level erfüllt ist, kann er mittels Observer eine entsprechende Meldung an den GameManager absetzen. Diese Meldung ist in Abb. 5 als Befehl sechs ersichtlich. Trifft eine solche Meldung ein, holt sich der GameManager beim LevelReader mittels getNextLevel() das neue Level-Objekt. Anschliessend startet er die gesamte Simulations-Umgebung neu und übergibt ihr gleichzeitig den neuen Level. Da das GUI in der neuen Version vom GameManager gestartet wird, kann lediglich die neue Simulationsumgebung dem GUI übergeben werden. Ein Neustart der Benutzeroberfläche ist nicht nötig.

Eine weitere kleine Anpassung betrifft nicht das Design des HDLCSimulators, sondern die Level Definition. In jedem Level wird eine TimeOut-Konstante definiert. Dieser Wert legt fest, wie lange der Benutzer Zeit hat, bis die Gegenstation ein Paket nochmals sendet. In der bisherigen Definition der Standard-Level dauert es in jedem Level jeweils 120 Sekunden bis ein Resend durchgeführt wird. Um den Schwierigkeitsgrad der Level zusätzlich zu erhöhen, wurde dieser TimeOut-Wert zum Teil bis auf 20 Sekunden gesenkt. Dies hat zur Folge, dass sich bei zu langsamer Benutzer-Reaktion immer mehr Pakete im Input-Buffer ansammeln und der Überblick verloren gehen kann.

2.1.3 Analyse des Timing-Problems

Die SimTimer von JTOOPS können zum zeitlich verzögerten Senden von Messages eingesetzt werden. Läuft der Timer ab, wird der Dispatcher informiert, dass der betroffene SimProzess Rechenzeit benötigt. Der HDLC-Simulator setzt die Timer lediglich für diese Benachrichtigungen ein. Bei Ablauf des Timers wird eine TimerExpired-Message gesendet und der Prozess erhält Rechenzeit. Alle jemals erstellten Timer, ob abgelaufen oder nicht, werden in einer HashMap des SimProzesses gespeichert.

Nachfolgender Code zeigt einen Ausschnitt aus der behaviour()-Methode der Klasse Layer3. Layer3 ist von der simProcess-Klasse von JTOOPS abgeleitet. Dieser Code steht repräsentativ für alle von simProcess abgeleiteten Klassen des HDLCSimulators.

```
 1 simMessage msg;
 2 while(true){
 3    try {
 4          msg = new simMessage(-1, "");
 5          while (msg.getType() == -1) {
 6                getProcessor().createTimer(socket, 1);
 7                msg = socket.receive();
 8
 9                Thread.yield();
10          }
11
12          state = state.handleMessageType(controller, this, msg);
13
14          setChanged();
15          notifyObservers();
16
17          } catch (simException e) {
18                e.printStackTrace();
19          }
20 }
```

Die Zeilen 3 bis 19 werden aufgrund der while(true)-Schleife während der gesamten Lebensdauer der Simulation immer wieder durchlaufen. In der Zeile 6 sieht man wie bei jedem Durchlauf der inneren Schleife ein neuer Timer erstellt und sogleich gestartet wird. Dies geschieht solange, bis die erhaltene Message keine TimerExpired-Message (Type: -1) mehr ist. Durch diesen Umstand werden in kürzester Zeit extrem viele Timer erstellt. Wie weiter oben erwähnt trifft dieses Problem bei allen SimProzessen des HDLCSimulators zu.

Um dieses Problem zu beheben muss darauf geachtet werden, dass pro SimProzess nur ein Timer existiert. Es gibt zwei Möglichkeiten: 1. Der Timer wird nach der Erstellung nur noch jeweils zurückgestellt und wiederverwendet. 2. Bevor ein neuer Timer erstellt wird, muss der alte, abgelaufene Timer gelöscht werden. Da JTOOPS die erste Möglichkeit nicht unterstützt, wird Möglichkeit zwei verwendet. Dadurch kann sichergestellt werden, dass während der gesamten Simulation nur sechs Timer existieren; für jeden SimProzess einer.

2.2 Analyse des Exception-Handlings

Performance	Exceptions	Fehler	Erweiterungen

Um weitere Unzulänglichkeiten im Code festzustellen, werden sämtliche Quellcodes aller Klassen untersucht. Dabei fiel besonders das bestehende Exception-Handling auf. Dieses ist über das ganze Programm betrachtet sehr uneinheitlich. Dies vermindert die Wartbarkeit der Applikation.

Im HDLCSimulator ist ein Log-Klasse implementiert. Innerhalb dieser Klasse kann eine Art Debug-Modus aktiviert werden. An einigen Stellen im Programmcode stehen fürs Logging bestimmte Zeilen. Nach Aktivierung des Debug-Modus werden die gewünschten Informationen auf die Konsole ausgegeben. Ist der Debug-Modus deaktiviert folgt keine Ausgabe. Leider wird diese Möglichkeit im aktuellen Programmcode nur sporadisch genutzt. An etwa 20 Stellen im Code wird eine try-catch-Struktur verwendet, um das Exception-Handling zu implementieren. Aber nur in ein paar wenigen Fällen wird innerhalb des catch-Blocks des Exception-Handlings eine Meldung an die Log-Klasse gesendet. Nachfolgend ein entsprechender Code-Ausschnitt aus der aktuellen Programm-Version. Dieser Ausschnitt ist repräsentativ für viele weitere Fälle.

```
 1 private simMessage waitFor(int event){
 2
 3     simMessage msg = new simMessage(-1, "");
 4     while(msg.getType() != event){
 5           try {
 6                 getProcessor().createTimer(socket, 1);
 7                 msg = socket.receive();
 8
 9                 if (msg.getType() != -1 && msg.getType() != event){
10                       Log.printRed("Unbehandelte Nachricht beim …);
11                 }
12
13                 // verhindert OutOfMemoryError: Java heap space
14                 // indem die while-schleife künstlich gebremst wird
15                 Thread.sleep(SLEEPTIME);
16           } catch (simException e) {
17                 e.printStackTrace();
18           } catch (InterruptedException e) {
19                 e.printStackTrace();
20           }
21     }
22     return msg;
23 }
```

In diesem Code-Ausschnitt sieht man in den Zeilen 16-20, dass bei einem allfälligen Auftreten einer Exception, dessen Inhalt auf die Konsole ausgegeben wird. Dies geschieht mit dem Aufruf e.printStackTrace(). Dabei wird nicht berücksichtigt, ob der Debug-Modus aktiviert ist oder nicht.

