Das interaktive Skript

Automatische Überprüfung und Hilfestellung
zu Vorlesungs–begleitenden Übungen

Software–Dokumentation

*Check*Applets*

Norman Hendrich
Universität Hamburg
Fachbereich Informatik

Universität Hamburg

E-LEARNING CONSORTIUM Hamburg

Inhaltsverzeichnis

Kontakt:
Prof. Dr. Klaus von der Heide
Universität Hamburg
Fachbereich Informatik
Vogt-Kölln-Str. 30
D 22 527 Hamburg

http://tams-www.informatik.uni-hamburg.de/forschung/interaktives-skript/

1 Einführung

Die vorliegende Software-Dokumentation beschreibt die im Rahmen des Projekts „Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungen“ des Hamburger Sonderprogramms E-Learning (ELCH) entwickelten Java-Applets zur Überprüfung von Übungsaufgaben. Zusammengenommen decken die Applets eine Vielzahl der typischen Aufgabentypen in der technischen Informatik ab, sollten sich aber darüber hinaus auch für Einsatz in vielen anderen technisch/naturwissenschaftlichen Fachgebieten nutzen lassen.

Anders als die bisherigen Projektberichte dient die vorliegende Dokumentation nicht nur zur Beschreibung der Konzepte und Architekturen, sondern gleichzeitig als Tutorial und als Spezifikation der Applets. Als Zielgruppe sind alle Lehrkräfte und Software-Entwickler angesprochen, die am Einsatz von interaktiven Übungen in Web-basierten Umgebungen oder E-Learning Plattformen interessiert sind. Die Beschreibungen setzen lediglich Grundkenntnisse in HTML und dem Aufruf von Java-basierten Programmen voraus. Die Erweiterung um zusätzliche selbstentwickelte Applets ist problemlos möglich, erfordert aber fundierte Java-Kenntnisse.

Einführung und Umfeld

Als Einführung fasst der folgende Abschnitt 1.1 noch einmal kurz die wesentlichen Aspekte eines *interaktiven Lehrbuchs* zusammen, während Abschnitt 1.2 die Einbettung von Übungsaufgaben motiviert. In Abschnitt 1.3 wird noch einmal die in der ersten Projektphase erarbeitete Klassifikation der Übungsaufgaben wiederholt. Für eine ausführliche Diskussion der mit den interaktiven Lehrbüchern verfolgten Konzepte sei auf den ersten Projektbericht [11] verwiesen.

*Check*Applets Dokumentation*

Kapitel 2 beschreibt die Konzepte und Eigenschaften der Check*Applets. Die ersten Abschnitte erläutern die Motivation für die Entwicklung der Applets, die gewählte Programmierschnittstelle und die bisher realisierte Klassenhierarchie. Außerdem werden Erweiterungsmöglichkeiten und die Anpassung an weitere Sprachen diskutiert.

Ab Abschnitt 2.9 beginnt die detaillierte Beschreibung der einzelnen bisher implementierten Applet-Klassen. Für jedes Applet wird zunächst das Konzept erläutert, bevor anschließend alle verfügbaren Parameter besprochen werden. Ein konkretes Codebeispiel zeigt den möglichen Einsatz. Die entsprechenden Abschnitte dienen zur Vertiefung und sind weitgehend voneinander unabhängig. Sie können beim Lesen ohne weiteres übersprungen werden.

1.1 Rahmen des Projekts: Das interaktive Lehrbuch

Konzept

Ein *interaktives Lehrbuch* bzw. *interaktives Skript* vereinigt die textuelle Beschreibung der zu lernenden Zusammenhänge und Sachverhalte mit interaktiven elektronischen Werkzeugen zur Darstellung und Anwendung dieser Sachverhalte — und zwar der Anwendung nicht nur auf die im Lehrbuch selbst integrierten Beispiele, sondern vielmehr auf beliebige, vom Lernenden jederzeit selbst veränderbare oder hinzugefügte Anwendungsfälle. Dadurch wird das Lehrbuch erweiterbar und kann auch an ursprünglich gar nicht vorgesehene oder vorhersehbare zukünftige Entwicklungen angepasst werden. Es bietet damit ideale Voraussetzungen zur Unterstützung des *life-long learning* und zum produktiven dauerhaften Einsatz während des Berufslebens.

Das dem Projekt zugrunde liegende Konzept des interaktiven Lehrbuchs hat folgende Zielsetzungen:

Problemlösung

- Die Spanne zwischen klassischem Lehrbuch und Anwendung wird zunehmend größer, weil die Komplexität der behandelten Themen sich nur noch mit Rechnergestützten Systemen beherrschen lässt. Durch die harmonische Integration von klassischem Lehrtext in ein zugrunde liegendes, universelles Softwaresystem zur Problemlösung kann das interaktive Lehrbuch sehr anwendungsspezifisch werden, ohne dabei jedoch auf ein Anwendungsgebiet festgelegt zu sein.

Nachhaltigkeit

- Der Rückblick auf die letzten Jahre der Softwareentwicklung zeigt, dass für nachhaltige Entwicklungen nur einfache und standardisierte Datenformate verwendet werden sollten. Beim Einsatz proprietärer Formate muss jederzeit damit gerechnet werden, nach einem Versionswechsel auf ältere Datensätze nur noch eingeschränkt oder eventuell überhaupt nicht mehr zugreifen zu können. Inhalte für das interaktive Lehrbuch basieren daher im Wesentlichen auf annotierten Texten im einfachen ASCII-Format.

Anpassbarkeit

- Ein klassisches Lehrbuch spiegelt immer nur die Sicht (und Absicht) des jeweiligen Autors wider. Um sich ein objektiveres Bild zu verschaffen, greifen viele Studierende und Lehrende deshalb parallel auf mehrere Lehrbücher zurück. Wegen des hohen Erstellungsaufwands im Falle von E-Learning Content stehen geeignete alternative Materialien bisher aber nur selten zur Verfügung. Daraus ergibt sich die Forderung, dass der Inhalt des interaktiven Lehrbuchs von den Lehrenden individuell nach ihren eigenen Vorstellungen ausgerichtet und geändert werden kann — und zwar mit Hilfe eines langfristig und auf allen Plattformen verfügbaren Werkzeugs. Dies betrifft sowohl die Auswahl als auch die Inhalte der Texte, Abbildungen, Animationen, Audioausgaben, Simulationen, usw.

Exploration

- Die Integration von interaktiven Elementen in das interaktive Lehrbuch hilft dabei, Interpretationsschwierigkeiten oder Verständnislücken durch aktive Exploration des Lehrstoffs zu überwinden. Viele Inhalte und insbesondere Graphiken im interaktiven Lehrbuch werden deshalb erst zur Laufzeit mit vom Benutzer individuell einstellbaren Parametern erzeugt und angezeigt. Ein Studierender kann auf diese Weise grundsätzlich nachvollziehen, wie es zu den Graphiken und Ergebnissen kommt.

1.2 Kurzbeschreibung des Projekts

Integrierte Übungen

Es liegt nahe, auch *Übungsaufgaben* in das oben beschriebene interaktive Lehrbuch zu integrieren. Während eine derartige Integration bei klassischer Lernsoftware im Sinne des *Computer Based Training* [27] sehr aufwendig sein kann, stellt das interaktive Lehrbuch mit seiner Softwareplattform bereits alle Werkzeuge zur Verfügung, um die Lernenden bei der Bearbeitung der Aufgaben zu unterstützen und zu motivieren.

Überprüfung und Hilfestellung

Die Erforschung und Erprobung dieser Techniken ist der Inhalt des vorliegenden Projekts. Ziel ist eine Softwarebibliothek, die für viele Typen von Übungsaufgaben eine automatische Überprüfung der Lösungen erlaubt und bei Fehlern geeignete Hilfestellung anbietet. Die Studierenden bekommen dadurch sofort eine Rückmeldung über ihren Lernfortschritt bzw. Hinweise auf noch verbliebene Fehler. Zusammen mit der nahtlosen Integration der Aufgaben in das Skript wird die Hemmschwelle zur Bearbeitung der Übungsaufgaben gesenkt, was die Studierenden zu einer intensiveren Beschäftigung mit dem Lehrstoff einlädt.

Methodenkompetenz

Das primäre Ziel der Übungen ist dabei nicht die Vermittlung von Faktenwissen, sondern die Verbesserung der Fertigkeiten bei der Auswahl und dem Einsatz von Methoden. Die im Projekt erzielten Ergebnisse werden sich deshalb auch auf viele andere Fachgebiete mit ähnlicher Methodik übertragen lassen, etwa die angewandte Mathematik, Experimentalphysik oder die Ingenieurwissenschaften.

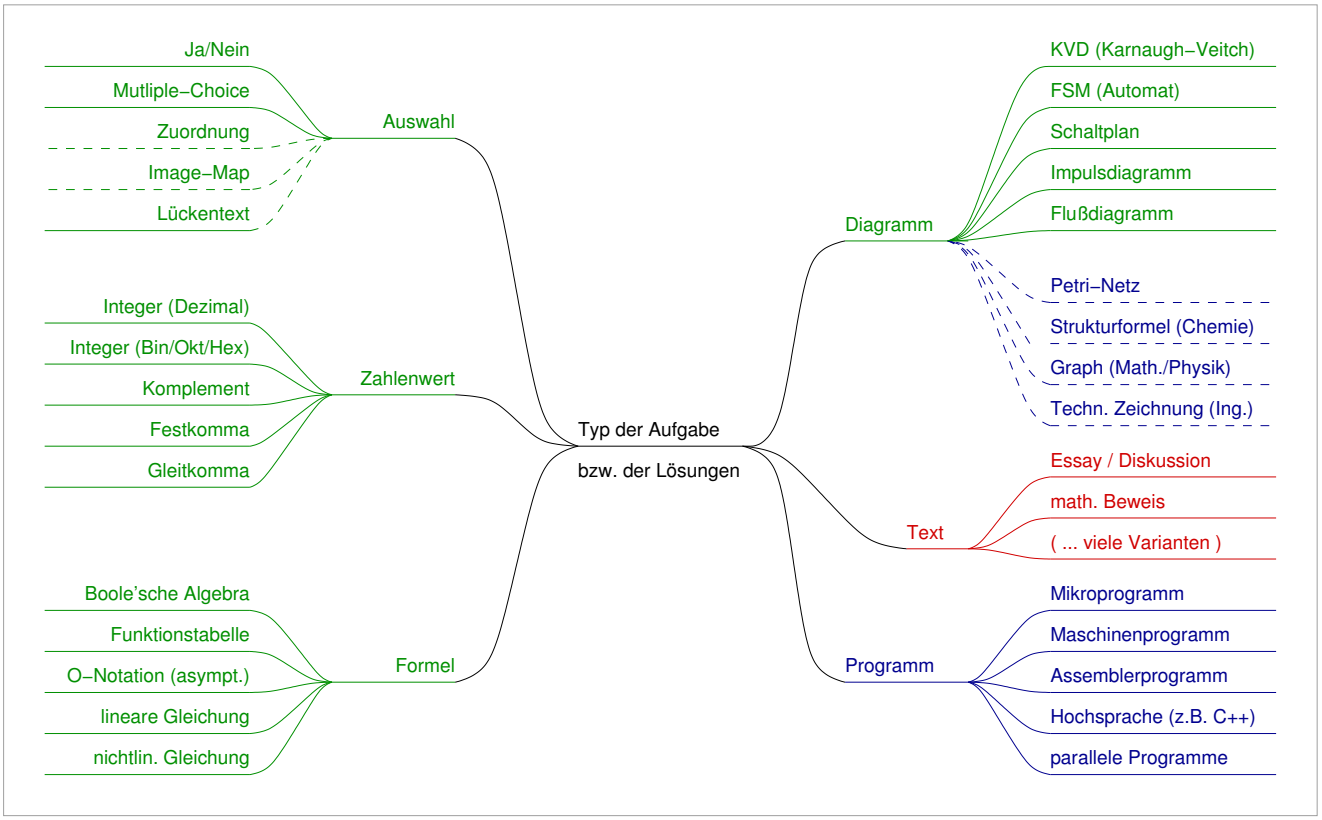

Abbildung 1: *Klassifikation der Übungsaufgaben zur technischen Informatik. Die sechs Hauptklassen dürften auch auf andere mathematisch naturwissenschaftliche Fächer übertragbar sein. Für fast alle Kategorien bis auf freie Texte ist eine automatische Überprüfung möglich.*

1.3 Klassifikation der Übungsaufgaben

An dieser Stelle ist es notwendig, die in der ersten Projektphase erarbeitete Klassifikation der Übungsaufgaben zur technischen Informatik noch einmal zu wiederholen, da die späteren Kapitel häufig auf diese Klassifikation zurückgreifen. Die Auswertung mehrerer Vorlesungsskripte und klassischer Lehrbücher ergab die in Abbildung 1 dargestellte Einteilung in sechs große Klassen von Aufgabentypen. Diese Klassen von Aufgaben dürften sich auch auf die meisten anderen technisch-naturwissenschaftlichen Fächer übertragen lassen, allerdings eventuell mit anderer Gewichtung und Häufigkeit der einzelnen Aufgabentypen.

Wie bereits im ersten Projektbericht erläutert wurde, ist es beim derzeitigen Stand der Technik zur Texterkennung und Sprachverarbeitung nicht einmal ansatzweise möglich, von den Studierenden eingesandte freie Texte sinnvoll auszuwerten [11]. Im Rahmen des Projekts wird diese Kategorie deshalb von vornherein ausgeklammert; derartige Aufgaben müssen wie bisher von den Übungsgruppenleitern von Hand korrigiert werden. Statt dessen ist geplant, die Textaufgaben zumindest soweit möglich durch gleichwertige Fragestellungen zu ersetzen, die der automatischen Überprüfung besser zugänglich sind.

2 Die Check*Applets

Um die im Rahmen des Projekts entwickelten Algorithmen nicht nur auf die dem interaktiven Skript zugrundeliegende Plattform Matlab zu beschränken, wurde rechtzeitig für das Wintersemester 2004 mit der Entwicklung von zusätzlichen Java-Applets zur Überprüfung von Übungsaufgaben begonnen. Dieses Kapitel beschreibt die Eigenschaften dieser *Check*Applets* und dient als Softwaredokumentation zu den einzelnen Applets.

Die folgende Aufzählung nennt die wesentlichen Überlegungen bei der Entwicklung der Check*Applet-Klassen. Nicht zuletzt ergeben sich durch die einfache Einbettung in gewöhnliche Web-Seiten oder auch E-Learning Plattformen breitere Einsatzmöglichkeiten:

- Einsatz rechtzeitig noch im Wintersemester 2004 in den Übungen zur Vorlesung Technische Informatik I.

  Da die Beschaffung des Matlab Compilers erst Ende November 2004 abgeschlossen werden konnte (vgl. [13]), kam ein Einsatz der vorcompilierten Matlab-Programme für die Übungen nicht mehr in Frage.

- Einfache Anpassung der Applets an die vorhandenen Übungsaufgaben und Musterlösungen aus vorangegangenen Semestern. Die ursprünglich geplante Umformulierung der Übungsaufgaben, um eine möglichst gute Überprüfbarkeit zu erreichen, konnte aus Zeitgründen noch nicht umgesetzt werden.

- Minimale Downloadgröße. Bei einer Gesamtgröße von ca. 180 KByte eignen sich die Check*Applets ohne Einschränkung auch für Modemverbindungen. Demgegenüber beträgt die Downloadgröße für die Matlab Component Runtime in Version 7.1 fast 90 MByte, und auch unsere Jython-Console benötigt immerhin noch über 2 MByte.

- Daher Beschränkung auf Java-Applets ohne externe Abhängigkeiten als Softwareplattform. Derartige Applets können derzeit auf allen gängigen Betriebssystemen und mit allen gängigen Web-Browsern ohne weitere Installation ausgeführt werden.

- Entwicklung einer Klassenhierarchie von flexiblen Applets, mit denen alle wesentlichen Aufgabentypen abgedeckt werden. Beschränkung auf einfache Schnittstellen, so dass die spätere Erweiterung um zusätzliche Applets und die Anpassung etwa an andere Sprachen jederzeit einfach möglich bleibt. Parallel dazu die Erstellung von Webseiten mit diesen Applets für die vorhandenen Übungsaufgaben.

- Bereitstellung eines flexibel konfigurierbaren Applets zur Stringüberprüfung als „Notnagel“ für alle Aufgabentypen, für die nicht rechtzeitig spezialisierte Applets fertiggestellt werden konnten. Trotzdem Integration von brauchbaren Fehler- bzw. Statusmeldungen als Hilfestellung für die Studierenden.

Aber keine interaktive Umgebung

Der Vorteil der kompakten Codegröße wird mit einem Nachteil erkauft — dem Verzicht auf eine vollständige interaktive Programmierumgebung. Die einzelnen Applets sind zwar weitgehend konfigurierbar und durch Ableiten von Unterklassen erweiterbar, verfügen jedoch nicht über die Möglichkeit, vom Benutzer eingegebenen Code interaktiv auszuführen. Falls dies erforderlich ist, können die Applets natürlich problemlos in unsere Matlab- oder Jython-Umgebungen integriert und von dort aus aufgerufen werden.

Übersicht

Die nächsten Abschnitte erläutern zunächst die Motivation und die Softwarekonzepte für die Check*Applets, inklusive einer kurzen Beschreibung der Programmierschnittstelle und der Klassenhierarchie. Ab Seite 12 werden anschließend die einzelnen, bisher fertig implementierten Applets vorgestellt. Für jedes Applet wird zunächst das geplante Einsatzgebiet

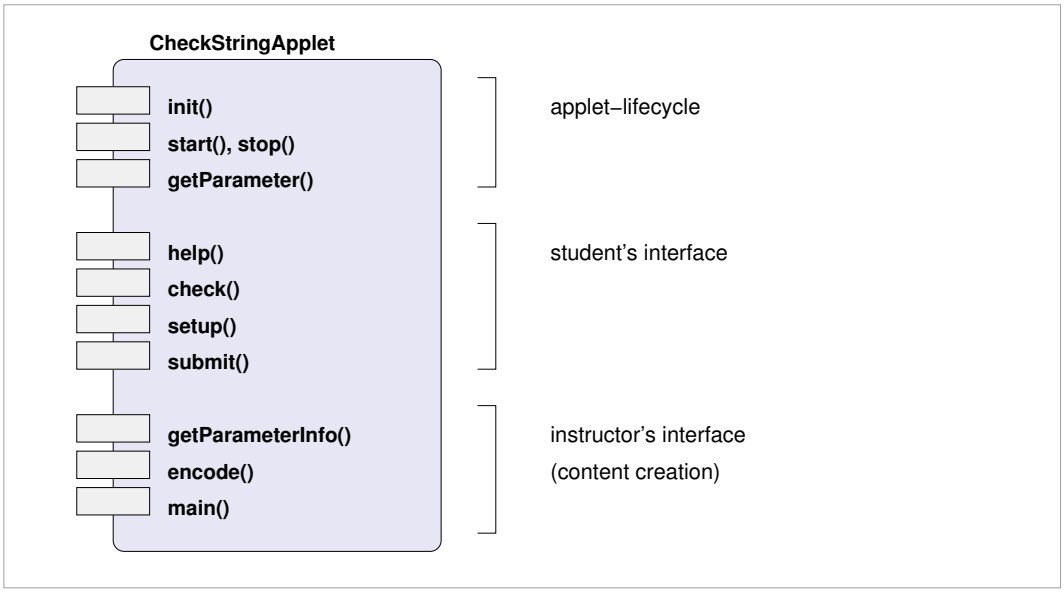

Abbildung 2: *Die Programmierschnittstelle der Check*Applets gliedert sich in drei Gruppen von Funktionen, wobei die erste Gruppe der Anbindung an den Webbrowser dient. Die zweite Gruppe von Funktionen leistet die eigentliche Überprüfung und die Interaktion mit dem Anwender, während die dritte Gruppe die Content-Erstellung und Berechnung der notwendigen Applet-Parameter unterstützt.*

skizziert, bevor die verfügbaren Parameter im Detail erläutert werden. Auf der jeweils gegenüberliegenden Seite steht ein Screenshot mit einer Abbildung des Applets und darunter der zugehörige HTML-Quelltext mit den Parametern für das dargestellte Applet.

Die Applets wurden im Wintersemester 2004 erstmals für die Übungen zu unserer Vorlesung Technische Informatik I [4] eingesetzt. Die Webseiten für die Vorlesung und Übungen sind frei zugänglich und eignen sich damit auch zur Demonstration der Check*Applets in ihrem eigentlichen Anwendungsgebiet. Zum Ausführen der Applets wird lediglich ein aktueller Webbrowser mit installierter Java Virtual Machine (JDK/JRE 1.4.2 oder höher empfohlen). Bitte besuchen Sie die Webseite für die Übungen:

`http://tams-www.informatik.uni-hamburg.de/lehre/ws2004/t1Uebung/`

Folgen Sie den *Selbstkontrolle*-Hyperlinks, um die Webseiten mit den Check*Applets zur Überprüfung der einzelnen Aufgabenblätter zu laden.

2.1 Programmierschnittstelle

Die grundlegende Programmierschnittstelle der Check*Applets gliedert sich in drei Gruppen von Funktionen, siehe Abbildung 2. Die erste Gruppe wird von den Standardfunktionen zur Interaktion der Applets mit dem Webbrowser gebildet.

Die zweite Gruppe umfasst nur vier sorgfältig ausgewählte Funktionen, die zur Interaktion mit dem Anwender dienen. Die Funktion `setup()` dient zur Konfiguration der Benutzereinstellungen (Name, Matrikelnummer, Übungsgruppenleiter, usw.), während `help()` eine Hilfe zum jeweiligen Applet und der erwarteten Eingabe anbietet. Das Kernstück des Applets bildet die Methode `check()`, mit der die eigentliche Überprüfung der Eingabe ausgelöst wird. Die bisher implementierten Applets liefern ihre Meldungen und Hilfestellungen über Dialogfenster, aber auch komplexere Lösungen sind denkbar. Schließlich erlaubt die Funktion `submit()` das automatische Einschicken der fertigen Lösung per Mail an den Übungsgruppenleiter.

Die dritte Gruppe von Funktionen kann als „Instructor's Interface“ bezeichnet werden und erlaubt die Anpassung der Applets an die jeweilige Aufgabenstellung, insbesondere also die Erstellung und Kodierung der notwendigen Applet-Parameter. Über die Methode `getParameterInfo()` können die für das jeweilige Applet verfügbaren Parameter auch zur Laufzeit ausgelesen werden, während die `main()` Methode die notwendigen Algorithmen zur Kodierung der Parameter bereitstellt.

2.2 Content-Erstellung

Zur Einbettung der Check*Applets in Webseiten ist keine spezielle Software erforderlich. Alle gängigen HTML-Editoren erlauben die Erstellung von Applet-Marken und der zugehörigen `param`-Blöcke, um die Applets mit ihren spezifischen Parametern zu versorgen. Natürlich können die HTML-Seiten aber wahlweise auch mit jedem gewöhnlichen Texteditor erstellt werden.

getParameterInfo

Alle Check*Applet Klassen implementieren die Java-Methode `getParameterInfo()`, mit der sich eine Liste der für das Applet zur Verfügung stehenden Parameter zusammen mit den zugehörigen Defaultwerten und den Kurzbeschreibungen direkt aus dem Applet-Code selbst auslesen lässt. Sofern der gewählte HTML-Editor diese Funktion nicht unterstützt, sollten aber auch die Beschreibungen der einzelnen Applets auf den folgenden Seiten ausreichen, um die Applets über ihre Parameter zu konfigurieren und an neue Aufgaben anzupassen. Als Vorlage können die oben erwähnten Webseiten für die Übungen zur Vorlesung T1 dienen, die für (fast) alle Applets entsprechenden fertigen HTML-Code mit allen benötigten Parametern enthalten. Durch Kopieren der Applet-Blöcke lassen diese sich leicht in neue HTML-Seiten integrieren.

Codierte Parameter

Während die meisten Applet-Parameter direkt im Klartext eingegeben werden, ist für die Berechnung der mit der Base64-Kodierung oder mit ähnlichen Algorithmen „verschlüsselten“ Parameter natürlich eine Softwareunterstützung notwendig. Um eine skalierbare Architektur zu gewährleisten, sind die Funktionen zum Erzeugen der kodierten Parameter direkt über das jeweilige Applet selbst zu erreichen. Realisiert wird dies durch eine eigene `main()` Methode in den Applets. Beim Aufruf ohne weitere Parameter geben die Applets eine kurze Hilfe mit der Liste der unterstützten Parameter aus. Die meisten Applets bieten den Parameter `-encode` an, um die auf der Kommandozeile als Klartext übergebene Musterlösung in das jeweilige Format umzurechnen.

Editor

Der geplante Editor zum Auslesen der Applet-Parameter über die getParameterInfo-Funktion und anschließender Präsentation eines User-Interface zur Auswahl bzw. zum Eingeben der ermittelten Werte konnte aus Zeitgründen leider noch nicht fertiggestellt werden. Nur für die Klasse *CheckFormulaApplet* mit ihren komplexen Parametern wie *matrix* steht bereits die Klasse *WriteFormulaApplet* zur Verfügung, die den vollständigen HTML-Code zur Konfiguration eines solchen Applets automatisch aus den angegebenen Kommandozeilen-Parametern erzeugen kann.

2.3 Klassenhierarchie

Die Klassenhierarchie der bisher implementierten Check*Applets ist in Abbildung 3 für die „einfachen“ und in Abbildung 4 für die „komplexen“ Applets skizziert. Zentrale Bedeutung hat die Klasse *CheckStringApplet*, die als Basisklasse die gemeinsamen Grundfunktionen aller übrigen Applets inklusive User-Interface, Konfiguration und Option zum Mailversand zur Verfügung stellt.

Während CheckStringApplet die Überprüfung von Texteingaben gegen vorgegebene Musterlösungen erlaubt, steht *CheckMultipleChoiceApplet* stellvertretend für die Überprüfung

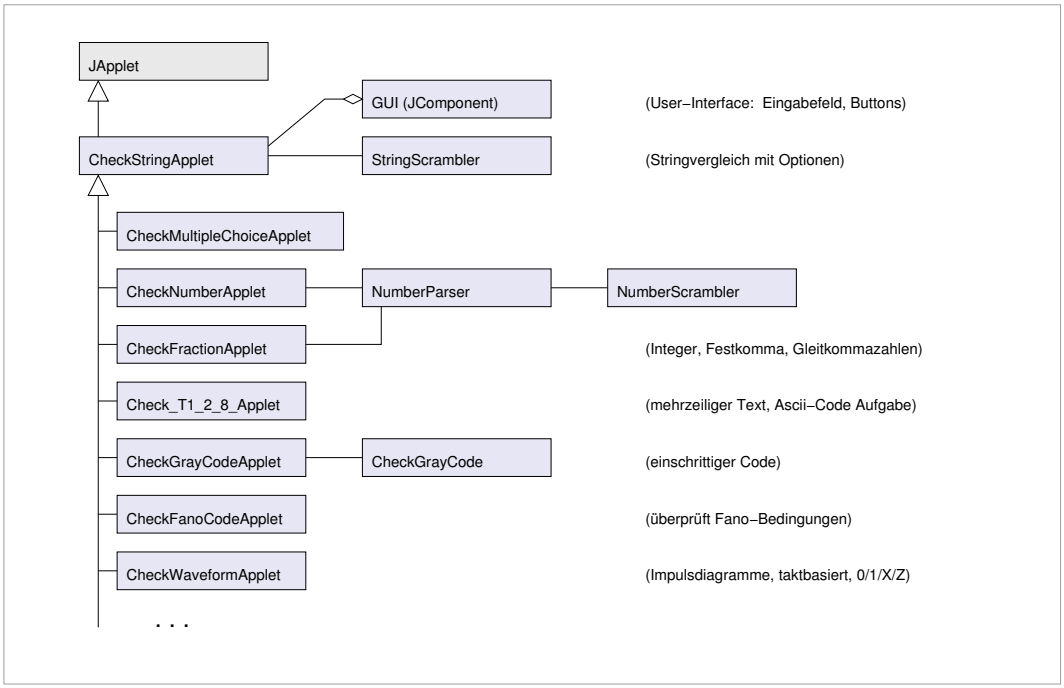

Abbildung 3: *Klassenhierarchie der Check*Applets (Teil 1). CheckStringApplet als Basisklasse dient zur Überprüfung der vom Anwender eingegebenen Texte mit fest vorgegebenen Musterlösungen. Weitere Applets wie CheckMultipleChoiceApplet oder CheckNumberApplet sind weitgehend konfigurierbar und dienen zur Überprüfung einer ganzen Kategorie von Aufgaben. Da CheckStringApplet die gemeinsamen Grundfunktionen inklusive Applet-Setup und graphischer Oberfläche fertig zur Verfügung stellt, ist aber auch die Entwicklung von spezialisierten Applets für jeweils nur eine einzige Aufgabe wie Check_T1_2_8_Applet problemlos möglich.*

von Auswahlaufgaben. Weitere Applets, etwa zum Anzeigen und Auswerten von Image-Maps oder Zuordnungsaufgaben wurden bisher noch nicht realisiert, da diese Art von Aufgaben in unserer Vorlesung und den Übungen nicht vorkommt.

Die Kategorie der Zahlenwertaufgaben wird durch *CheckNumberApplet* zur Überprüfung von Integer- und Gleitkommawerten sowie *CheckFractionApplet* für Festkommazahlen abgedeckt. Beide Klassen delegieren das eigentliche Parsen der Zahlenformate an die Hilfsklasse *NumberParser*, während *NumberScrambler* Funktionen zum Umkodieren von Zahlenwerten anbietet.

Die Klasse *Check_T1_2_8_Applet* liefert ein Beispiel für ein Check*Applet, das speziell nur für eine einzige Aufgabe entworfen wurde. Da die Musterlösung Sonderzeichen enthält, die sich nur schwer in HTML-Quellcode unterbringen lassen, wurde sie fest in das Applet hineincompiliert.