In den folgenden Code-Ausschnitten sehen wir nun noch zwei Fälle, in denen kein Exception-Handling gemacht wird, obwohl der Code eine Exception werfen könnte.

```
 1 private static byte[] toByteArray(String s){
 2     byte[] b = new byte[s.length()/8];
 3
 4     for (int i = 0; i < b.length; i++){
 5           int a = i*8;
 6           try {
 7                 String tmp = s.substring(a, a + 8);
 8                 if (tmp.charAt(0) == '1'){
 9                 // wenn 1. Bit '1' ist, dann wäre das eine negative Zahl,
10                 // das darf nicht sein, daher einfach auf positiv wechseln
11                       tmp = "0"+tmp.substring(1);
12                 }
13                 b[i] = Byte.parseByte(tmp,2);
14           } catch (NumberFormatException e) {
15                 e.printStackTrace();
16           }
17     }
18
19     return b;
20 }
```

Hier sehen wir zwar, dass ein Exception-Handling gemacht wird, dieses bezieht sich jedoch nur auf die Methode parseByte() in Zeile 13. Diese Methode kann die aufgeführte NumberFormatException (Zeile 14) werfen. Das Problem liegt hier in Zeile 7. Die substring()-Methode kann eine IndexOutOfBoundsException werfen. Diese Exception wird hier jedoch nicht aufgeführt. Für diesen Fall wäre der try-catch-Block völlig nutzlos.

```
1 private void sendBtnClickedSDU() {
2
3     if (message.getType() == IMessageTypes.DATA_indication){
4           message = new SDU(true);
5           message.setType(IMessageTypes.DATA_indication);
6           message.setData(sduDropPanel.getMessage().getData());
7     }
8     userLayer.getOutputBuffer().add(message);
9
10    // Panels leeren
11    clearInspectorAndComposer();
12 }
```

Das zweite Beispiel zeigt die Methode sendBtnClickedSDU(). Wenn der Benutzer im ComposerPanel des GUIs auf den Knopf „Send message“ im Bereich „New SDU“ klickt wird sie ausgeführt. Interessant ist der if-Block (ab Zeile 3). Er wird nur ausgeführt, wenn vorher „Data.indication“ ausgewählt wurde. Das Problem liegt in Zeile 6. Der Simulator geht ohne Überprüfung davon aus, dass der Benutzer tatsächlich Daten in das DropPanel gezogen hat. Er weist den Inhalt des DropPanels der Nachricht zu. Hat der Benutzer es unterlassen, Daten zu hinterlegen, wird das Argument der setData()-Methode NULL. Dies hat zur Folge, dass das Programm eine NullPointerException wirft. Nach Auftritt dieser Exception kann ohne Beeinträchtigung weitergearbeitet werden und auch das Protokoll der Schicht 3 verhält sich korrekt indem es nicht auf die SDU reagiert. In einer Applikation sollte jedoch jede eventuell auftretende Exception behandelt oder noch besser von vorne herein vermieden werden.

Wie in der Analyse des Exception-Handlings festgestellt wurde, existieren verschiedenartige Problem im HDLCSimulator. Hier sind nochmals die beiden Arten von Problemen aufgeführt:

- Fehlende Verwendung der Log-Klasse
- Nicht-Abfangen von möglichen Exceptions

Die Folgen dieser Probleme reichen von beeinträchtigter Wartbarkeit bis hin zu Systemabstürzen. Aus diesem Grund ist es nötig, die gefunden Probleme zu korrigieren und das Exception-Handling einheitlich zu gestalten.

2.2.1 Exception-Handling der neuen Version

Bei eingeschaltetem Debug-Modus steht die Log-Klasse dem Entwickler mit wichtigen Informationen zur Seite. Da diese Struktur bereits besteht, sollte sie auch beim Exception-Handling konsequent eingesetzt werden. Das heisst, dass bei aktiviertem Debug-Modus, alle auftretenden Exceptions an die Log-Klasse gesendet werden müssen. Es soll auch eine Beschreibung mitgeliefert werden, was schief gelaufen ist. Ist der Modus deaktiviert, soll auf der Konsole gar kein Output erscheinen und das Programm wie im Normal-Betrieb laufen. Falls bei Tests mit ausgeschaltetem Debug-Modus nun trotzdem eine Exception auf der Konsole ausgegeben wird, ist sofort klar, dass man mit einem völlig neuen Fehler konfrontiert ist. Ansonsten wäre die Exception an die Log-Klasse weitergeleitet worden.

Falls es möglich ist, eine Exception so zu behandeln, dass nach dessen Auftritt wieder ein normaler Programm-Ablauf garantiert werden kann, wird ein entsprechender Code implementiert.

Im Fall der fehlenden Exceptions werden verschiedene Lösungsansätze verwendet. Bei der fehlenden Exception der substring()-Methode wird für diese noch einen eigenen, zusätzlichen catch-Block implementiert.

Um die NullPointerException zu vermeiden, wird kein Exception-Handling ausgeführt, sondern der Programmcode so verändert, dass dieser Fall schon vorher überprüft wird. Es muss mit einer if-Anweisung überprüft werden, ob überhaupt Daten im sduDropPanel vorhanden sind. Ist dies nicht der Fall, soll ein Button-Klick keine Message verschicken.

2.3 Analyse auftretender Fehler

Performance	Exceptions	Fehler	Erweiterungen

Beim Testen der Applikation über das GUI, kam noch ein Problem zum Vorschein. Er betrifft die Klasse BitStuffer. Sie ist für das Bitstuffing/Bitunstuffing in den Paketen zuständig. Diese Klasse enthält für beide Operationen eine Funktionen.

```
 1 public static String stuff(String bits){
 2     // Einsen zählen
 3     int countOnes = 0;
 4
 5     StringBuffer buffer = new StringBuffer();
 6     for(int i = 0; i < bits.length(); i++){
 7
 8           // wenn eine Eins kommt, Zähler erhöhen
 9           if(bits.charAt(i) == '1'){
10                 countOnes++;
11           }
12           // wenn eine Null kommt, dann wieder von vorne beginnen zählen
13           else{
14                 countOnes = 0;
15           }
16           // Zahl anfügen
17           buffer.append(bits.charAt(i));
18
19           // wenn fünf Einsen gekommen sind, dann eine Null einfügen
20           if(countOnes == 5){
21                 buffer.append('0');
22                 countOnes = 0;
23           }
24     }
25     return buffer.toString();
26 }
```

In der stuff()-Funktion ist zu sehen, wie in der for-Schleife (ab Zeile 6) jedes Bit im String „bits“ analysiert wird und dann zum „buffer“ hinzugefügt wird. Falls nun fünfmal in Folge eine „1“ auftritt, wird zusätzlich noch eine „0“ zum „buffer“ hinzugefügt (Zeilen 20-23).

```
 1 public static String unstuff(String bits){
 2     // Einsen zählen
 3     int countOnes = 0;
 4     StringBuffer buffer = new StringBuffer();
 5     for(int i = 0; i < bits.length(); i++){
 6
 7           // wenn eine Eins kommt, Zähler erhöhen
 8           if(bits.charAt(i) == '1'){
 9                 countOnes++;
10                 if (countOnes == 5){
11                       buffer.append(bits.charAt(i));
12                       i++;
13                       countOnes = 0;
14                 }
15                 else{
16                       buffer.append(bits.charAt(i));
```

```
17                      }
18              }
19              // wenn eine Null kommt, dann wieder von vorne beginnen mit dem Zählen
20              else{
21                      buffer.append(bits.charAt(i));
22                      countOnes = 0;
23              }
24      }
25      return buffer.toString();
26  }
```

Die unstuff()-Funktion ist ähnlich aufgebaut. Ihre Aufgabe besteht darin, nach fünfmaligem Auftreten einer „1“ das nächste Zeichen (die hinzugefügte „0“) zu überlesen.

Wird das BitStuffing einmal ausgeführt, funktioniert die stuff()-Funktion tadellos. Das Problem tritt auf, wenn der Anwender das BitStuffing mehrmals ausführt. In der jetzigen Implementation wird bei jedem Aufruf eine weitere „0“ nach den fünf „1“ eingefügt. Dies führt dazu, dass der Empfänger bei einmaligem Aufrufen der unstuff()-Funktion falsche Daten erhält.

Das gleiche resultiert bei mehrmaligem Ausführen der unstuff()-Funktion. Da das nächste Zeichen nach fünf „1“ einfach überlesen wird, führt dies bei jedem zusätzlichen Ausführen der Funktion zum Verlust von je einem Bit.