Alle bisher besprochenen Klassen verwenden lediglich das simple User-Interface von CheckStringApplet mit einem einzelnen Textfeld zur Eingabe der Lösung. Die Klasse *CheckWaveformApplet* erweitert das User-Interface um die interaktive, graphische Eingabe von Impulsdiagrammen, während die zugrundeliegende Überprüfung weiterhin auf dem direkten Vergleich mit der vorgegebenen Musterlösung basiert. Erst die beiden Klassen *CheckGrayCodeApplet* und *CheckFanoCodeApplet* verwenden aufwendigere Algorithmen, um die Eigenschaften der vom Anwender eingegebenen Binärcodes zu überprüfen.

Die in Abbildung 4 dargestellten und deutlich komplexeren Applets basieren ebenfalls auf CheckStringApplet und dienen zur Überprüfung von Aufgaben aus unserer Kategorie der *Formelaufgaben*. Zur (numerischen) Überprüfung von arithmetischen Ausdrücken

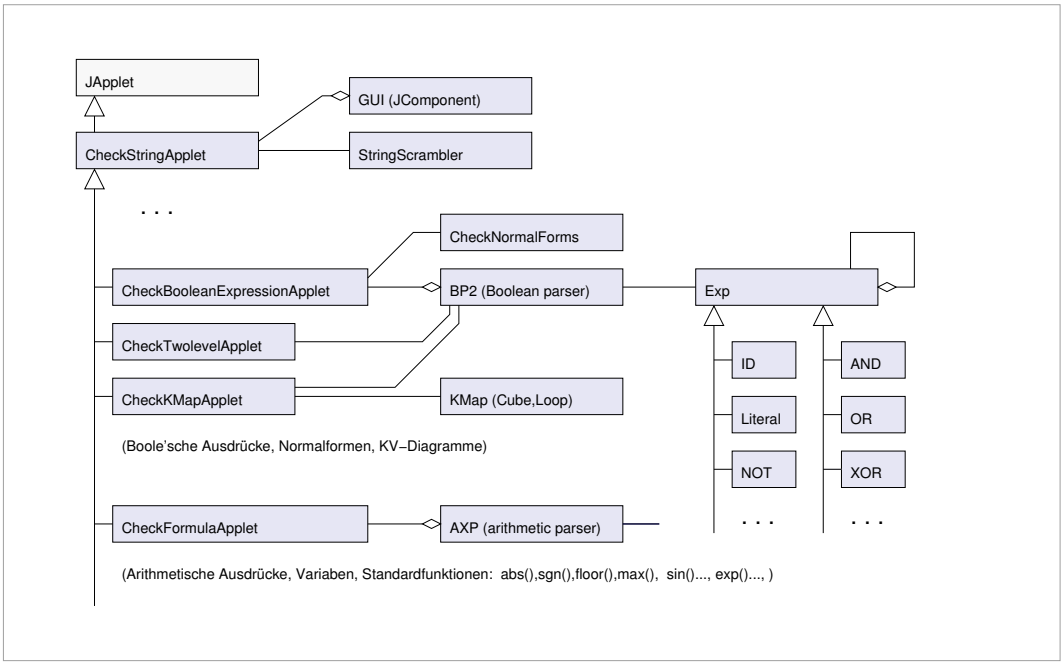

Abbildung 4: *Klassenhierarchie der Check*Applets (Teil 2). Ebenfalls auf der Basisklasse CheckStringApplet basieren CheckFormulaApplet zur Überprüfung von allgemeinen arithmetischen Ausdrücken sowie die verschiedenen Applets zur Überprüfung von Boole'schen Ausdrücken. Für die Auswertung dieser Ausdrücke kommen selbstentwickelte Parser* BP2 *für Boole'sche- und* AXP *für arithmetische Ausdrücke zum Einsatz, die ihren Parse-Baum mit inneren Klassen repräsentieren.*

und Formeln dient *CheckFormulaApplet*. Da im Kontext der Check*Applets keine interaktive Umgebung wie Matlab oder Jython zur Verfügung steht, dient ein eigener Parser *AXP* mit diversen Hilfsklassen zur interaktiven und effizienten Auswertung der vom Anwender eingegebenen Formeln. Ein entsprechender Ansatz wurde mit dem Parser *BP2* auch für die Auswertung von Boole'schen Ausdrücken gewählt, wobei die Hilfsklasse *CheckNormalForms* alle gängigen Normalformen überprüfen kann. Die Eingabe erfolgt bei *CheckBooleanExpressionApplet* in textueller Form, bei *CheckTwolevelApplet* als Schaltplan und bei *CheckKMapApplet* als KV-Diagramm.

2.4 Das Klartext-Problem

Auch für die Check*Applets ergibt sich das bereits im ersten Projektbericht für die interaktiven Skripte angesprochene „Klartext"-Problem. Damit die Applets möglichst universell eingesetzt werden können, sind sie über Parameter weitgehend konfigurierbar; die Parameter wiederum sind in den HTML-Seiten direkt für die Anwender zugänglich. Böswillige Anwender könnten deshalb versuchen, die Lösungen entweder direkt aus den Webseiten abzulesen oder mit zusätzlichen Programmen aus den Applet-Parametern zu entschlüsseln.

Sicherheit vs. Hilfestellung...

Dies ist aber nur für die „einfachen" Applets ein Problem, bei denen die Musterlösung direkt in einem Parameter kodiert ist. Statt die Musterlösung zu dekodieren und dann mit der Eingabe zu vergleichen, ist es alternativ auch möglich, die Eingabe zu kodieren und mit der kodierten Musterlösung zu vergleichen.

In diesem Fall ist kein Dekoder notwendig und es könnten daher auch kryptographisch sichere Verfahren eingesetzt werden, für die eine Dekodierung als unmöglich gilt, beispielsweise die bekannten Signaturalgorithmen SHA und MD5. Allerdings würde die zusätzliche Sicherheit gegen Schummeln mit dem völligen Verlust der Hilfestellungen erkauft, da auch

eine fast korrekte Lösung nach Kodierung mit SHA oder MD5 eine vollständig andere Signatur liefert als die Musterlösung selbst. Als Kompromiss setzt die aktuelle Version von *CheckStringApplet* auf eine Variante der Base64 Kodierung, die von gängigen Dekodern nicht unterstützt wird. Die Studierenden müssten daher ihren eigenen Dekoder schreiben, um die Musterlösung zu entschlüsseln. Dies ist in jedem Fall wesentlich aufwendiger als das Lösen der zugrundeliegenden Übungsaufgaben.

Nebenbedingungen

Auf der anderen Seite ist die Übertragung der Applet-Parameter im Klartext für viele der „komplexeren“ Aufgaben überhaupt kein Problem, weil die Lösung sich erst durch Umrechnung aus den Parametern ergibt. Ein typisches Beispiel sind Aufgaben, die auf der Manipulation von Boole'schen Ausdrücken basieren. Wenn etwa mit dem *form* Parameter die minimierte disjunktive Form als Lösung gefordert wird, kann der originale nicht minimierte Ausdruck aus der Aufgabenstellung problemlos im Klartext als Musterlösung für *CheckBooleanExpressionApplet* verwendet werden.

Decompiler

Das Hineincompilieren einer Lösung in die Applets bietet übrigens auch keinen besseren Schutz als zum Beispiel das Base64-Verfahren. Zwar stellen wir den Quellcode der Applets nicht pauschal allen Anwendern zur Verfügung, aber dieser lässt sich mit gängigen Decompilern weitgehend rekonstruieren.

Ein Einsatz der Applets für Prüfungssituationen ist deshalb nur unter Aufsicht sinnvoll (von uns aber auch nicht vorgesehen). Natürlich ist es möglich, den direkten Zugriff auf die Applets beispielsweise über ein Webinterface abzuschirmen, wobei allerdings der Vorteil der direkten und schnellen Interaktion mit den Applets verloren geht.

2.5 Erweiterungen

Die bisher implementierten Applets decken bereits ein weites Spektrum von Aufgabentypen ab und erlauben die Überprüfung fast aller Aufgaben unserer Vorlesung Technische Informatik I [4]. Aber bereits das Beispiel dieser Vorlesung zeigt, dass für bestimmte Übungsaufgaben durchaus spezialisierte Applets notwendig sein können, mitunter für genau eine einzelne Aufgabe, vgl. die Beschreibung von *Check_T1_2_8_Applet* auf Seite 14.

Hier bietet Java ideale Voraussetzungen für Erweiterungen. Durch das objektorientierte Prinzip der Vererbung kann ein großer Teil der erforderlichen Funktionalität von geeigneten Basisklassen direkt übernommen werden. Für weitere Check*Applets bedeutet dies, dass lediglich die `check()` und die zugehörigen `help()` und `encode()` Methoden neu implementiert werden müssen. Da Softwarentwicklung in Java auf dem Class-Dateiformat beruht, können derartige anwendungsspezifische neue Klassen auch erstellt werden, ohne dass der Quelltext für die Basisklassen zur Verfügung steht. Es ist deshalb durchaus möglich (und von uns ausdrücklich erwünscht), dass Anwender aus anderen Fachgebieten die Check*Applets um entsprechende neue Unterklassen erweitern.

2.6 Internationalisierung

Natürlich ist ein möglichst weiter Einsatz der Check*Applets wünschenswert. Während die zur Überprüfung eingesetzen Algorithmen universell einsetzbar sind, enthalten die Applets derzeit nur Code zum Erzeugen von deutschen Meldungen und Hilfestellungen. Der Grund dafür ist einfach die im Rahmen des Projekts begrenzte Personalkapazität bei der Software- und Content-Entwicklung. Eine Anpassung an weitere Sprachvarianten, insbesondere natürlich Englisch, wäre jederzeit möglich. Dazu müssten die bisher fest kodierten Meldungen allerdings auf den Java *ResourceBundle*-Mechanismus umgestellt werden.

2.7 Deployment

Voraussetzung für den Einsatz der Check*Applets ist ein Webbrowser mit installierter Java Virtual Machine, möglichst in Version 1.4.2 oder neuer.

Die Applets werden in einem JAR-Archiv `checkapplets.jar` ausgeliefert, das alle notwendigen Java-Klassen enthält. Es genügt also, diese Datei zusammen mit den HTML-Seiten für die Applets auf einen Webserver zu installieren oder zum Download freizugeben.

Die Dateigröße des `checkapplets.jar` Archivs beträgt lediglich 180 KByte, so dass in einem lokalen Intranet oder über eine DSL-Verbindung Downloadzeiten von 1-2 Sekunden zu erwarten sind. Auch beim Download der Applets über eine langsame Modemverbindung (bei ca. 4 KByte/sec) ergeben sich noch akzeptable Downloadzeiten in der Größenordnung von maximal einer Minute.

Für die Option zum direkten Versenden einer erarbeiteten Lösung aus dem Applet an den zuständigen Übungsgruppenleiter setzen die Check*Applets auf die JavaMail-Klassen, die leider nicht im Standardumfang des JDK enthalten sind. Für das Nutzen dieser Funktion sind deshalb zwei zusätzliche JAR-Archive notwendig, und zwar das sogenannte Bean-Activiation Framework (activation.jar, ca. 55 KByte) mit einigen Hilfsklassen und die JavaMail-Klassen (mail.jar, ca. 330 KByte).

2.8 Dokumentation der einzelnen Check*Applets

Die Beschreibungen auf den folgenden Seiten dienen als Programm-Dokumentation und Spezifikation für die einzelnen bisher implementierten Applets.

Die Klassen sind dabei grob anhand ihrer Anwendungsgebiete sortiert; auf die trivial überprüfbaren Auswahlaufgaben folgen die Zahlenwertaufgaben und zunehmend komplexere Applets. Die Liste der Applets steht im Inhaltsverzeichnis und wird daher an dieser Stelle nicht wiederholt. Jede Beschreibung besteht aus den folgenden Teilen:

- Konzept und Einsatzgebiet des Applets, außerdem Hinweise auf besondere Eigenschaften oder Einschränkungen.
- Beschreibung der Applet-Parameter und eventueller Abhängigkeiten.
- Beschreibung der Optionen zur Content-Erstellung und zum Erstellen der kodierten Parameter.
- Abbildung mit einem Screenshot des Applets, soweit möglich mit einer typischen Hilfestellung oder Meldung des Applets.
- HTML-Codebeispiel mit den Parametern für das jeweils in der vorhergehenden Abbildung dargestellte Applet.

2.9 CheckStringApplet

Konzept

Die Klasse *CheckStringApplet* dient zunächst als Basisklasse für die Hierarchie der Check*Applets und stellt die gemeinsamen Grundfunktionen für das User-Interface und die Konfiguration bereit.

Gleichzeitig ermöglicht CheckStringApplet die Überprüfung des vom Anwender eingegeben Texts gegen eine Musterlösung. Durch einige Varianten, darunter Optionen zum Entfernen von Leerzeichen oder das Ignorieren von Groß- und Kleinschreibung, kann CheckStringApplet auch für Anwendungsfälle wie die Überprüfung von Zahlenwerten flexibel eingesetzt werden.

Anders als die auf den folgenden Seiten besprochenen differenzierten Unterklassen setzt CheckStringApplet zur Überprüfung nur einen einfachen Stringvergleich ein. Ausgefeilte Hilfestellungen mit anwendungsspezifischen Hinweisen sind daher nicht möglich. Immerhin kann CheckStringApplet auch die Länge von Strings und die Überlappung der Eingabe mit der Musterlösung auswerten und daraus Hilfestellungen ableiten. Als Test für die Akzeptanz der Überprüfung verrät die aktuelle Version von CheckStringApplet nach mehrfachen fehlerhaften Eingabeversuchen stückweise die Lösung. Dieses Verhalten kann aber leicht wieder entfernt werden.

Allgemeine Parameter

Die folgenden Applet-Parameter dienen zur Konfiguration der Applets und des User-Interfaces und stehen deshalb auch für die meisten Unterklassen von CheckStringApplet zur Verfügung:

check	Beschriftung des Check-Buttons, z.B. `"Überprüfen"`.
help	Beschriftung des Help-Buttons, z.B. `"Hilfe"`.
setup	Beschriftung des Setup-Buttons, z.B. `"Einstellungen"`.
submit	Beschriftung des Submit-Buttons, z.B. `"Abschicken"`.
prompt	Beschriftung des Labels links neben dem Eingabe-Textfeld, z.B. `"Eingabe"`.
default	Anfänglicher Inhalt des Eingabe-Textfelds, z.B. `"xx.yyyy"`. Durch Auswahl eines geeigneten Texts kann die Art und das Format der vom Applet erwarteten Eingabe angedeutet werden, was eine große Hilfe für die Anwender darstellt.
textarea	Optionaler Parameter zur Umschaltung auf ein mehrzeiliges Textfeld der angegeben Größe anstelle des einzeiligen Textfelds, z.B. `"6x80"` für sechs Zeilen à 80 Zeichen.
scrollpane	Optionaler Parameter zur Einbettung des Eingabe-Textfelds in ein scrollbares Feld. Mögliche Werte sind `"true"` und `"false"`.