Die Implementierung ist gemäss Protokoll korrekt, dabei ist jedoch auch gewährleistet, dass diese Funktionen nur einmal ausgeführt werden. In unserem Fall des Lernspiels, bei dem der Benutzer diesen Vorgang manuell ausführt, ist es angebracht, dies zu berücksichtigen.

2.3.1 BitStuffing/Unstuffing der neuen Version

Die Implementierung wird so korrigiert, dass ein wiederholtes Ausführen die Daten nicht mehr verändert. Es ist jedoch nicht möglich, auf der Ebene des Algorithmus eine Änderung vorzunehmen. Der folgende simple Ansatz liegt zwar nahe, ist aber bei genauerer Betrachtung nicht durchführbar:

Falls beim BitStuffing auf die fünf „1“ eine „0“ folgt wird nichts gemacht und falls beim BitUnstuffing nach fünf „1“ wieder eine „1“ folgt wird ebenfalls nichts gemacht.

Das Problem zeigt sich zum Beispiel beim Empfangen der Bitfolge „11111011“. Verwendet man den oben erwähnten Ansatz, wäre die Unstuff()-Methode überfordert. Es ist für sie nicht entscheidbar, ob die ursprüngliche Bitfolge „1111111“ war und ein BitStuffing durchgeführt wurde oder ob die ursprüngliche Bitfolge bereits „11111011“ war und kein BitStuffing nötig war.

Aus diesem Grund muss für die Behebung dieses Problems bei der GUI-Implementierung angesetzt werden. Dort wird sichergestellt, dass der entsprechende Button nur einmal pro Message gedrückt werden kann.

2.4 Analyse möglicher Erweiterungen

Performance	Exceptions	Fehler	Erweiterungen

Bei der Verwendung des Lernspiels vermisst man diverse Features. Diese würden den Spielgenuss und damit auch den angestrebten Lerneffekt erheblich steigern.

Es ist in der gegenwärtigen Version nicht möglich, eine begonnene Spielstufe neu zu starten. Dies wäre jedoch wünschenswert, falls man zum Beispiel ein noch benötigtes Paket bereits verworfen hat. Desweiteren ist es auch nicht möglich die Spielstufe auszuwählen. Beim Programmstart wird immer der erste Level gestartet. Bei erfolgreicher Absolvierung wird dann Level zwei geladen. Wer also bei einem früheren Spiel die ersten fünf Levels bewältigt hat, muss beim nächsten Spiel wieder bei Level eins beginnen.

Bei der Implementierung dieser beiden Features kommt die vorgeschlagene Umstrukturierung aus dem Unterkapitel „Analyse der Performance“ entgegen. Zur Erinnerung: Es soll mit jedem neuen Level auch die Simulationsumgebung neugestartet werden. Dabei wird der neuen Simulationsumgebung die gewünschte Spielstufe mitgegeben.

In der Toolbar des GUIs wird ein Restart-Button hinzugefügt. Bei Betätigung wird die Simulationsumgebung neu gestartet und die gleiche Spielstufe nochmals mitgegeben. Desweiteren wird auch ein Dropdown-Menü in der Toolbar platziert. Dieses listet alle verfügbaren Levels auf. Bei einer getroffenen Auswahl wird die Simulationsumgebung ebenfalls neu gestartet und der ausgewählte Level mitgegeben.

Ein weiteres praktisches Feature ist ein Hint-System. Bei gewissen Levels hat der Benutzer die Möglichkeit, sich einen oder mehrere Tipps anzeigen zu lassen. Damit soll verhindert werden, dass der Anwender in einer gewissen Spielsituation hängen bleibt. Der Hint-Button wird ebenfalls in der Toolbar der Applikation untergebracht. Er zeigt bei Betätigung ein Dropdown-Menü mit den verfügbaren levelspezifischen Hinweisen. Nach dem Anklicken eines Eintrages wird eine Dialogbox mit dem entsprechenden Hint angezeigt.

Der Send Buffer im GUI hat die Funktion, an die Gegenstation verschickte Messages zu speichern. Somit besteht die Möglichkeit bei einer fehlgeschlagenen Übertragung die Daten nochmals zu versenden.

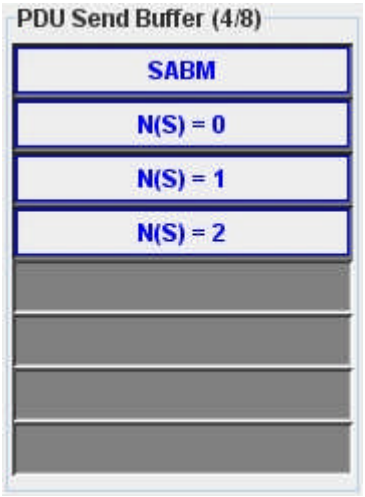

Abb. 7 - Send Buffer des GUIs

Das Wiedersenden gestaltet sich aber unnötig kompliziert: Die Message muss in eine neue PDU gedroppt werden, um die versendetet Daten zu erhalten. Anschliessend müssen alle Kontroll-Bits nochmals gesetzt werden. Auch Checksum-Berechnung und Bitstuffing müssen nochmals durchgeführt werden.

Das Kontext-Menü des Buffers bietet nur die Möglichkeit, nicht mehr benötigte Pakete zu löschen. Um den Komfort zu steigern wird dem Kontext-Menü ein weiterer Punkt hinzugefügt. Dieser erlaubt das nochmalige Versenden des Pakets ohne dass alle Einstellungen nochmals vorgenommen werden müssen.

2.5 Ergebnis der Analyse

Performance	Exceptions	Fehler	Erweiterungen

Nach Abschluss der Analyse geht es darum, die verschiedenen Überarbeitungen zu priorisieren. Dies geschieht im Hinblick darauf, dass der HDLCSimulator nach Abschluss dieser Diplomarbeit im Unterricht als Lernhilfe eingesetzt werden soll.

Das wichtigste Refactoring betrifft die Performance-Problematik. Dieses muss unbedingt als erstes durchgeführt werden. Ohne diese Aktion ist der HDLCSimulator nicht einsetzbar, da er nach wenigen Minuten langsamer wird und danach sogar zum Stillstand kommt. Als nächstes muss der BitStuffing Fehler sowie die nicht behandelten Exceptions behoben werden. Ansonsten können während des Spiels verheerende Situationen auftreten. Diese führen zwangsläufig zu einem Neustart der Simulation. Nachdem die grössten Unzulänglichkeiten beseitigt sind, können die Erweiterungen implementiert werden. Sie sorgen für eine stark gesteigerte Bedienbarkeit der Applikation. So kann der Spielgenuss und somit auch der Lerneffekt erhöht werden. Die niedrigste Priorität hat die Vereinheitlichung der Exceptions-Struktur. Sie erhöht die Wartbarkeit der Applikation hat aber keinen direkten Einfluss auf die Spielerfahrung.

3 Ergebnis der Aktionen

3.1 Ergebnis des Performance-Refactorings

Performance	Exceptions	Fehler	Erweiterungen

Zur Erinnerung zeigt die Grafik unten nochmals das Klassendiagramm der wichtigsten Klassen. Dies ist der Stand bevor mit den Refactorings begonnen wurde.

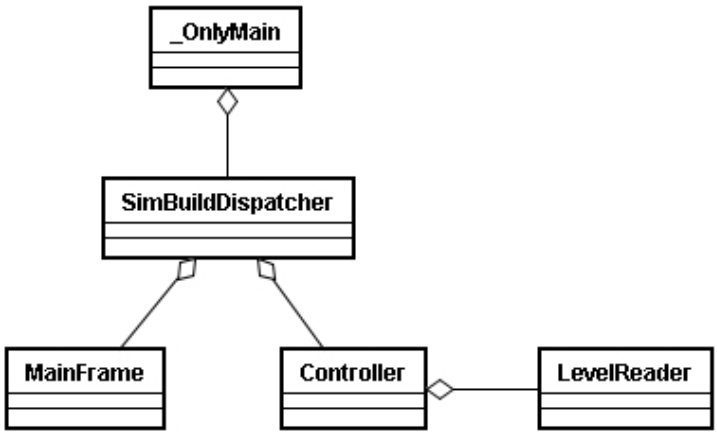

Abb. 8 - Diagramm der wichtigsten Klassen der alten Kontrollstruktur

Es ist zu erkennen, dass die Simulation (SimBuildDispatcher) für die Generierung des GUIs verantwortlich ist. Ebenfalls liegt das Level-Management bei der Simulation (Controller). Der erste Schritt des Refactorings besteht darin, die GameManager-Klasse zu erstellen. Sie wird von _OnlyMain instanziiert und startet ihrerseits die Simulation. So entsteht eine Kontrollklasse, die in der Hierarchie über der Simulation steht. Dies ist nötig, wenn wir während des Spiels die Möglichkeit haben wollen, die Simulation zu beenden und wieder neu zu starten. Als nächstes soll verhinder werden, dass das GUI auch bei jedem Simulationsstart neu gestartet wird. Dazu wird die Erstellung des GUIs von der SimBuildDispatcher-Klasse in die GameManager-Klasse verschoben. Hierbei ist zu beachten, dass das GUI weiterhin die Simulationsumgebung kennen muss. Grund dafür sind die Observer des GUIs. Mit Hilfe der Observer kann das GUI Veränderungen in der Simulation erkennen. Somit ist eine Aktualisierung der Benutzeroberfläche während dem Betrieb möglich. Da die Simulation bei jedem Level neu gestartet wird, muss das GUI die Möglichkeit habe, diese neu zu registrieren. Diese Tatsache bedingt die Programmierung einer neuen Methode im MainFrame: registerNewSimEnv(). Dieser Methode wird die neue Simulationsumgebung übergeben. Sie führt bei allen relevanten GUI-Elementen die Aktualisierung durch. Zur Veranschaulichung zeigt der folgende Code-Ausschnitt die Methode registerNewBuffer(). Sie befindet sich in der GUI-Klasse SendBuffer und wird von registerNewSimEnv() aufgerufen. Dieser Ausschnitt steht repräsentativ für weitere fünf ähnliche Methoden in anderen GUI-Elementen.

```
1 public void registerNewBuffer(OutputBuffer buffer){
2     this.buffer = buffer;
3     buffer.addObserver(this);
4     nrOfRows = buffer.getMaxNrOfMessages()/NUMBER_OF_SLOTS_PER_ROW;
5     fillGrid();
```

```
6 }
```

Das übergebene Argument (Zeile 1) ist der OutputBuffer der neuen Simulationsumgebung. Er wird in Zeile 2 der Buffer-Variablen der GUI-Klasse zugewiesen. In Zeile 3 sieht man die weiter oben erwähnte Zuweisung der GUI-Klasse als Observer von buffer. In der nächsten Zeile wird noch eine benötigte Variable des Buffers ausgelesen. Abschliessend erfolgt noch das Neuzeichnen des GUI-Elements mittels fillGrid(). Somit ist der OutputBuffer der neuen Simulation korrekt im GUI registriert.

Die bis jetzt durchgeführten Veränderungen der Klassenstruktur sind im nächsten Diagramm ersichtlich.

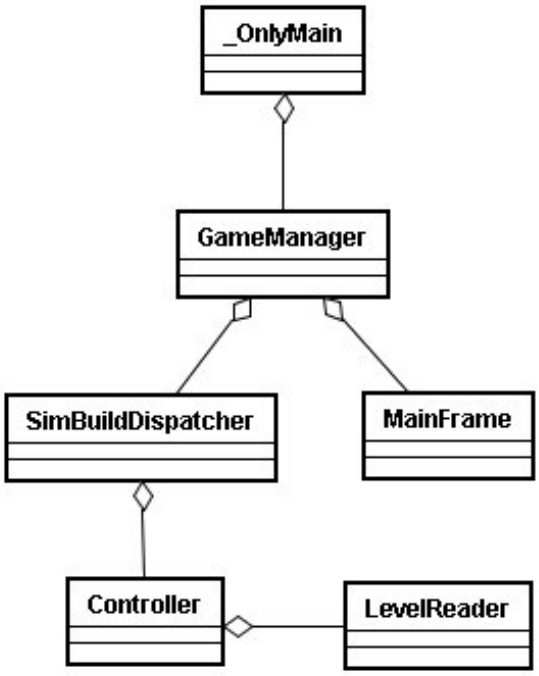

Abb. 9 - GameManager eingefügt und GUI-Erzeugung verlagert

Wie in der Analyse beschrieben ist der GameManager für den gesamten Spiel-Ablauf zuständig. Dazu gehört ausser dem Starten der Simulationsumgebung und dem Erstellen des GUI auch noch das Laden des richtigen Levels. Deshalb muss auch die Levelverwaltung dem Controller entzogen werden. Ansonsten wäre der aktuelle Spielstand beim jedem Neustart der Simulation wieder verloren. Zu diesem Zweck wird der LevelReader neu ebenfalls vom Gamemanager instanziiert. Damit nicht alle Levelparameter einzeln an die Simulation weitergegeben werden müssen, wird die Level-Klasse eingeführt. Sie enthält für jeden Levelparameter eine Variable. Um diese Variablen speichern und auslesen zu können, verfügt sie noch über die entsprechenden Setter- und Getter-Methoden. Wird eine neue Spielstufe gestartet, wird ein Level-Objekt mit den entsprechenden Daten gefüllt und anschliessend der Simulationsumgebung übergeben. In der bisherigen Version war der LevelReader für die Auswahl des richtigen Levels zuständig. Wurde ein Level beendet, bekam der LevelReader eine Mitteilung und lud den Level mit der nächst höheren Levelnummer. Um mehr Flexibilität zu erhalten, wird diese Laden-Methode in der neuen Version umgeschrieben. Ihr kann eine Levelnummer übergeben werden und sie gibt das entsprechende Level-Objekt zurück. Somit ist neu der Gamemanager für die Auswahl des korrekten Levels zuständig.