Parameter zur Überprüfung

Die zweite Gruppe der Parameter von CheckStringApplet dient zur Einstellung verschiedener Optionen zur String-Überprüfung. Diese Parameter werden von den meisten übrigen Check*Applet einfach ignoriert:

value	Musterlösung im Klartext (für Testzwecke).
scrambled	Mit der `-encode` Option kodierte Musterlösung.
oneway	Mit der `-oneway` Option kodierte Musterlösung. Es muss genau einer der drei Parameter `value`, `scrambled`, oder `oneway` angegeben werden.
nchars	Optionale Vorgabe der gewünschten Länge der Eingabe; oder `"-1"` für beliebige Länge der Eingabe.

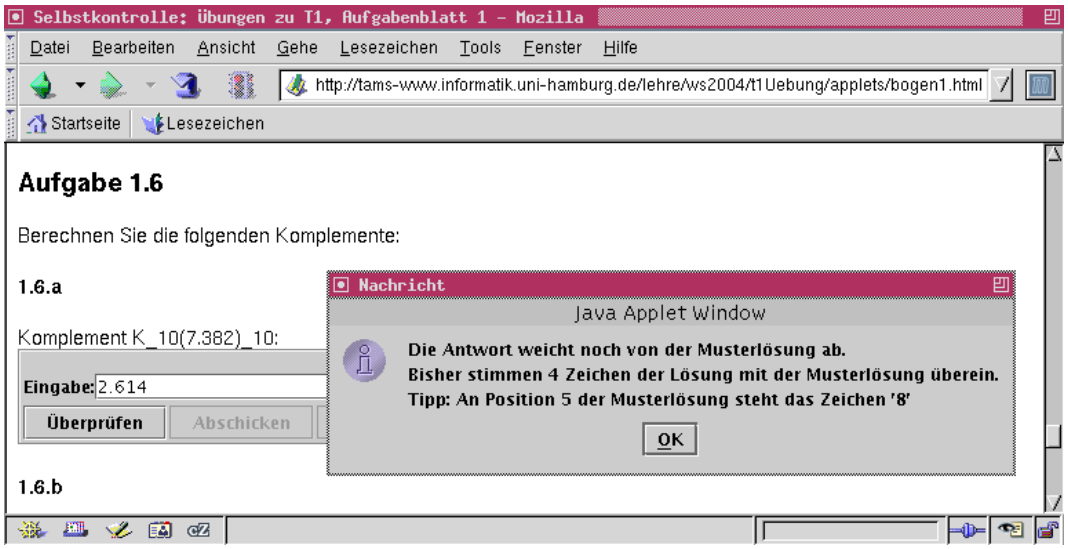

Abbildung 5: *CheckStringApplet dient als Basisklasse und ermöglicht den Vergleich des eingegebenen Textes mit einer vorgegebenen Musterlösung. Ausgehend von der Übereinstimmung zwischen Eingabe und Musterlösung werden unterschiedliche Meldungen erzeugt; nach einigen Fehlversuchen verrät das Applet sukzessive Teile der Musterlösung.*

```
<applet code=de.mmkh.tams.CheckStringApplet.class codebase="."
 archive="checkapplets.jar" width="600" height="70"
>
<param name="default"   value="0.xxxx" />
<param name="scrambled" value="MC42NzMy" />
</applet>
```

stripspaces Optional, wenn `"true"` werden vor der Überprüfung alle Leerzeichen aus der Eingabe entfernt.

purgespaces Optional, wenn `"true"` werden vor der Überprüfung aufeinanderfolgende Leerzeichen durch einzelne Leerzeichen ersetzt.

case-sensitive Optional, wenn `"false"` wird die Eingabe vor der Überprüfung in Kleinbuchstaben umgewandelt.

unicode Optional, wenn `"true"` werden vor der Überprüfung alle Zeichenfolgen `"\uxxxx"` im eingegebenen Text durch das zugehörige Unicode-Zeichen ersetzt.

Encoding

Die (Base64) verschlüsselten Strings für den `scrambled` Parameter lassen sich direkt mit der `-encode` Option von CheckStringApplet erzeugen. Bei Einsatz des `-oneway` Parameters wird eine andere Kodierung verwendet, die sich mit gängiger Software nicht wieder entschlüsseln lässt. In beiden Fällen können außer den üblichen ASCII-Zeichen auch beliebige Unicode-Zeichen in der `\uxxxx` Schreibweise übergeben werden, mit `xxxx` als hexadezimaler Kodierung des gewünschten Zeichens, zum Beispiel `\u0020` für ein Leerzeichen und `\u000d` für einen Linefeed:

```
java de.mmkh.tams.CheckStringApplet -encode 'Musterlösung'
TXVzdGVybPZzdW5n
java de.mmkh.tams.CheckStringApplet -oneway 'Musterlösung'
imkCsVkBqeoCslI2
```

2.10 Check_T1_2_8_Applet

Konzept Aufgrund der objektorientierten Architektur kann die Funktionalität von CheckStringApplet leicht durch Bilden neuer Unterklassen erweitert und an neue Anwendungsfälle angepasst werden. Im einfachsten Fall muss nur die `check()`-Methode entsprechend neu geschrieben werden, obwohl dann natürlich auch `help()` modifiziert werden sollte, um die zugehörige Hilfe anzuzeigen.

Ein Beispiel für dieses Vorgehen liefert die Klasse *Check_T1_2_8_Applet*, die speziell für die Übungsaufgabe T1.2.8 [4] erstellt wurde. Anders als die ursprüngliche Version von CheckStringApplet, die nur ein einzeiliges Textfeld zur Eingabe bereitstellte, verwendet das User-Interface von *Check_T1_2_8_Applet* ein Textfeld mit 8 Zeilen und erlaubt die Eingabe eines mehrzeiligen Textes. Die Musterlösung für dieses Applet ist fest hineincompiliert, da sie Sonderzeichen enthält, die sich nicht direkt in HTML und damit einen Applet-Parameter einbetten lassen.

Zwar verfügt auch CheckStringApplet durch nachträgliche Erweiterungen mittlerweile über die Möglichkeit, direkt ein mehrzeiliges Textfeld anzuzeigen und Sonderzeichen in Applet-Parameter einzubetten, siehe die Beschreibung der Parameter auf Seite 12. Angesichts der Codegröße von weniger als 2 KByte ist es trotzdem unkritisch, *Check_T1_2_8_Applet* als eigenständige Klasse beizubehalten.

Encoding Die Musterlösung für Aufgabe T1.2.8 ist fest in das Applet hineincompiliert. Weitere Funktionen zum Kodieren der Musterlösung sind daher nicht erforderlich.

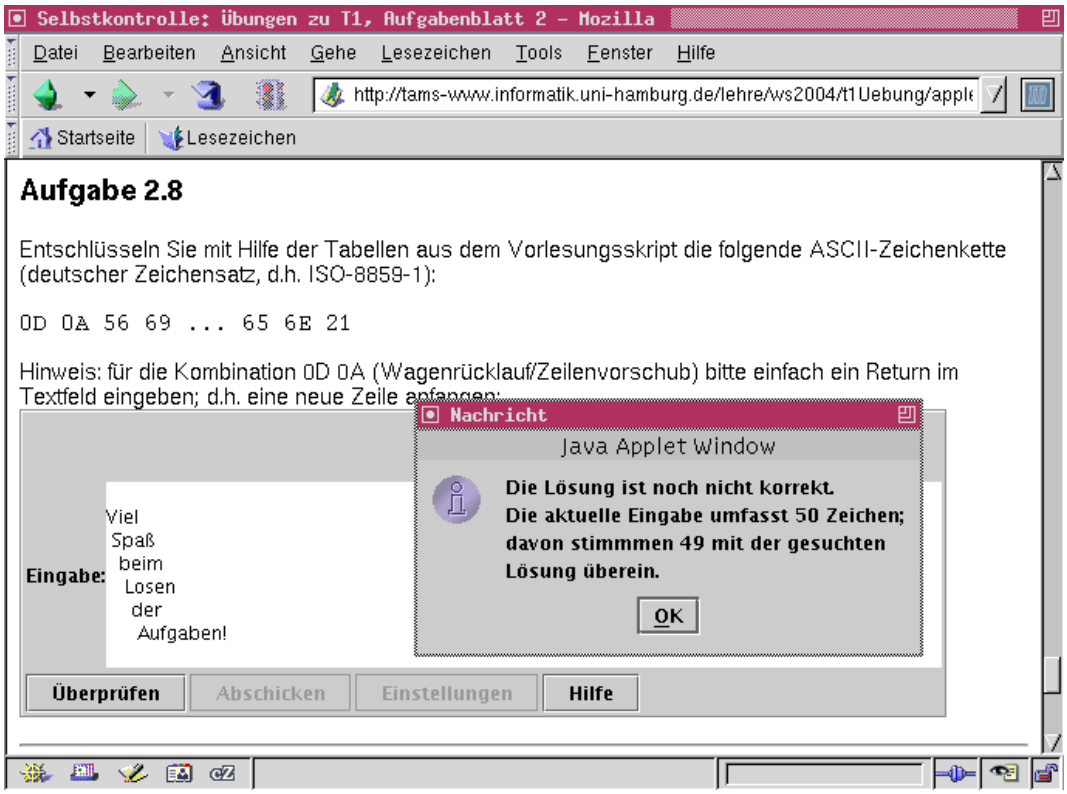

Abbildung 6: *Check_T1_2_8_Applet dient als Beispiel für ein Applet, das speziell für eine einzige Aufgabe (hier T1.2.8) zum Einsatz kommt. Als Unterklasse von CheckStringApplet erlaubt es die Eingabe eines mehrzeiligen Texts und vergleicht diesen mit der fest kodierten Musterlösung.*

```
<applet code=de.mmkh.tams.Check_T1_2_8_Applet.class codebase="."
 archive="checkapplets.jar" width="600" height="200"
>
<param name="debug"    value="true" />
<param name="textarea" value="8x40" />
<param name="default"  value="Bitte den Text eingeben" />
</applet>
```

2.11 CheckMultipleChoiceApplet

Konzept Wegen der einfachen Überprüfbarkeit durch direkten Vergleich mit einer Musterlösung ist die Kategorie der *Auswahlaufgaben* in gängigen E-Learning Umgebungen sehr beliebt. Neben den bekannten Multiple-Choice Fragen gehören zum Beispiel auch Zuordnungsaufgaben in diese Kategorie. Solche Aufgaben lassen sich mit einfachen JavaScript-Funktionen überprüfen und direkt in Webseiten einbetten.

Auf der anderen Seite spielen Auswahlaufgaben für das dem Projekt zugrundeliegende Themengebiet der technischen Informatik aber kaum eine Rolle [11], wie bereits im ersten Projektbericht ausführlich erläutert wurde. So enthalten die Übungen zur Vorlesung Technische Informatik I [4] nur genau eine einzige Multiple-Choice-Aufgabe.

In dieser Situation dient die Klasse *CheckMultipleChoiceApplet* nur zur Demonstration und als Prototyp für die Integration in die Klassenhierarchie der Check*Applets. Die einzelnen Einträge für die Auswahl werden dabei über Parameter an das Applet übergeben, das diese zusammen mit jeweils einer einfachen Checkbox untereinander darstellt. Die Überprüfung berechnet dann die Summe der korrekt ausgewählten Einträge und zeigt diese an. Weitere Optionen, etwa zum Lösen der Aufgabe, sind bisher nicht implementiert.

Parameter CheckMultipleChoiceApplet wird über die folgenden Parameter gesteuert:

choices	Anzahl n der darzustellenden Auswahlmöglichkeiten, es wird Platz für je einen Label reserviert.
multiple	Anzahl der gleichzeitig auswählbaren Einträge.
label(i)	Text für den Eintrag i ($0 \leq i < n$).
scrambled	Kodierte Musterlösung; der Wert kann mit der `-encode` Option berechnet werden.

Encoding Da die Musterlösung für eine Auswahlaufgabe nur aus wenigen Bits besteht, ist die einfache Base64-Kodierung kaum zur Darstellung der Musterlösung geeignet. Die erzeugten Bitmuster wären bereits auf den ersten Blick zu erkennen. Deshalb verfügt CheckMultipleChoiceApplet über eine `-encode` Methode, die aus einem angegeben String mit der Musterlösung (je ein Zeichen 0 oder 1 für jeden Eintrag der Aufgabe) den zugehörigen `scrambled` Parameter berechnet:

```
java de.mmkh.tams.CheckMultipleChoiceApplet -encode '100010'
gFblqt
```

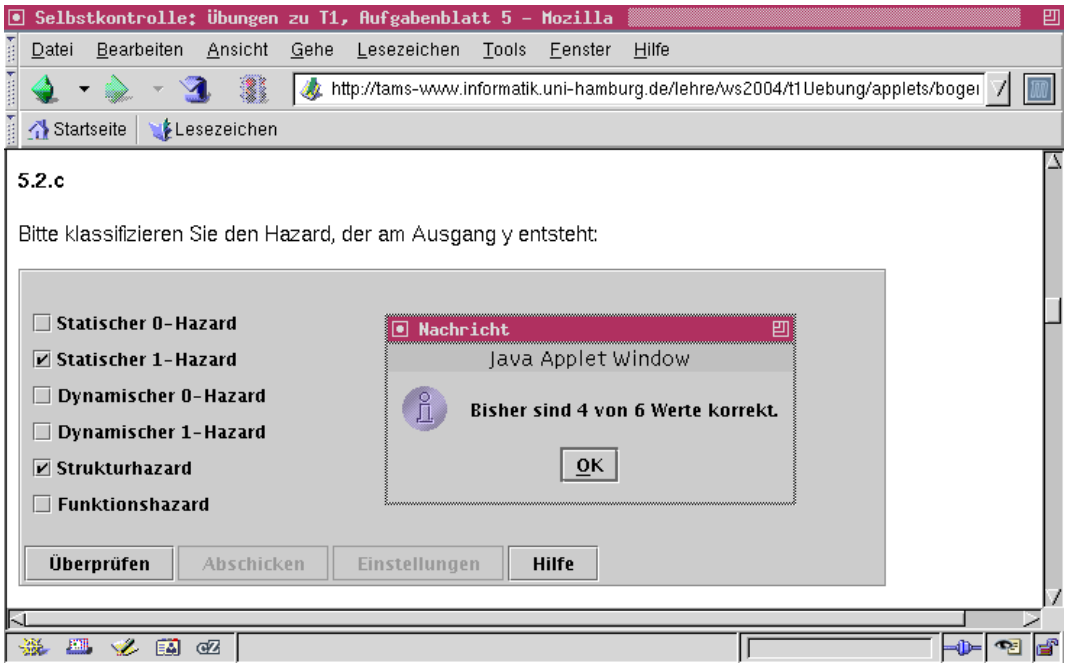

Abbildung 7: *CheckMultipleChoiceApplet erlaubt die Einbindung der üblichen Multiple-Choice Fragen.*

```
<applet
 code=de.mmkh.tams.CheckMultipleChoiceApplet.class  codebase="."
 archive="checkapplets.jar" width="600" height="220"
>
<param name="choices"    value="6" />
<param name="multiple"   value="2" />
<param name="label0"     value="Statischer 0-Hazard" />
<param name="label1"     value="Statischer 1-Hazard" />
<param name="label2"     value="Dynamischer 0-Hazard" />
<param name="label3"     value="Dynamischer 1-Hazard" />
<param name="label4"     value="Strukturhazard" />
<param name="label5"     value="Funktionshazard" />
<param name="scrambled"  value="kZh4qh" />
</applet>
```

2.12 CheckNumberApplet

Konzept Die Klassifikation der Zahlenwertaufgaben im ersten Projektbericht [11] unterscheidet mehrere Aufgabentypen nach der geforderten Zahldarstellung. Allen Varianten ist gemeinsam, dass sich die Antwort problemlos durch Vergleich mit der bekannten Musterlösung überprüfen lässt. Die Klasse *CheckNumberApplet* unterstützt sowohl Gleitkommazahlen nach dem IEEE-Standard als auch Integerzahlen, während für Aufgaben mit Festkommazahlen die verwandte Klasse *CheckFractionApplet* zum Einsatz kommt (siehe Seite 20).

Für die Überprüfung von Integerzahlen erlaubt CheckNumberApplet die Auswahl einer Zahlenbasis (im Bereich 2..36) und die Vorgabe einer festen Anzahl von Ziffern. Beide Varianten kommen im Kontext der technischen Informatik häufig vor. Der Parser erkennt die gängige Schreibweise von Binärzahlen mit führendem 0b und Hexadezimalzahlen mit führendem 0x. Für Integerzahlen mit fester Basis wird das Format `[-]ddd...d_bb` erwartet, wobei `bb` die dezimale Schreibweise der Basis ist und `d` eine Ziffer der gewählten Basis repräsentiert, zum Beispiel `cafe_16` oder `10110001_2`.

Für Gleitkommazahlen ist optional die Vorgabe eines Toleranzintervalls möglich, innerhalb dessen die Eingabe noch als korrekt interpretiert wird. Dies erlaubt einerseits das Ignorieren von Rundungsfehlern, vor allem aber eine der Aufgabenstellung angemessene Behandlung gegenüber Näherungen und Schätzungen im gewählten Lösungsweg.

Abhängig von den Voreinstellungen sowie dem eingegeben Zahlenwert und dem Zahlenwert der Musterlösung erzeugt CheckNumberApplet eine Vielzahl von Meldungen als Hilfestellungen, die dem Anwender das Herantasten an die korrekte Lösung erleichtern.

Parameter Zusätzlich zu den Standardparametern von CheckStringApplet unterstützt CheckNumberApplet die folgenden Parameter:

format	Auswahl der Betriebsart, mögliche Werte sind `"integer"` und `"double"`.
base	Geforderte Zahlenbasis für Integerzahlen im Bereich 2..36, oder `"-1"` für beliebige Basis.
ndigits	Geforderte Anzahl der Ziffern der Lösung oder `"-1"` für beliebige Anzahl. Bei ndigits>1 müssen vom Anwender eventuell führende Nullen eingegeben werden.
tolerance	Maximal zulässige Abweichung der Eingabe von der Musterlösung als Gleitkommazahl.
value	Angabe der Musterlösung im Klartext (für Testzwecke).
scrambled	Kodierte Musterlösung.

Encoding Die für den `scrambled` Parameter benötigten Daten lassen sich mit der `-encode` Option erzeugen; die übrigen Werte können problemlos im Klartext in die Applet-Parameter eingetragen werden:

```
java de.mmkh.tams.CheckNumberApplet -encode '3.14159265'
My4xNDE1OTI2NQ==
```

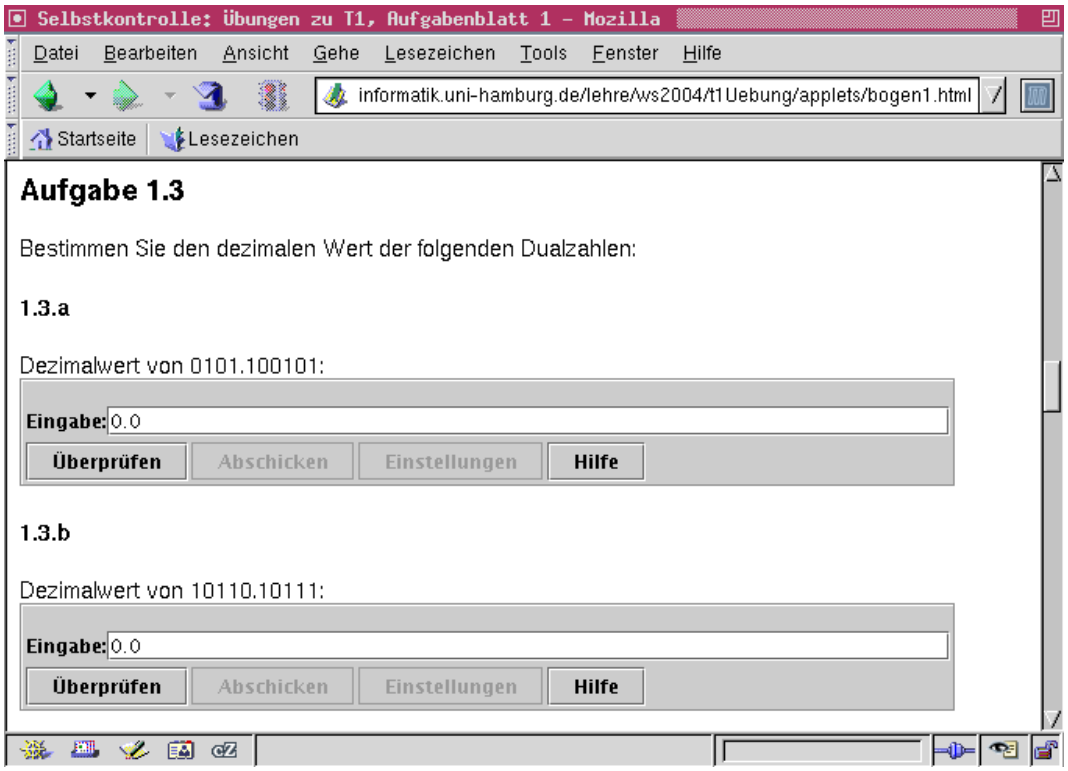

Abbildung 8: *CheckNumberApplet ist weitgehend konfigurierbar und eignet sich für viele Zahlenwertaufgaben mit Integerzahlen zur Basis 2..36 und Gleitkommazahlen.*

```
<applet code=de.mmkh.tams.CheckNumberApplet.class codebase="."
 archive="checkapplets.jar" width="600" height="70"
>
<param name="default"   value="x.yy" />
<param name="format"    value="double" />
<param name="tolerance" value="0.01" />
<param name="value" value="4" />
<param name="scrambled" value="NC4wMCA="/ >
</applet>
```

2.13 CheckFractionApplet

Konzept Die Klasse *CheckFractionApplet* dient zur Überprüfung von Zahlenwertaufgaben, deren Lösung Festkommazahlen erfordert. Ähnlich wie bei CheckNumberApplet kann eine Zahlenbasis im Bereich 2..36 gefordert sowie die Anzahl der Vor- und Nachkommastellen festgelegt werden. Da die Auswertung numerisch erfolgt, ist die Behandlung periodischer Brüche nicht möglich. In solchen Fällen müssen die Aufgabenstellung und Musterlösung auf entsprechend gerundete Werte umgestellt werden.

Abhängig von den Voreinstellungen sowie dem eingegeben Zahlenwert und dem Zahlenwert der Musterlösung erzeugt CheckFractionApplet eine Vielzahl von Meldungen als Hilfestellungen, die dem Anwender das Herantasten an die korrekte Lösung erleichtern.

Parameter Zusätzlich zu den Standardparametern von CheckStringApplet unterstützt CheckFractionApplet die folgenden Parameter:

base	Geforderte Zahlenbasis für Integerzahlen im Bereich 2..36.
ndigits	Geforderte Anzahl der Vorkommastellen der Lösung oder `"-1"` für beliebige Anzahl. Bei ndigits$>$ 1 müssen vom Anwender eventuell führende Nullen eingegeben werden.
fdigits	Geforderte Anzahl der Nachkommastellen der Lösung oder `"-1"` für beliebige Anzahl.
tolerance	Maximal zulässige Abweichung der Eingabe von der Musterlösung als Gleitkommazahl.
scrambled	Kodierte Musterlösung.

Encoding Die für den `scrambled` Parameter benötigten Daten lassen sich mit der `-encode` Option erzeugen; die übrigen Werte können problemlos im Klartext in die Applet-Parameter eingetragen werden:

```
java de.mmkh.tams.CheckFractionApplet -encode 'cafe.babe'
Y2FmZS5iYWJl
```

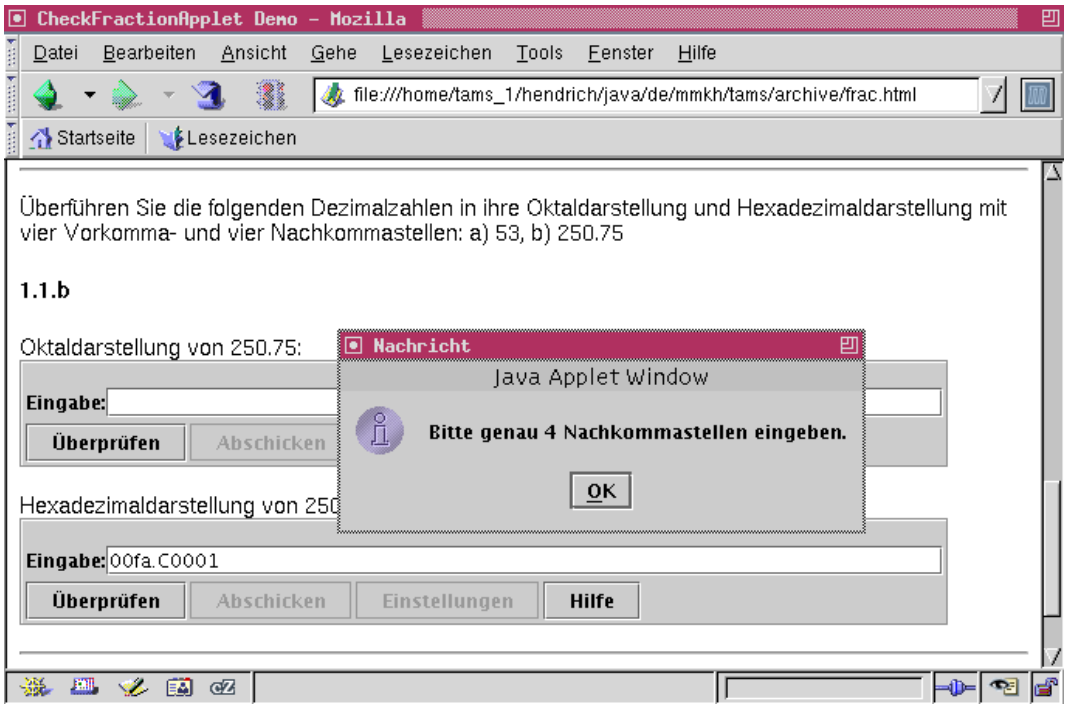

Abbildung 9: *CheckFractionApplet kommt bei Fragen nach Festkommazahlen zum Einsatz; die Zahlenbasis kann dabei im Bereich 2..36 gewählt werden.*

```
<applet code=de.mmkh.tams.CheckFractionApplet.class codebase="."
 archive="checkapplets.jar" width="600" height="70"
>
<param name="default"          value="0000.0000" />
<param name="scrambled"        value="MDBGQS5DMDAw" />
<param name="base"             value="16" />
<param name="ndigits"          value="4" />
<param name="fdigits"          value="4" />
<param name="tolerance"        value="0" />
</applet>
```

2.14 CheckWaveformApplet

Konzept Ein typisches Hilfsmittel bei der Analyse und Diskussion digitaler Schaltungen ist die Darstellung der Signalverläufe als Funktion der Zeit, die sogenannten Impulsdiagramme oder Waveforms. Die Klasse *CheckWaveformApplet* ermöglicht die graphische Darstellung derartiger Impulsdiagramme in Webseiten, sowohl zur reinen Illustration als auch für Übungsaufgaben mit Überprüfung.

Um eine aufwendige Einarbeitung in die Bedienung zu vermeiden, stellt CheckWaveformApplet nur die notwendigen Grundfunktionen bereit, ist jedoch keinesfalls als Ersatz für einen professionellen WaveformViewer gedacht. Durch die Beschränkung auf ein festes Zeitraster und das einfache 0/1/X/Z-Logikmodell ist eine direkte Benutzerinteraktion durch mehrfaches Anklicken eines Feldes möglich, wobei nacheinander die Werte 0, 1, X (undefiniert) und Z (tristate) durchlaufen werden. Die Überprüfung erfolgt durch direkten Vergleich mit entsprechend vorgegebenen Musterlösung, die zusätzlich auch den Wert '*' für einen don't care enthalten können.

Leider liefert das Base64-Verfahren bei der Kodierung vieler typischer Impulsdiagramme auffällige Wiederholungen bestimmter Zeichenfolgen, die im Quellcode auf den ersten Blick zu erkennen sind. Deshalb setzt CheckWaveformApplet auf ein leicht modifiziertes Verfahren, das derartige Wiederholungen vermeidet. Die Applet-Parameter lassen sich daher nur mit der `encode`-Methode von CheckWaveformApplet selbst erzeugen.

Parameter Zusätzlich zu den Standard-Parametern aus CheckStringApplet unterstützt CheckWaveformApplet die folgenden Parameter:

timesteps	Anzahl der Zeitschritte in den einzelnen Waveforms.
nwaves	Anzahl der in der Aufgabenstellung fest vorgegebenen Eingangs-Signale n_i.
wavenames	Liste der Signalnamen mit Komma als Trennzeichen.
wave(j)	String mit den Logikwerten für das Eingangssignal j mit je einem Zeichen (01XZ*) pro Zeitschritt. Für jedes Eingangssignal $j = 0, 1, \ldots (n_i - 1)$ wird ein einzelner Parameter (`"wave0"`, `"wave1"`, ...) benötigt.
nxwaves	Anzahl der von den Studierenden auszufüllenden Ausgangs-Signale n_y.
xwave(k)	String mit den Default-Logikwerten für das jeweilige Ausgangssignal $k = 0, 1, \ldots (n_y - 1)$.
xencoded(k)	Kodierte Musterlösung für das Ausgangssignal k. Die Musterlösung kann mittels `-encode` aus einem 01XZ*-String berechnet werden.