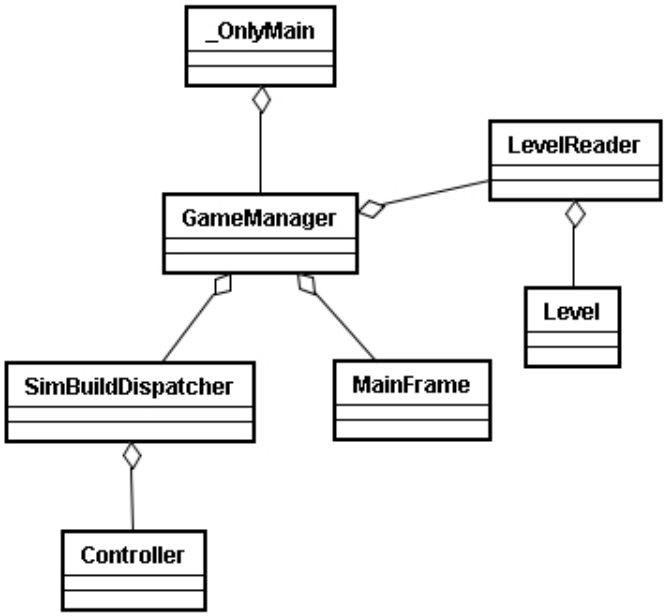

Abb. 10 - Diagramm der wichtigsten Klassen der neuen Kontrollstruktur

Dieses Diagramm zeigt Abhängigkeiten der wichtigsten Klassen wie sie in der neuen Version implementiert sind. Gut zu sehen ist, dass der GameManager eine wichtige Stellung einnimmt. Er ist für die Verwaltung der einzelnen Teile des Lernspiels zuständig: Simulationsumgebung, GUI und Level-Management. Um die zentrale Gamemanager-Klasse besser zu verstehen, folgt ein detailliertes Klassendiagram und eine Beschreibung der wichtigsten Methode.

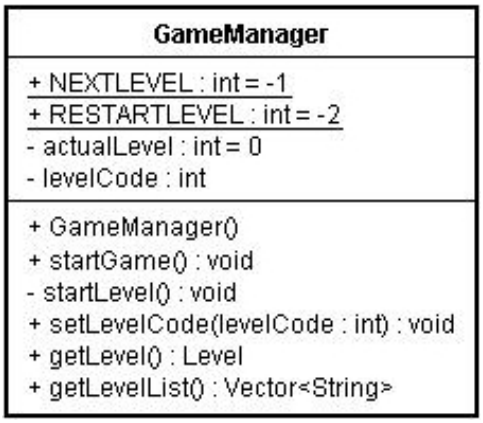

Abb. 11 - GameManager-Klasse

Die wichtigsten beiden Methoden für den Programmablauf sind startGame() und startLevel:

```
 1 public void startGame(){
 2    while(true){
 3          level = levelReader.getLevel(actualLevel);
 4          if (level == null){
 5                JOptionPane.showMessageDialog(null,
 6                "Congratulations!\nYou accomplished all the levels.",
 7                "HDLCSimulator: DONE!", JOptionPane.INFORMATION_MESSAGE);
 8                System.exit(0);
 9          }
10
11          startLevel();
12          Log.print("GameManager: Simulation gestoppt");
13          if(levelCode == NEXTLEVEL){
14                actualLevel++;
15          } else if(levelCode >= 0){
16                actualLevel = levelCode;
17          }
18    }
19 }
```

startGame(): Enthält eine while(true)-Schleife (Zeilen 2-18). Diese wird pro Level einmal durchlaufen. Zuerst wird der aktuelle Level (actualLevel) mit Hilfe des LevelReaders geladen (Zeile 3). Die folgende if-Abfrage dient zur Prüfung, ob der letzte Level beendet wurde. Danach wird in Zeile 11 startLevel() aufgerufen. Nach der Beendigung des Levels wird anhand des levelCode geprüft, welcher Level als nächstes gestartet werden muss. Die while-Schleife startet wieder von vorne.

```
 1 private void startLevel(){
 2    sbd = new SimBuildDispatcher();
 3    sbd.createProcs(this);
 4    if(frame == null){
 5          frame = new MainFrame(sbd);
 6          frame.setVisible(true);
 7    } else{ // Falls GUI schon aufgebaut wurde, werden nur Änderungen
 8            // durchgeführt
 9          frame.registerNewSimEnv(sbd);
10          frame.setVisible(true);
11    }
12    try {
13          Log.print("Nächstes Level beginnt");
14          sbd.startSimulation();
15    } catch (simException e) {
16          JOptionPane.showMessageDialog(null, "Fehler in der
17          Simulation.\nProgramm wird beendet.", "Fehler in der
18          Simulation", JOptionPane.WARNING_MESSAGE);
19          Log.printRed("GameManager: " + e.getMessage());
20          System.exit(0);
21    }
21 }
```

startLevel(): Zuständig für das Starten der Simulationsumgebung und das Erstellen oder Aktualisieren des GUIs. In Zeile 2 wird ein neuer SimBuildDispatcher instanziiert. Danach folgt dessen Initialisierung. Die if-else-Struktur ab Zeile 4 dient zur Erkennung, ob es sich nicht um den ersten Levelstart der Applikation handelt. Falls ja, wird die Haupt-GUI-Klasse MainFrame instanziiert (Zeile 5). Falls nein, wird lediglich die neue Simulationsumgebung im GUI registriert. Dies erfolgt mit der weiter oben erwähnten Methode registerNewSimEnv() (Zeile 9). Damit kann gewährleistet werden, dass das GUI nur die nötigen Elemente neu erstellt. In Zeile 14 wird dann die Simulation mit Hilfe der startSimulation()-Methode des SimBuildDispatchers gestartet.

Damit das Konzept mit diesen beiden Methoden korrekt funktioniert, muss sich die Simulation bei Erfüllung des aktuellen Levels selber herunterfahren. Somit wird die startLevel()-Methode verlassen und der Programmablauf kehrt in die while(true)-Schleife von startGame() zurück. Diese Funktionalität ist jedoch in der alten Version nicht implementiert. Es bestand bis anhin auch keine Notwendigkeit, da während des ganzen Programmablaufs immer dieselbe Simulationsumgebung aktiv war. JTOOPS bietet eine Möglichkeit an, die Simulation von ausserhalb zu stoppen. Dies ist in diesem Fall jedoch nicht praktizierbar. Die Applikation befindet sich innerhalb der Simulation, wenn erkannt wird, dass dieselbe beendet werden muss. Es ist also nötig, dass sich die Simulationsumgebung selbst beenden kann, sobald ein Level erfüllt ist. Dies führt dazu, dass die startSimulation()-Methode des SimBuildDispatchers verlassen wird und der Programmfluss wieder in das GameManager-Objekt zurückkehrt.

Um diese neue Funktion zu implementieren ist ein Eingriff in JTOOPS nötig. Realisiert wird das Ganze mit der kurzen Dispatcher-Methode terminateSimulation():

```
1 public void terminateSimulation(){
2     setState(STATE_SIM_TERMINATED);
3     System.out.println("SimDispatcher: State STATE_SIM_TERMINATED set");
4 }
```

Während der Simulation befindet sich der Dispatcher innerhalb der dispatch()-Methode. Sie ist für die Zuteilung der Rechenzeit der einzelnen SimProzessoren zuständig. Am Anfang der Methode wird in der Abbruchbedingung einer while-Schleife geprüft, welchen Status die Simulation hat. Durch das setzen des Status auf „terminated“ (Zeile 2), fällt diese Prüfung negativ aus und die while-Schleife wird verlassen. Dies führt dann dazu, dass die dispatch()-Methode und schliesslich auch die startSimulation()-Methode beendet werden. Das Ergebnis ist somit die gewünschte Rückkehr des Programmflusses zum GameManager. Die anderen Threads der Simulationsumgebung wurden beim Start der Simulation vom SimBuildDispatcher erzeugt. Nach dem Beenden des Dispatchers haben sie daher keine Verbindung mehr zum Root-Objekt der Java Virtual Machine (JVM). Dadurch werden sie vom Garbage Collector der JVM zerstört.