Encoding Durch Aufruf mit dem `-encode` Parameter kann der String mit dem Werteverlauf (01XZ*) der Musterlösung für ein Ausgangssignal in das für die `xencoded`-Parameter benötigte Format umgerechnet werden. Im Resultat steht zunächst eine Zufallszahl und anschließend der mit dieser Zufallszahl verrechnete String der Musterlösung. Bei mehreren Ausgangssignalen sind entsprechend mehrere Aufrufe notwendig:

```
java de.mmkh.tams.CheckWaveformApplet -encode "0010x***00011z00"
1903800871,JK0U0InfJUqnJwte
```

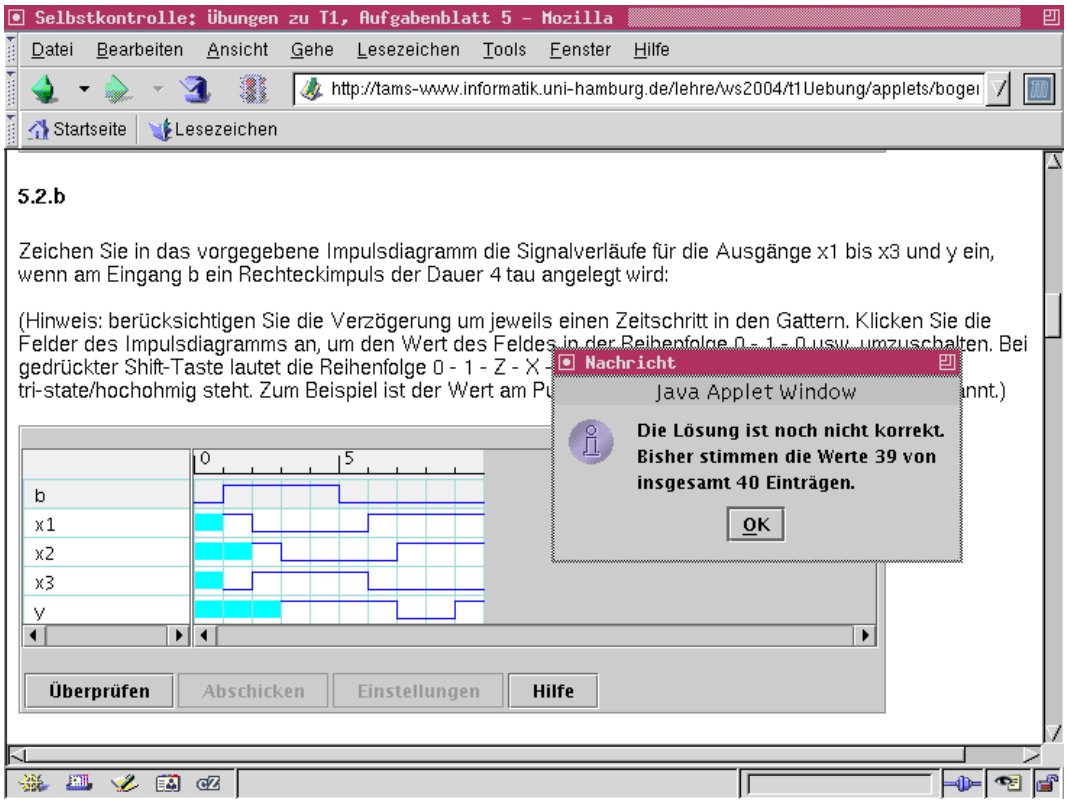

Abbildung 10: *CheckWaveformApplet dient zur interaktiven Eingabe und Überprüfung von Impulsdiagrammen (waveforms). Um eine möglichst einfache Bedienung zu erreichen, wird das 0/1/X/Z-Logikmodell auf einem festen Zeitraster verwendet. Durch mehrfaches Anklicken eines Feldes kann der gewünschte Logikwert eingestellt werden.*

```
<applet
 code=de.mmkh.tams.CheckWaveformApplet.class  codebase="."
 archive="checkapplets.jar" width="600" height="200"
>
 <param name="nwaves"     value="1"  />
 <param name="nxwaves"    value="4"  />
 <param name="timesteps"  value="10" />
 <param name="wavenames"  value="b"  />
 <param name="xwavenames" value="x1,x2,x3,y" />
 <param name="wave0"      value="0111100000" />
 <param name="xwave0"     value="U000000000" />
 <param name="xwave1"     value="UU00000000" />
 <param name="xwave2"     value="U000000000" />
 <param name="xwave3"     value="UUU0000000" />
 <param name="xencoded0"  value="1617717555,zBfY/7+1lh"  />
 <param name="xencoded1"  value="1685448685,C+hGANpBbY"  />
 <param name="xencoded2"  value="-1241320874,RAwJNC6fXF" />
 <param name="xencoded3"  value="1311534051,RvhAs8zgfO"  />
</applet>
```

2.15 CheckFanoCodeApplet

Konzept Dieses Applet interpretiert die Texteingabe als Binärcode und überprüft, ob die Codewörter einen Fano-Code mit den von der Aufgabenstellung geforderten Eigenschaften bilden. Ein trivialer Vergleich mit einer vorgegebenen Musterlösung ist dabei nicht möglich, da fast immer viele gleichwertige Codes existieren. Zum Beispiel entsteht durch Invertieren aller Bits eines Codes wieder ein gültiger Code, aber es sind auch viele komplexere Operationen und Permutationen möglich.

Zur Überprüfung ist daher ein komplexeres Vorgehen notwendig. Im ersten Schritt wird die Eingabe in die einzelnen Codewörter zerlegt und deren Anzahl und Aufbau als gültige Binärzahlen überprüft. Der zweite Schritt erkennt doppelt oder mehrfach vorkommende Codewörter. Danach wird für alle Codewörter die Fano-Bedingung (kein Codewort ist Anfang eines anderen Codeworts) überprüft. Schließlich wird aus den Längen der einzelnen Codewörter und den vorgegebenen Symbolwahrscheinlichkeiten die Effizienz des Codes berechnet und mit dem vorgegebenen Wert verglichen. Für alle Schritte wird eine entsprechende Meldung als Hilfestellung ausgegeben, sobald die zugehörige Bedingung verletzt ist.

Auf diese Weise akzeptiert das Applet alle gültigen Codes für die jeweilige Aufgabenstellung. Für unsere Aufgabe T1.3.3 mit sieben Codewörten und zugehörigen Symbolwahrscheinlichkeiten (0.25, 0.25, 0.125, 0.125, 0.125, 0.0625, 0.0625) sind dies zum Beispiel die Codes `00 01 100 101 110 1110 1111` oder `00 10 010 011 110 1111 1110`.

Parameter Zusätzlich zu den Standard-Parametern aus CheckStringApplet unterstützt CheckFanoCodeApplet die folgenden Parameter:

costs — Vorgabe der geforderten Effizienz des Codes, berechnet aus der Anzahl der Bits n_i eines Codeworts und der zugehörigen Symbolwahrscheinlichkeit p_i als $\sum_i p_i \cdot n_i$.

probabilities — Liste mit den Symbolwahrscheinlichkeiten der einzelnen Codewörter als Gleitkommazahlen. Die notwendige Anzahl der Codewörter ergibt sich implizit aus der Anzahl der Werte. Wenn dieser Parameter fehlt, gelten die oben genannten Werte für unsere Aufgabe T1.3.3.

Encoding CheckFanoCodeApplet bietet keine zusätzlichen Funktionen an.

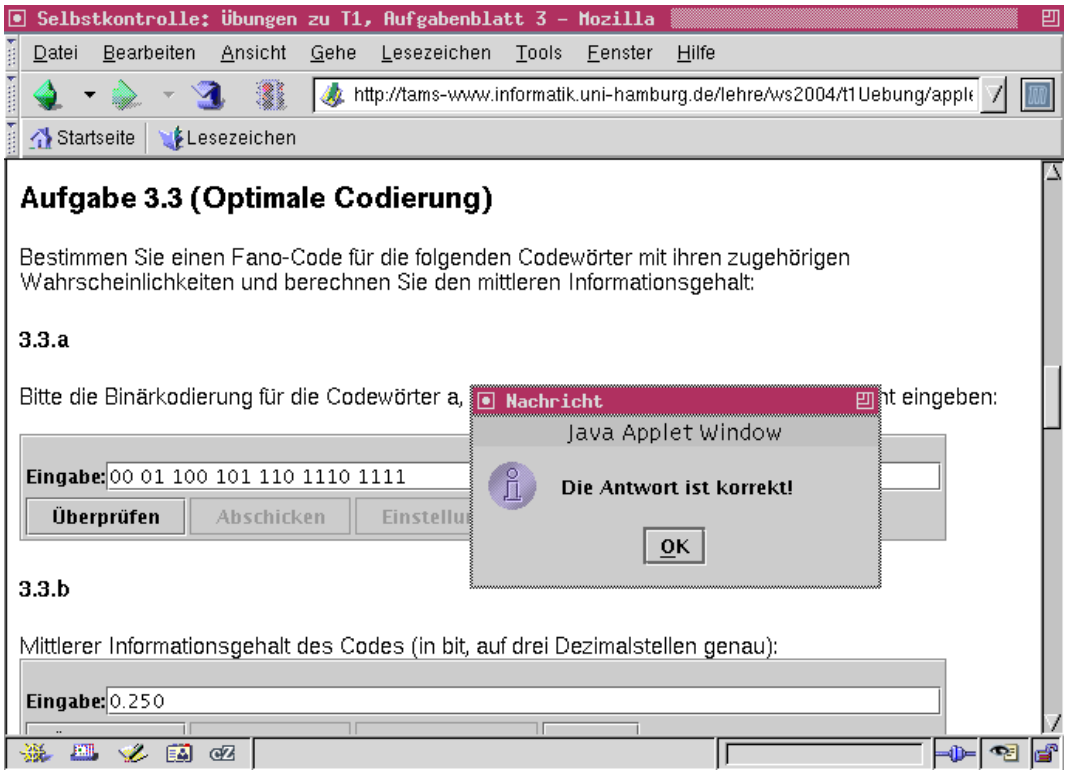

Abbildung 11: *CheckFanoCodeApplet erlaubt die Überprüfung der eingegebenen Codewörter auf die Fano-Bedingungen zum Entwurf optimaler Codes. Da fast immer viele gleichwertige Codes existieren, kommt ein einfacher Vergleich mit einer Musterlösung nicht in Frage. Vielmehr überprüft das Applet diese Bedingungen und die Güte des Codes anhand der vorgegebenen Symbolwahrscheinlichkeiten. Damit werden alle gültigen Lösungen akzeptiert.*

```
<applet
 code=de.mmkh.tams.CheckFanoCodeApplet.class  codebase="."
 archive="checkapplets.jar" width="600" height="70"
>
<param name="default"       value="000 001 ..." />
<param name="costs"         value="2.625" />
<param name="probabilities"
        value="0.25,0.25,0.125,0.125,0.125,0.0625,0.0625" />
</applet>
```

2.16 CheckGrayCodeApplet

Konzept Dieses Applet interpretiert die Texteingabe als Binärcode und überprüft, ob die Codewörter einen einschrittigen Code bilden. Dabei kann die Anzahl der Codewörter frei vorgegeben werden, unsere Aufgabe T1.2.10 fordert zum Beispiel $n = 12$. Auch bei dieser Aufgabe ist ein trivialer Vergleich mit einer vorgegebenen Musterlösung nicht möglich, da viele gleichwertige Codes existieren, die sich nur teilweise durch einfache Operationen wie das Invertieren von Bits oder Umsortieren der Codewörter ineinander überführen lassen.

Zur Überprüfung wird daher das folgende Prinzip angewendet. Im ersten Schritt wird die Eingabe in die einzelnen Codewörter zerlegt und deren Anzahl und Aufbau als gültige Binärzahlen überprüft. Der zweite Schritt erkennt doppelt oder mehrfach vorkommende Codewörter. Danach wird für alle Codewörter die Hamming-Distanz zum jeweils vorangehenden Codewort überprüft. Für alle Schritte wird eine entsprechende Meldung als Hilfestellung ausgegeben, sobald die zugehörige Bedingung verletzt ist.

Parameter Außer den Standard-Parametern aus CheckStringApplet unterstützt CheckGrayCodeApplet nur einen zusätzlichen Parameter:

nwords Anzahl der geforderten Codewörter des einschrittigen Codes.

Encoding CheckGrayCodeApplet benötigt keine Zusatzfunktionen zum Kodieren einer Musterlösung.

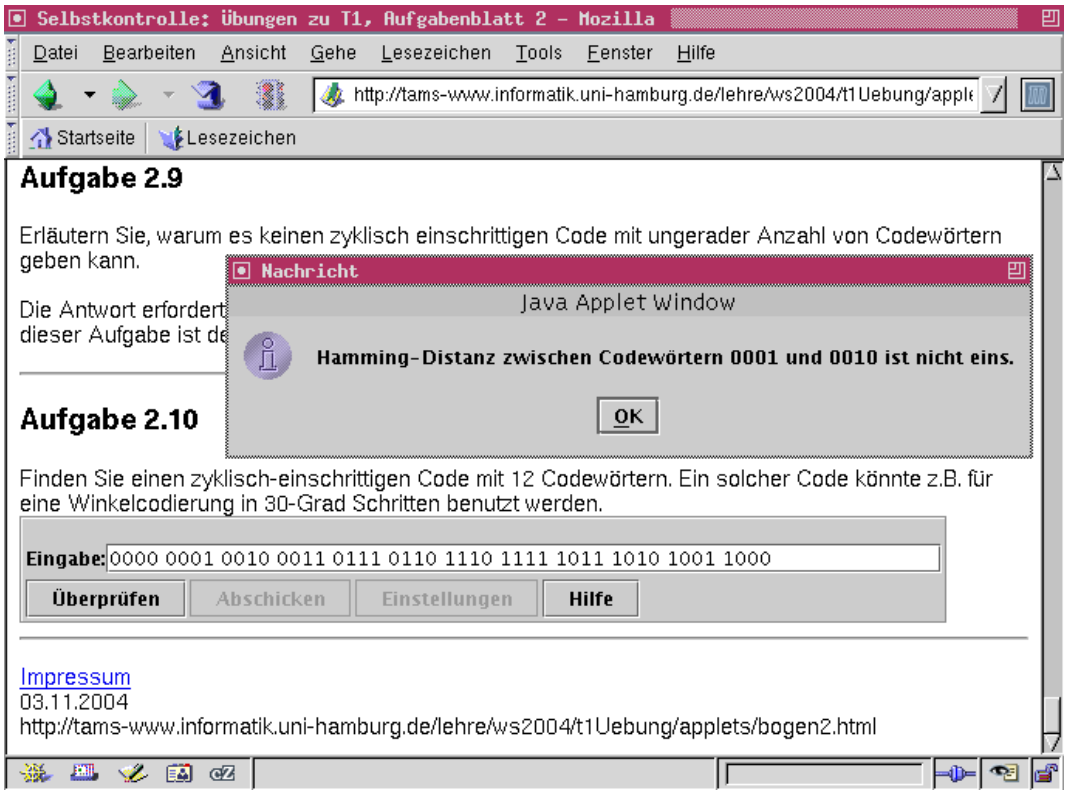

Abbildung 12: *CheckGrayCodeApplet überprüft, ob die eingegebenen Binärwörter einen einschrittigen Code bilden. Da eine Vielzahl von möglichen (und gleichwertigen) Lösungen existiert, ist ein einfacher Vergleich mit einer Musterlösung unmöglich.*

```
<applet
 code=de.mmkh.tams.CheckGracCodeApplet.class codebase="."
 archive="checkapplets.jar" width="600" height="90"
>
<param name="default"    value="0000 0001 0010 .... ...." />
<param name="nwords"     value="12" />
</applet>
```

2.17 CheckBooleanExpressionApplet

Konzept Wegen ihrer Bedeutung für das Verständnis und den Entwurf digitaler Schaltungen bildet die Boole'sche Algebra einen besonderen Schwerpunkt in der technischen Informatik. Dies gilt auch für die Übungsaufgaben — fast die Hälfte aller Aufgaben zur Vorlesung Technische Informatik 1 behandelt die Umrechnung und Minimierung von Boole'schen Ausdrücken bzw. Funktionen [4].