3.1.1 Behebung des Timing-Problems

Es muss dafür gesorgt werden, dass jeder SimProzess während der gesamten Simulation nur einen Timer besitzt. Deshalb wurde eine Änderung in der behaviour()-Methoden aller SimProzesse gemacht:

```
 1 simMessage msg;
 2 //Timer (Simulationstime) erstellen
 3 try {
 4    timerID = getProcessor().createTimer(socket, 1, false);
 5 } catch (simException e) {
 6    JOptionPane.showMessageDialog(null, "Fehler in der
 7    Simulation.\nProgramm wird beendet.", "Fehler in der
 8    Simulation", JOptionPane.WARNING_MESSAGE);
 9    Log.printRed("Layer3: " + e.getMessage());
10    System.exit(0);
11 }
12 while(true){
13    try {
14          msg = new simMessage(-1, "");
15          while (msg.getType() == -1) {
16                getProcessor().killTimer(timerID);
17                timerID = getProcessor().createTimer(socket, 1);
18                msg = socket.receive();
19
20                Thread.yield();
21          }
22 ...
```

Der Timer wird ausserhalb der while(true)-Schleife erstellt (Zeile 4). Zum späteren Auffinden wird die timerID gespeichert. Wie in der alten Version wird nun solange die while-Schleife ausgeführt (Zeilen 15 – 21), bis socket.receive() eine Message liefert, die keine TimerMessage ist (Zeile 18). Der Unterschied in der neuen Version liegt am Anfang der while-Schleife. Trifft eine TimerMessage ein, wird zuerst der aktuelle Timer zerstört (Zeile 16). Danach wird ein neuer Timer erstellt (Zeile 17). Somit kann gewährleistet werden, dass sich der Heap von Java nicht stetig mit neuen TimerMessages füllt. Das Performance-Problem ist behoben.

3.2 Ergebnis der Exception-Handling-Refactorings

Performance	Exceptions	Fehler	Erweiterungen

Die Idee im Analyse-Teil war, alle Exceptions an die Log-Klasse weiterzugeben, falls keine Behebung möglich ist. Um dies zu verwirklichen wurden die jeweiligen catch-Blöcke ausgebaut. Der folgende Code-Ausschnitt zeigt repräsentativ für alle try-catch-Strukturen die entsprechenden Erweiterungen. Es handelt sich um das Exception-Handling in der LevelReader-Klasse.

```
 1 try {
 2    SAXBuilder builder = new SAXBuilder();
 3    doc = builder.build(levelFile);
 4    root = doc.getRootElement();
 5    levelList = root.getChildren();
 6    Log.print("Anzahl Level: " + levelList.size());
 7 }
 8 catch (JDOMException e) {
 9    JOptionPane.showMessageDialog(null, "Fehler beim Parsen der XML-
10    Datei.\nProgramm wird beendet.", "Fehler beim Parsen",
11    JOptionPane.WARNING_MESSAGE);
12    Log.printRed("JDOM in LevelReader: " + e.getMessage());
13    System.exit(0);
14 }
15 catch (IOException e) {
16    JOptionPane.showMessageDialog(null, "Level-Datei nicht
17    gefunden.\nProgramm wird beendet.", "Datei nicht gefunden",
18    JOptionPane.WARNING_MESSAGE);
19    Log.printRed("LevelReader: " + e.getMessage());
20    System.exit(0);
21 }
```

Im try-Block können zwei verschiedene Exceptions auftreten (Zeilen 1 – 7). Die Fehlerbehandlungen in den beiden catch-Blöcken laufen jedoch gleich ab. Die Zeilen 9 – 11 erzeugen eine Message-Box. Diese informiert den Benutzer, dass eine Exception aufgetreten ist. Weiter gibt sie noch an, bei was oder wo der Fehler passiert ist. Zeile 12 sorgt dafür, dass der Inhalt der Exception an die Log-Klasse weitergegeben wird. Standardmässig erfolgt eine Ausgabe auf die Konsole (falls der Debug-Modus eingeschaltet ist). Am Schluss wird der HDLCSimulator noch mit dem System.exit(0) beendet (Zeile 13). Damit wird gewährleistet, dass bei einem Fehler die ganze Applikation und somit auch das GUI beendet werden.

Um die möglich NullPointerException im ComposerPanel des GUIs zu verhindern, wird mit if-Blöcken gearbeitet:

```
 1 private void sendBtnClickedSDU() {
 2
 3    if (message.getType() == IMessageTypes.DATA_indication){
 4          // Bei einem leeren DropPanel wird nichts gemacht
 5          if (sduDropPanel.getMessage() != null){
 6                message = new SDU(true);
 7                message.setType(IMessageTypes.DATA_indication);
 8                message.setData(sduDropPanel.getMessage().getData());
 9                userLayer.getOutputBuffer().add(message);
10          }
11    } else {
12          userLayer.getOutputBuffer().add(message);
13    }
14    // Panels leeren
15    clearInspectorAndComposer();
16 }
```

Zuerst wird überprüft, ob es sich um eine DATA.indication-SDU handelt (Zeile 3). Ist dies nicht der Fall kann das Problem gar nicht auftreten und die SDU wird an den Layer 3 geschickt (Zeile 12). Handelt es sich jedoch um eine DATA.indication-SDU, kommt der neue if-Block zum Einsatz (Zeilen 5 – 10). Der Inhalt des if-Blocks wird nur ausgeführt, wenn sich im DropPanel eine Message befindet. Damit wird gewährleistet, dass der kritische Aufruf sduDropPanel.getMessage().getData() in Zeile 8 garantiert nicht in einer NullPointerException endet.

3.3 Ergebnis der Fehlerkorrektur

Performance	Exceptions	Fehler	Erweiterungen

Auf GUI-Ebene wird sichergestellt, dass das BitStuffing und BitUnstuffing nur einmal pro Message ausgeführt werden kann. Es wird sowohl im InspectorPanel (Eingehende Nachrichten) wie auch im ComposerPanel (Ausgehende Nachrichten) die neue Boolean-Variable btnClicked eingeführt.

```
 1 private void createNewMessage(int typ){
 2    btnClicked = false;
 3
 4    middlePanel.removeAll();
 5    middlePanel.setBorder(BorderFactory.createEmptyBorder(3, 0, 1, 1));
 6
 7    // Panel unten
 8    bottomPanel = new JPanel();
 9    bottomPanel.setLayout(new BorderLayout());
10
11    // Panel mit den Buttons unten rechts
12    JPanel btnPanel = new JPanel();
13    btnPanel.setLayout(new GridLayout(1,2));
14
15    // bitstuff-button mit listener
16    JButton bitStuffBtn = new JButton("Bitstuffing");
17    bitStuffBtn.addMouseListener(new MouseAdapter(){
18          @Override
19          public void mouseReleased(MouseEvent arg0) {
20                if(!btnClicked){
21                      btnClicked = true;
22                      bitStuffBtnClicked();
23                }
24          }
25    });
26 ...
```

Repräsentativ für beide Implementationen zeigt der obige Code-Ausschnitt den Anfang der createNewMessage()-Methode des ComposerPanels. Wird eine neue Nachricht erstellt, wird die btnClicked-Variable auf „false“ gesetzt (Zeile 2). Beim Anklicken des BitStuffing -Buttons wird die Methode mouseReleased() ausgeführt (Zeilen 19 – 23). Mit der if-Abfrage in Zeile 20 kann entschieden werden, ob die BitStuffing -Methode noch ausgeführt werden muss. Ist dies der Fall, wird die btnClicked-Variable auf „true“ gesetzt und das BitStuffing ausgeführt. Auf dem Wert „true“ bleibt sie solange bis wieder eine neue Nachricht erstellt wird. Dann wird die btnClicked-Variable wieder auf „false“ gesetzt und der Ablauf wiederholt sich.

Ein weiterer Fehler wurde während der Analyse noch nicht bemerkt. Erst im Verlauf der Funktionstests trat sporadisch eine ConcurrentModificationException auf. Diese Exception wird geworfen, falls zwei verschiedene Threads mit derselben Collection arbeiten wollen. Als Beispiel: Thread A durchläuft eine Liste mittels Iterator. Während der Arbeit erhält Thread B Rechenzeit und möchte dabei ein neues Objekt in die erwähnte Liste einfügen. Da ein Einfügen den Iterator unbrauchbar machen würde, wird eine ConcurrentModificationException geworfen.

Tests ergaben, dass es sich um zwei Listen im SimBuildDispatcher handelt: mInactiveProcessorlst und mActiveProcessorlst. Darin werden die momentan inaktiven respektive aktiven Prozessoren

aufgeführt. Während des Dispatchings führt der SimBuildDispatcher eine Neuorganisation dieser Listen durch. Dabei wird durch die mInactiveProcessorlst iteriert und gegebenenfalls einzelne Prozessoren in die mActiveProcessorlst verschoben. Es kann vorkommen, dass einer der SimProzessoren Rechenzeit erhält bevor diese Arbeit abgeschlossen ist. Falls dieser andere Thread seine Aufgabe beendet, versucht er sich in die mInactiveProcessorlst zu verschieben. Das Ergebnis ist die erwähnte Exception.

Um dieses Situation zu vermeiden, wird der kritische Bereich mit synchronized-Blöcken abgesichert:

```
1 synchronized (mActiveProcessorlst){
2 synchronized (mInactiveProcessorlst){
3     mInactiveProcessorlst.sort();
3     Iterator it = mInactiveProcessorlst.iterator();
4
5     while (it.hasNext()) {
6           tmpProcessor = (simProcessor) it.next();
7           waitingTime = tmpProcessor.getWaitingTime();
8 ...
```

Solange sich der Programmfluss innerhalb der synchronized-Blöcke befindet, ist sichergestellt, dass kein anderer Thread auf die angegebenen Variablen zugreifen kann. In diesem Fall sind dies die beiden Listen mInactiveProcessorlst und mActiveProcessorlst (Zeilen 1 und 2). Diese Massnahme hat jedoch auch einen Nachteil. Jeder Thread der in dieser Zeit auf die Listen zugreifen will, ist blockiert und kann nicht weiterarbeiten. Da aber keine rechenintensiven Operationen innerhalb der synchronized-Blöcke durchgeführt werden, ist das vernachlässigbar.

3.4 Ergebnis der Erweiterungen

Performance	Exceptions	Fehler	Erweiterungen

Wie in der Analyse erwähnt, zeigen sich die Vorteile der neuen Kontrollstruktur bei der Implementierung des „Level Restart“-Buttons und des „Level Select“-Dropdown-Menüs. Bei diesen beiden Operationen muss dem GameManager lediglich die gewünschte Spielstufe mitgeteilt werden. Daraufhin kann dieser den Level laden und die Simulationsumgebung neu starten.

Zur Realisierung muss der Controller im Wesentlichen um zwei Methoden ergänzt werden:

```
1 public void restartLevel(){
2     gm.setLevelCode(RESTARTLEVEL);
3     getDispatcher().terminateSimulation();
4 }
5
6 public void selectLevel(int levelNumber){
7     gm.setLevelCode(levelNumber);
8     getDispatcher().terminateSimulation();
9 }
```

Diese beiden Methoden werden durch die entsprechende Operation auf der Benutzeroberfläche aufgerufen. In den Zeilen 2 und 7 ist zu erkennen wie der LevelCode des GameManager gesetzt wird. Das geschieht mittels dessen setLevelCode()-Methode. Dem GameManager ist jetzt das nächste zu ladende Level bekannt. Anschliessend kommt die neue Methode terminateSimulation() aus dem Performance-Refactoring zum Zug. Die Simulation wird heruntergefahren und der Programmfluss

kehrt in den GameManager zurück. Dort wird der vorhin gesetzte LevelCode ausgewertet und der entsprechende Level wird geladen.

Die nächste Erweiterung ist die Implementierung des Hint-Systems. Die Anforderungen sind:

- Beliebig viele Hints integrierbar
- Jedes Level hat eigene Hints
- Multi Language Support

Zur Realisierung wird wie schon beim Level Select mit einem Dropdown-Menü gearbeitet. Dieses wird ebenfalls in die Toolbar integriert. Die verfügbaren Hints des Levels können dann bequem ausgewählt werden. Um einen Multi Language Support zu garantieren wird mit Textfiles gearbeitet. In der XML-Datei mit allen Level-Informationen kann optional ein Textfile mit den Hints angegeben werden. Dabei entspricht jede Zeile einem eigenen Hint. Durch dieses System kann bei Bedarf pro Level jeweils ein Textfile pro Sprache erstellt werden.

Da mit den Hints zusätzliche Level-Parameter hinzukommen, muss die Level-Klasse entsprechend ergänzt werden. Dies erfolgt analog zur bestehenden Integration der zu versendenden Texte: Es wird im Level-Objekt lediglich der Dateiname der Textdatei gespeichert. Das eigentliche Auslesen wird vom Controller übernommen, wenn er den Dateinamen erhält. Dazu ist eine neue Methode im Controller nötig:

```
 1 private Vector<String> generateHints(String file){
 2    Vector<String> hints = new Vector<String>();
 3    BufferedReader buf;
 4    String line ="";
 5
 6    try{
 7          buf = new BufferedReader(new FileReader(file));
 8          while((line = buf.readLine())!= null){
 9                hints.add(line);
10          }
11    } catch (Exception e) {
12          hints = null;
13    }
14    return hints;
15 }
```

Die Methode erhält den Dateinamen (Zeile 1). In Zeile 7 wird mittels FileReader der Inhalt der Hint-Datei ausgelesen und im BufferReader gespeichert. Da im Textfile pro Zeile ein Hint steht, kommt die Methode readLine des Buffers zum Einsatz (Zeile 8). Jeder gefundene Hint wird dabei in einen String-Vektor geschrieben (Zeile 9). Im Code-Ausschnitt nicht ersichtlich ist, dass der Vektor nach dem Erstellen in einer Variablen des Controllers gespeichert wird. Somit hat das GUI die Möglichkeit die Daten auszulesen und darzustellen.

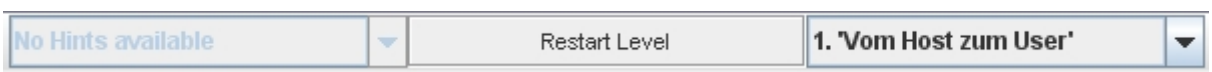

Abb. 12 - Die neuen Toolbar-Elemente: Hints-Dropdown, Restart-Button, Level-Select-Dropdown

Die letzte Erweiterung betrifft den SendBuffer. Gefordert ist ein Kontextmenü-Eintrag zum nochmaligen Senden eines bereits verschickten Pakets. Dies kann zum Beispiel nötig sein, wenn ein Paket auf dem Weg zum Kommunikationspartner verloren ging. Um diese Anforderung umsetzten zu

können, wird die neue Klasse ResendMessage geschrieben. Sie ist von der Swing-Klasse AbstracAction abgeleitet. Sie stellt eine individuelle Action für den Resend-Kontextmenü-Eintrag dar:

```
 1 public class ResendMessage extends AbstractAction {
 2
 3    private OutputBuffer buffer;
 4    private Message message;
 5
 6    public ResendMessage(String text,OutputBuffer buffer, Message
 7                            message) {
 8          super(text);
 9          this.buffer = buffer;
10          this.message = message;
11    }
12
13    public void actionPerformed(ActionEvent arg0) {
14          buffer.resend(message);
15    }
16
17 }
```

Jeder einzelne Eintrag im Send Buffer des GUIs erhält solch eine Action. Bei der Konstruktion ist es deshalb möglich, gleich die entsprechende Message mitzugeben (Zeile 7). Diese wird im Action-Objekt gespeichert (Zeile10). Wird auf der Benutzeroberfläche bei einem Eintrag „Resend Message“ ausgewählt, wird die actionPerformed-Methode() ausgeführt (Zeilen 13 – 15). Dem Buffer wird dann mitgeteilt, dass er die gespeicherte Message nochmals senden soll (Zeile 14).