BooleanParser Wie bereits im letzten Projektbericht [13] erläutert wurde, setzen wir für die Überprüfung von Boole'schen Ausdrücken und den Vergleich von Schaltfunktionen auf einen selbstentwickelten Parser *BP2*. Dieser wurde mit dem JavaCC Parsergenerator [47] erzeugt und unterstützt sowohl die Infix-Schreibweise als auch eine Reihe von vordefinierten Funktionen. Ausdrücke dürfen die Literale `0` und `1`, benutzerdefinierte Variablennamen, Klammern, sowie die Operatoren `&`, `|`, `+`, `~` für UND, ODER, XOR und Negation enthalten, die übliche Operatorpriorität wird berücksichtigt. Die Liste der vordefinierten Funktionen umfasst unter anderem `NOT`, `AND`, `OR`, `XOR`, `XNOR`, sowie die negierten Formen wie `NOR`, `NAND` und `NAND3`.

Der Parser erlaubt die Auswertung eines Ausdrucks für vorgegebene Variablenbelegungen, kann aber auch automatisch die vollständige Funktionstabelle aufstellen. Zusätzliche Funktionen erlauben die Überprüfung von Nebenbedingungen wie der Art und Anzahl der vorkommenden Funktionen oder das Einhalten spezieller Formen wie der disjunktiven Normalform oder der Reed-Muller-Form.

Die Klasse *CheckBooleanExpressionApplet* verwendet diesen Parser zur Überprüfung von als Text eingegebenen Boole'schen Ausdrücken. Die zugehörige Musterlösung kann dabei entweder direkt als Funktionstabelle oder alternativ ebenfalls als (kodierter) Boole'scher Ausdruck vorgegeben werden. Ein direktes Ablesen der Lösung aus diesen Parametern ist kein Problem, da die meisten Aufgaben eine oder mehrere Zusatzbedingungen fordern, die von der Musterlösung selbst nicht erfüllt werden.

Parameter Zusätzlich zu den Standard-Parametern aus CheckStringApplet unterstützt CheckBooleanExpressionApplet die folgenden Parameter:

nvars	Vorgabe von n Variablen mit Defaultnamen 'a', 'b', ...
varnames	Liste der Variablennamen mit Komma als Trennzeichen, dies setzt auch die Anzahl der Variablen n.
table	Funktionstabelle der Musterlösung als String aus (01*) der Länge 2^n Zeichen.
scrambled	Kodierter Ausdruck der Musterlösung. Bitte nur genau einen der beiden Parameter table und scrambled verwenden.
form	Optionale Vorgabe der Form der Lösung. Mögliche Werte sind SOP, DNF, POS, KNF, RMF für disjunktive Form, disjunktive Normalform, konjunktive Form, konjunktive Normalform und Reed-Muller-Form (infix-Schreibweise). Die Werte NOR, NAND, NANDNOT stehen für die Realisierung mit den jeweiligen Gattern (funktionale Form).
costs	Optionale Vorgabe der Realisierungskosten, nacheinander die Gesamtanzahl der Gatter und dann die Anzahl der, UND, ODER, XOR, NOT-Gatter, also z.B. 5,2,3,0,-1. Ein Wert von -1 wird für die Überprüfung ignoriert.

Encoding Da der Aufwand zur Aufstellung der Funktionstabelle mit der Anzahl n der Variablen exponentiell anwächst, sollten überflüssige Variablen soweit möglich vermieden werden. Je nach Aufgabenstellung kann die Angabe der Musterlösung über `table` oder `scrambled` günstiger sein; unter anderem lassen sich don't care-Werte nur über die Funktionstabelle vorgeben.

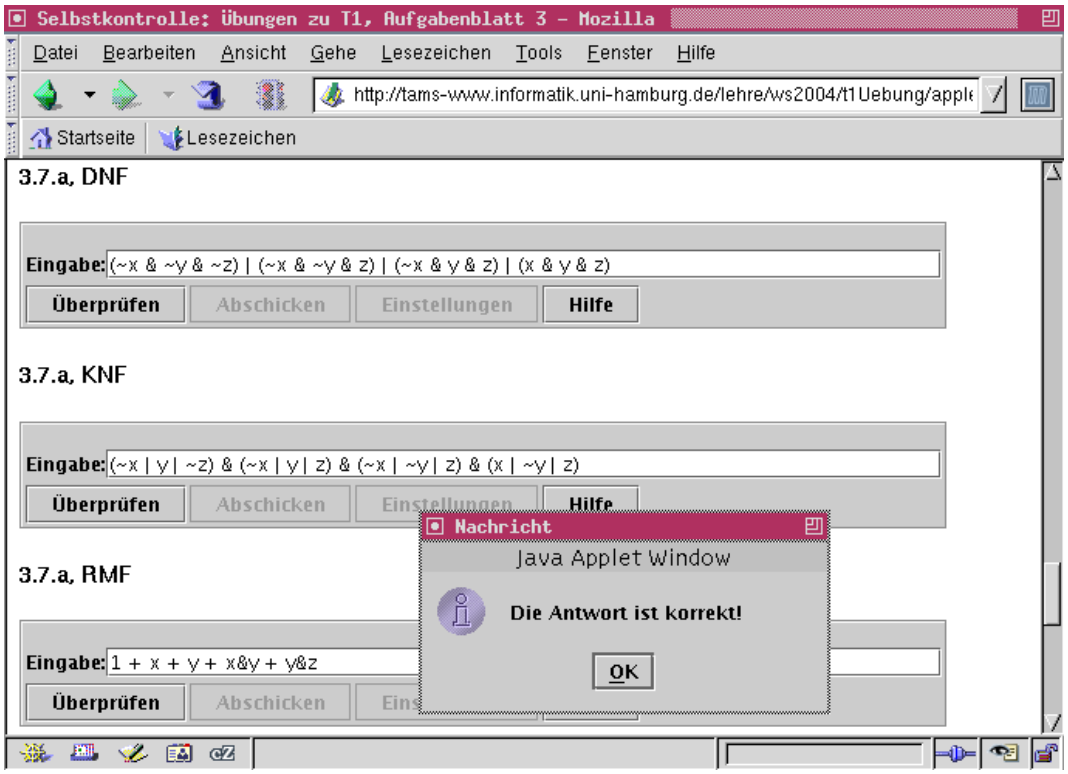

Abbildung 13: *CheckBooleanExpressionApplet dient zur Eingabe und Überprüfung von Boole'schen Ausdrücken. Dabei können diverse Nebenbedingungen berücksichtigt werden, unter anderem die Beschränkung auf bestimmte Operationen oder der Realisierungskosten. Im Beispiel wird dieselbe logische Funktion nacheinander in disjunktiver und konjunktiver Normalform und in Reed-Muller Form gefordert.*

Zur Kodierung der Musterlösung kann StringScrambler verwendet werden, während BP2 einen Ausdruck in eine Funktionstabelle umwandeln kann:

```
java de.mmkh.tams.StringScrambler -encode '(a&b&c)|(~a&~b&c)|(b+d)'
KGEmYiZjKSB8ICh+YSZ+YiZjKSB8IChiK2Qp

java de.mmkh.tams.BP2 '(a&b&c)|(~a&~b&c)|(b+d)' 'a,b,c,d'
0011101111001101
```

```
<applet
 code=de.mmkh.tams.CheckBooleanExpressionApplet.class
 codebase="." archive="checkapplets.jar"
 width=600 height=90
>
<param name="varnames"    value="x,y,z" />
<param name="default"     value="x + (x&y) ... " />
<param name="table"       value="10001011" />
<param name="form"        value="RMF" />
</applet>
```

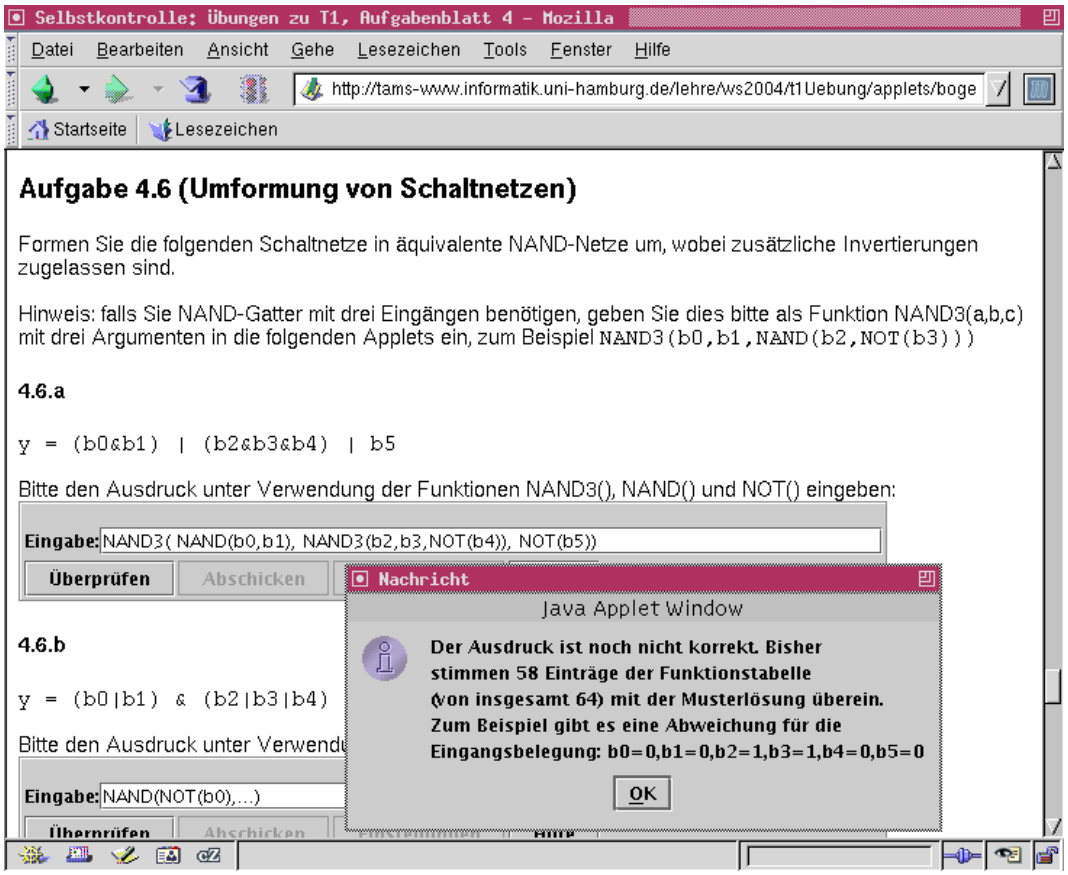

Abbildung 14: *Beispiel zum Einsatz von CheckBooleanExpressionApplet zur Überprüfung eines Ausdrucks in funktionaler Schreibweise mit den Funktionen NAND3(), NAND(), und NOT(). Die Hilfestellung verweist gezielt auf die noch vorhandenen Fehler.*

```
<h4>4.6.a</h4>
<p>
<code>y = (b0&b1) | (b2&b3&b4) | b5</code>
<p>
Bitte den Ausdruck unter Verwendung der Funktionen NAND3(),
NAND() und NOT() eingeben:
<applet
 code=de.mmkh.tams.CheckBooleanExpressionApplet.class
 codebase="." archive="checkapplets.jar"
 width=600 height=90
>
<param name="default"    value="NOT(NAND3(b0,b1,b2))" />
<param name="table"      value="0001000100010001000100010001000\
                  1111111111111111111111111111111111111111" />
<param name="varnames"   value="b0,b1,b2,b3,b4,b5" />
<param name="form"       value="NANDNOT" />
</applet>
<p>
```

(Leerseite)

2.18 CheckTwolevelApplet

Konzept Die Klasse *CheckTwolevelApplet* ermöglicht die graphische Eingabe einer zweistufigen Schaltfunktion in disjunktiver bzw. konjunktiver Form durch interaktives Setzen von Verbindungen („Lötpunkten“) in einem fest vorgegebenen Schaltplan. Die Darstellung entspricht dabei der in Datenblättern üblichen Darstellung von programmierbaren Logikbausteinen wie PLA oder GALs, so dass das Applet gleichzeitig auch zur Einführung in die Anwendung programmierbarer ICs dienen kann.

Die Lötpunkte werden durch einfaches Anklicken eines Kreuzungspunktes gesetzt und beim zweiten Anklicken wieder entfernt. Die zugehörige Logikfunktion wird vom Applet automatisch berechnet und in der Statuszeile (unten) als logischer Ausdruck angezeigt. Zur Überprüfung des Ausdrucks auf Korrektheit und Realisierungskosten dient wiederum die Klasse *BP2* als Parser.

Parameter CheckTwolevelApplet unterstützt fast dieselben Parameter wie CheckBooleanExpression-Applet, vgl. Seite 28:

varnames	Liste der Variablennamen mit Komma als Trennzeichen, dies setzt auch die Anzahl der Variablen *n*.
table	Funktionstabelle der Musterlösung als String aus (01*) der Länge 2^n Zeichen.
form	Auswahl der disjunktiven bzw. konjunktiven Form über die Werte SOP bzw. POS.
costs	Optionale Vorgabe der Realisierungskosten, nacheinander die Gesamtanzahl der Gatter und dann die Anzahl der, UND, ODER, XOR, NOT-Gatter, also z.B. 5,2,3,0,-1. Ein Wert von -1 wird für die Überprüfung ignoriert.