4 Schlussfolgerung

Performance	Exceptions	Fehler	Erweiterungen

Die zu Beginn der Diplomarbeit geplanten Arbeitspakete konnten alle erledigt werden. Durch die stark verbesserte Performance ist jetzt ein normaler Einsatz des Lernspiels möglich. Auch wenn der HDLCSimulator eine Stunde lang läuft, treten noch immer keine Geschwindigkeits-Einbussen auf. Das Überarbeiten des Exception-Handlings sorgt für eine bessere Wartbarkeit. Die Vereinheitlichung macht weitere Updates angenehmer. Zusätzlich wird bei einer Exception der Benutzer informiert und die Applikation komplett beendet. Bis anhin konnte der Benutzer das GUI weiter gebrauchen, es folgte jedoch keine Reaktion der Simulationsumgebung mehr. Durch das verbesserte Exception-Handling und die Korrektur verschiedener Fehler konnte die Applikation stabiler gemacht werden. Somit kann ein flüssigerer Programmablauf garantiert werden. Die implementierten Erweiterungen erhöhen den Benutzungskomfort enorm. Neben der verbesserten Performance tragen sie am meisten dazu bei, dass der HDLCSimulator in Zukunft sinnvoll eingesetzt werden kann.

Insgesamt gesehen ist der HDLCSimulator in seiner jetzigen Version bereit, im Unterricht als Lernhilfe eingesetzt zu werden.

4.1 Punkte für eine nächste Version

Am HDLCSimulator konnte im Verlaufe dieser Diplomarbeit vieles verbessert werden. Dennoch gibt es immer noch einige Verbesserungspunkte. Die meisten davon tauchten während dem Einsatz in den Übungsstunden auf. Diese Übungen fanden leider erst in den letzten zwei Tagen vor der Abgabe der Diplomarbeit statt. Somit konnten die Verbesserungsvorschläge nicht mehr in die Applikation einfliessen. Damit das eingegangene Feedback nicht verloren geht, wird es in diesem Unterkapitel festgehalten.

- Das Hint-System besteht zwar, die Hints selbst müssen aber noch geschrieben werden.
- Für die Befehle (SABM, RR, usw.) wären Tooltips sinnvoll. Diese könnten eine kurze Beschreibung liefern, für was der entsprechende Befehl eingesetzt wird.
- Wenn ein Level erfolgreich beendet wurde, sollte eine Dialogbox angezeigt werden. Ein direkter Neustart des nächsten Levels kann für Verwirrung sorgen.
- Teilweise kommt es immer noch vor, dass die Simulationsumgebung sich aufhängt/abstürzt. Ein Klick auf „Restart“ nützt nichts, da die Simulation in diesem Zustand nicht heruntergefahren werden kann.
- SDUs und PDUs gehen zu schnell verloren. Wird ein Drag and Drop an eine unzulässige Stelle gemacht, verschwindet das Paket von seiner ursprünglichen Position. Beim drücken des „Senden“-Button im ComposerPanel können Daten verloren gehen. Dies passiert, wenn vergessen wurde, die Daten vom InspectorPanel ins ComposerPanel zu ziehen. Beim Klick werden die Panels geleert und das Daten-Paket ist weg.
- In JTOOPS fehlt eine Methode zum Zurücksetzen eines Timers. Jeder Timer kann nur einmal gebraucht werden.

5 Testing

Das Testen des HDLCSimulators erfolgt hauptsächlich in Form von funktionalen Tests. Da es sich bei dieser Diplomarbeit um ein Refactoring handelt, besitzt man von Anfang an eine lauffähige Applikation. Der Vorteil davon ist, dass jederzeit einige Levels des Lernspiels absolviert werden können. Dadurch hat man die Möglichkeit nach jeder Etappe der Überarbeitung die Funktionalität zu testen. Dies gilt sowohl für Refactoring wie auch für neu implementierten Features. Durch diese Methode geht man bei den Tests gleich vor wie später auch der Benutzer. Zu beachten ist, dass man ebenfalls fehlerhafte Benutzung miteinschliesst. Damit ist gemeint, dass man manchmal absichtlich falsche Aktion durchführt, um die Flexibilität und Stabilität der Applikation zu testen.

Wie im Unterkapitel 4.1 erwähnt, fand der Test mit mehreren Benutzern erst in den letzten zwei Tagen vor Abgabe statt. Einige Testergebnisse sind dort bereits erwähnt. Es wurde eine Umfrage betreffend der Benutzung des HDLCSimulators verschickt. Wenn die Ergebnisse der Umfrage eintreffen werden diese noch nachgereicht. Sie werden einen wertvollen Beitrag zur Weiterentwicklung des HDLCSimulators leisten.

6 Glossar

Begriff	Erklärung
Drag and Drop	Bezeichnung für das Bedienen einer Benutzeroberfläche mit der Maus. Dabei können Objekte der Oberfläche gepackt und gezogen (drag) und an einem anderen Ort wieder losgelassen (drop) werden.
GUI	Graphical User Interface. Grafische Benutzeroberfläche.
HDLC	High-Level Data Link Control. Ein Protokoll der Sicherungsschicht im OSI-Modell. Zentrales Thema dieser Diplomarbeit.
JTOOPS	Java Tool for Objectoriented process simulation. Framework für das Erstellen von Simulationen. Dient dem HDLCSimulator als Grundlage.
OSI-Modell	Modell zur Datenübertragung zwischen Computersystemen. Es beschreibt sieben aufeinander aufbauende Abstraktionsschichten mit jeweils definierten Aufgaben und Schnittstellen.
PDU	Protocol Data Unit. Diese Pakete werden verwendet, wenn Daten mit dem Kommunikationspartner ausgetauscht werden müssen.
SDU	Service Data Unit. Diese Pakete werden verwendet, wenn Daten mit der eigenen übergeordneten Schicht ausgetauscht werden müssen.
UML	Unified Modeling Language. Dient zur Modellierung von Software durch verschiedene Arten von Diagrammen.
XML	Extensible Markup Language. Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien.

7 Abbildungsverzeichnis

8 Literaturverzeichnis

1. **Wikipedia.** http://de.wikipedia.org/wiki/OSI-Modell. [Online]

2. —. http://de.wikipedia.org/wiki/HDLC. [Online]

3. **Lazar, Andreas und Keller, Fabian.** *Diplomarbeit HDLCSimulator.* 2007.

4. **JProbe.** http://www.quest.com/jprobe/. [Online]

5. **Pfister, Tobias und Küderli, Yves.** *Diplomarbeit Re-Design von JTOOPS.* 2002.