Encoding Da der Aufwand zur Aufstellung der Funktionstabelle mit der Anzahl *n* der Variablen exponentiell anwächst, sollten überflüssige Variablen soweit möglich vermieden werden. Die Angabe der Musterlösung kann derzeit nur als Funktionstabelle erfolgen. Diese kann bei Bedarf mit BP2 aus einem Ausdruck berechnet werden:

```
java de.mmkh.tams.BP2 '(a&b&c&d)|(a&~b&c&~d)' 'a,b,c,d'
0000010000000001
```

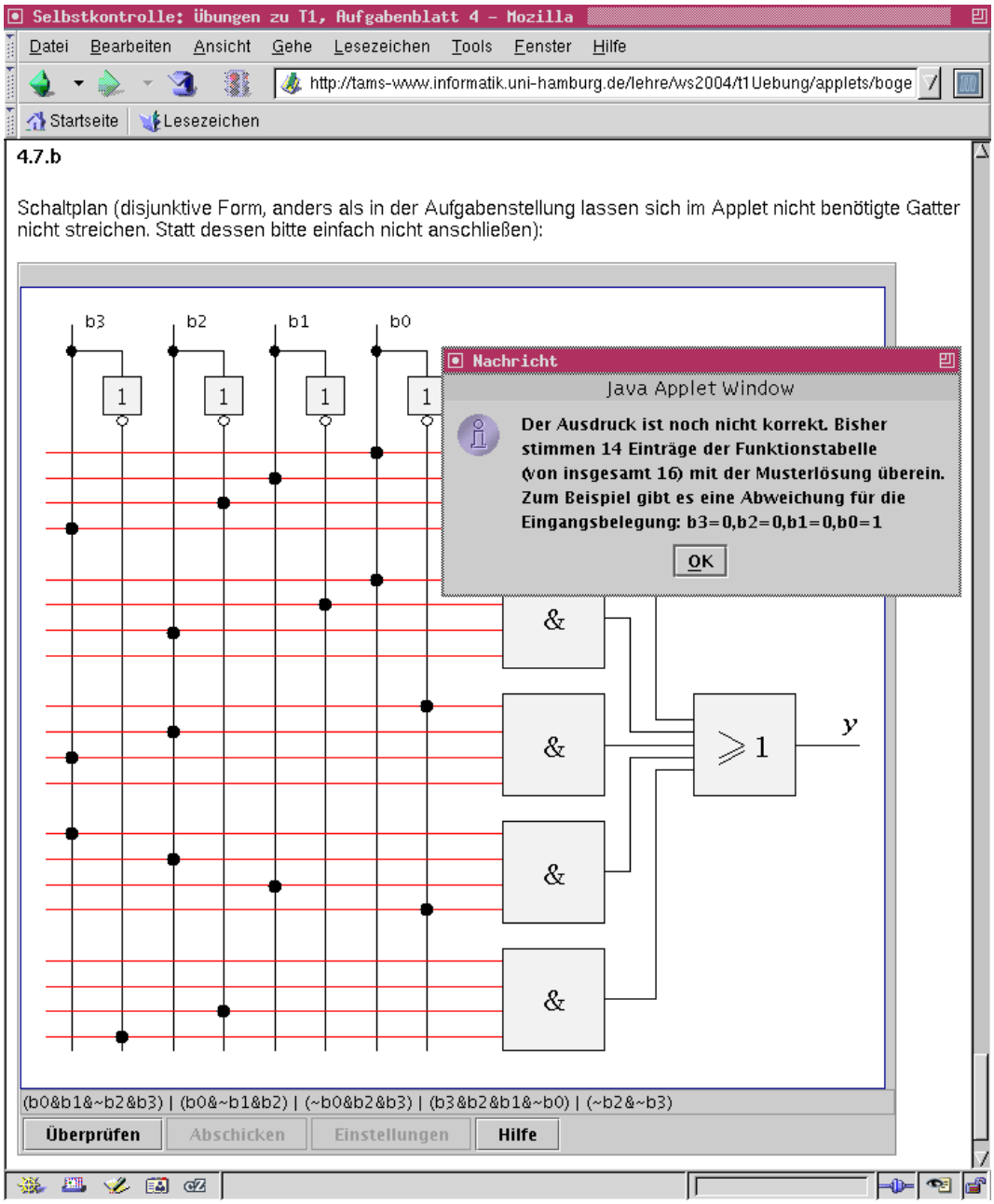

Abbildung 15: *CheckTwolevelApplet ermöglicht die graphische Eingabe einer zweistufigen (UND-ODER bzw. ODER-UND) Schaltfunktion. Durch Anklicken der Leitungskreuzungen können Verbindungspunkte gesetzt und wieder gelöscht werden. Dies dient als Einführung in die Funktionsweise und den Entwurf programmierbarer Logik-Bausteine wie GALs und FPGAs. Zur Überprüfung wird intern der zugehörige Boole'sche Ausdruck aufgestellt (siehe untere Zeile im Applet) und mit der vorgegebenen Funktion verglichen.*

```
<applet code=de.mmkh.tams.CheckTwolevelApplet.class  codebase="."
 archive="checkapplets.jar" width="650" height="660"
>
<param name="default"    value="(b0&b1&b2&b3) | (b0..."> </param>
<param name="table"      value="1001100100110100"> </param>
<param name="varnames"   value="b3,b2,b1,b0"> </param>
<param name="form"       value="SOP"> </param>
<param name="costs"      value="12,-1,-1,-1,-1"> </param>
</applet>
```

2.19 CheckKMapApplet

Konzept und Bedienung

Die graphische Darstellung in Karnaugh-Veitch Diagrammen („K-maps“) ist ein beliebtes Hilfsmittel zur Visualisierung von Schaltfunktionen. Insbesondere die Minimierung des Realisierungsaufwands durch Zusammenfassen von Mintermen (Schleifen) lässt sich quasi auf einen Blick erfassen. Die Klasse *CheckKMapApplet* ermöglicht die einfache Integration von KV-Diagrammen in Webseiten und unterstützt die interaktive Logikminimierung sowohl in disjunktiver Form als auch in konjunktiver Form.

Die Interaktion mit dem Applet erfolgt in zwei Schritten. Zunächst wird der Betriebsmodus ausgewählt, wobei das Editieren der Funktion sowie die Minimierung in disjunktiver (SOP) und konjunktiver Form (POS) zur Verfügung stehen. Zum Editieren einer Funktion werden die einzelnen Felder des Diagramms angeklickt, wobei der zugehörige Funktionswert nacheinander die Werte 0, 1, und * (don’t care) durchläuft.

Anschließend kann auf die Minimierung umgeschaltet werden. Dann erzeugt ein einfacher Mausklick auf ein Feld des Diagramms eine neue Schleife für dieses Feld, die durch Anklicken anderer Felder bei gedrückter Umschalt-Taste (shift+click) entsprechend erweitert werden kann. Unzulässige Felder werden dabei von vornherein blockiert. Durch Anklicken eines Feldes bei gedrückter Steuerungs-Taste (cntl+click) wird eine Schleife über diesem Feld gelöscht.

Nach jeder Benutzerinteraktion berechnet das Applet den zur aktuellen Funktion und den vorhandenen Schleifen zugehörigen Boole’schen Ausdruck und zeigt diesen an. Für die Überprüfung wird dieser Ausdruck ähnlich wie in *CheckBooleanExpressionApplet* an den *BP2*-Parser übergeben und dort ausgewertet.

Parameter

CheckKMapApplet unterstützt die folgenden Parameter:

varnames	Liste der Variablennamen mit Komma als Trennzeichen, dies setzt auch die Anzahl der Variablen *n*.
fname	Name der im KV-Diagramm angezeigten Funktion.
table	Funktionstabelle der Musterlösung als String aus (01*) der Länge 2^n Zeichen.
form	Liste der erlaubten Betriebsarten, zulässige Werte sind EDIT, SOP, POS.
costs	Optionale Vorgabe der Realisierungskosten, nacheinander die Gesamtanzahl der Gatter und dann die Anzahl der, UND, ODER, XOR, NOT-Gatter, also z.B. 21,16,5,0,-1. Ein Wert von -1 wird für die Überprüfung ignoriert.

Encoding

Da der Aufwand zur Aufstellung der Funktionstabelle mit der Anzahl *n* der Variablen exponentiell anwächst, sollten überflüssige Variablen soweit möglich vermieden werden. Zur Kodierung der Musterlösung aus einem Ausdruck kann BP2 dienen:

```
java de.mmkh.tams.BP2 '(a&b&c)|(~a&~b&c)|(b+d)' 'a,b,c,d'
0011101111001101
```

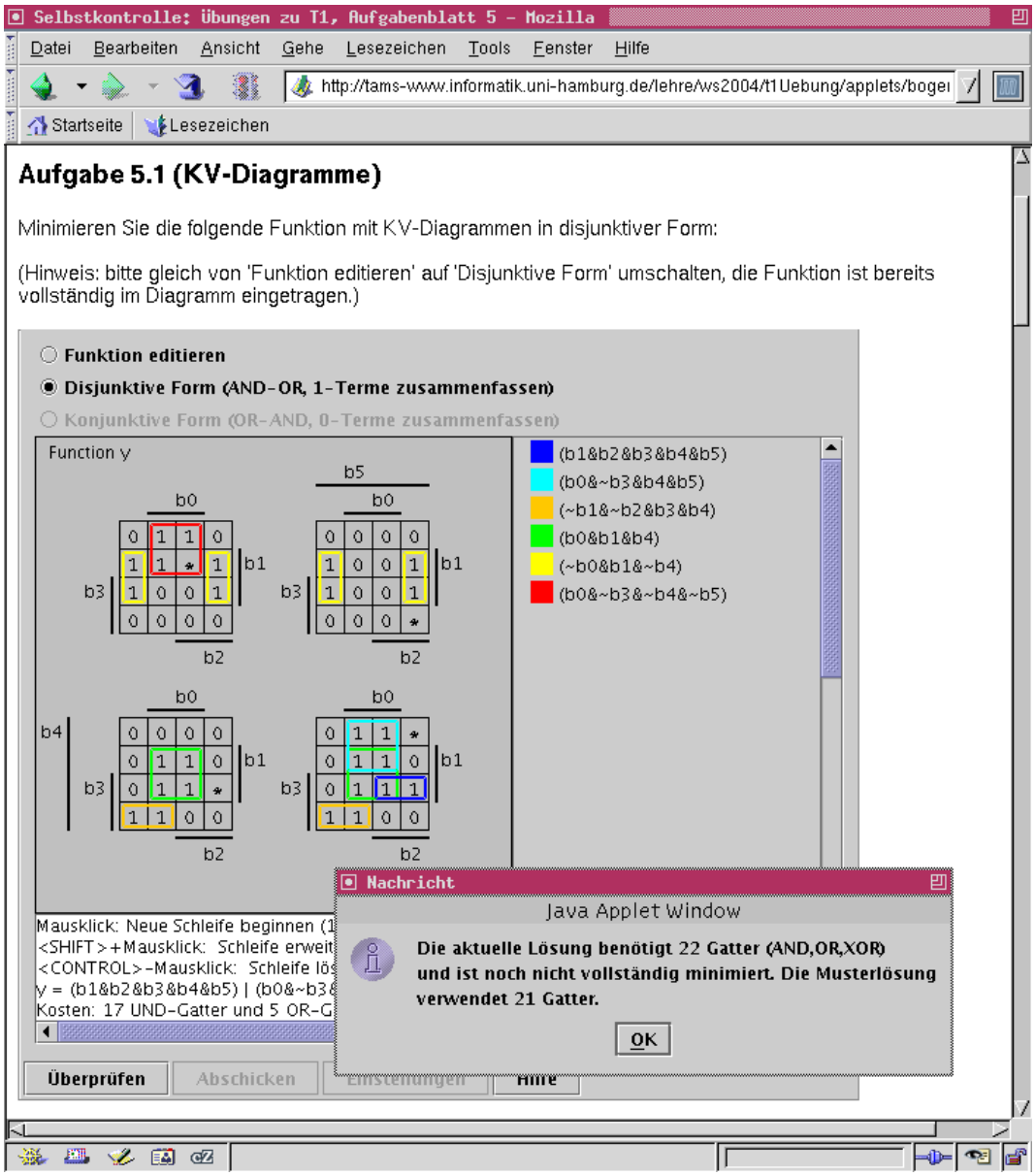

Abbildung 16: *Beispiel für den Einsatz von CheckKMapApplet zur Demonstration der Logikminimierung mit Karnaugh-Veitch Diagrammen. Das Applet erlaubt die interaktive Eingabe der geforderten Funktion und anschließend das ebenfalls interaktive Setzen und Löschen der Schleifen zum Zusammenfassen der Terme.*

```
<applet code=de.mmkh.tams.CheckKMapApplet.class
 codebase="." archive="checkapplets.jar"
 width="600" height="550" >
<param name="nvars"     value="6" />
<param name="varnames"  value="b0,b1,b2,b3,b4,b5" />
<param name="form"      value="SOP" />
<param name="costs"     value="21,16,5,-1,-1" />
<param name="table"     value="0111011*00100010000\
 10001110100*1001000100010*0100101*10111010011" />
<param name="default"   value="0111011*00100010000\
 10001110100*1001000100010*0100101*10111010011" />
</applet>
```

2.20 CheckFormulaApplet

Konzept Zur Überprüfung der Kategorie von *Formelaufgaben* liegt der Einsatz kommerzieller Softwarepakete wie Mathematica, Matlab oder Maple nahe. Im Kontext der Check*Applets kommt dies jedoch nicht in Frage, da die Applets bei möglichst kleiner Downloadgröße ja lizenzkostenfrei und standalone einsetzbar sein sollen. Die Klasse *CheckFormulaApplet* greift deshalb auf einen selbstentwickelten Parser *AXP* für arithmetische Ausdrücke zurück, der wiederum mit Hilfe des JavaCC Parsergenerators [47] erstellt wurde.

Der Parser erkennt arithmetische Ausdrücke mit Integer- und Gleitkommawerten als Literalen, die Konstante pi, benutzerdefinierte Variablen, die elementaren Rechenoperationen (+,-,*,/) und ** für Potenzrechnung, sowie die üblichen mathematischen Funktionen (abs, min, max, sgn, ceil, floor, sin, cos, tan, asin, acos, atan, exp, log, log2, log10, sqrt). Die Funktion sum() erlaubt die Berechnung von Summen über abhängige Parameter.

Der intern angelegte Parse-Baum erlaubt die mehrfache Auswertung eines einmal geparsten Ausdrucks für unterschiedliche Variablenbelegungen, so dass die numerische Überprüfung gegen vorgegebene Datenwerte oder einen zweiten Ausdruck effizient möglich ist.

Parameter Die Klasse CheckFormulaApplet unterstützt die folgenden Parameter:

varnames	Liste der Variablennamen mit Komma als Trennzeichen, dies setzt auch die Anzahl der Variablen n.
range(i)	Definition des Wertebereichs für die Variable i (mit $0 \leq i < n$) als Liste mit der unteren und oberen Grenze sowie der Anzahl der Teilschritte. Zum Beispiel entspricht die Angabe von `range0="0,8,5"` der Angabe von `samples0="0,2,4,6,8"`, vgl. die Matlab `linspace()` Funktion.
samples(i)	Definition der für die Überprüfung zu verwendenden Abtastwerte für die Variable i (mit $0 \leq i < n$) als Liste, z.B. `"0,1,3,8,115"`. Für jede Variable muss entweder der range oder der samples Parameter angegeben werden.
tolerance	Gleitkommawert mit der maximal zulässigen Abweichung zwischen der eingegebenen Funktion und der Musterlösung.
function	Ausdruck für die Musterlösung im Klartext (für Testzwecke).
scrambled	kodierter Ausdruck für die Musterlösung.
matrix	Liste mit den Gleitkommawerten der Musterlösung an den für die Überprüfung via `range` bzw. `samples` ausgewählten Variablenwerten. Für die Überprüfung muss die Musterlösung entweder über den scrambled- oder den matrix-Parameter zur Verfügung stehen.

Für die Überprüfung wird der eingegebene Ausdruck f_{user} für alle Kombinationen der angegeben Variablenwerte aller Variablen $(x_0, \ldots x_{n-1})$ ausgewertet und mit dem zugehörigen Wert der Musterlösung $f_{\text{master}}(x_0, \ldots, x_{n-1})$ verglichen. Die maximale Abweichung der beiden Werte wird protokolliert. Bei Funktionen mit vielen Variablen ist zu beachten, dass der Rechenaufwand für die Überprüfung sehr schnell anwachsen kann. Die Anzahl der Funktionsaufrufe ist offenbar $n_{\text{total}} = \prod_{i=0}^{n-1} n_i$.

Die geplante Unterstützung für separate obere und untere Grenzen der zulässigen Werte anstelle einer über den gesamten Wertebereich konstanten Toleranz ist bisher noch nicht implementiert.

Encoding Zur Kodierung der Musterlösung für den `scrambled` Parameter kann StringScrambler dienen:

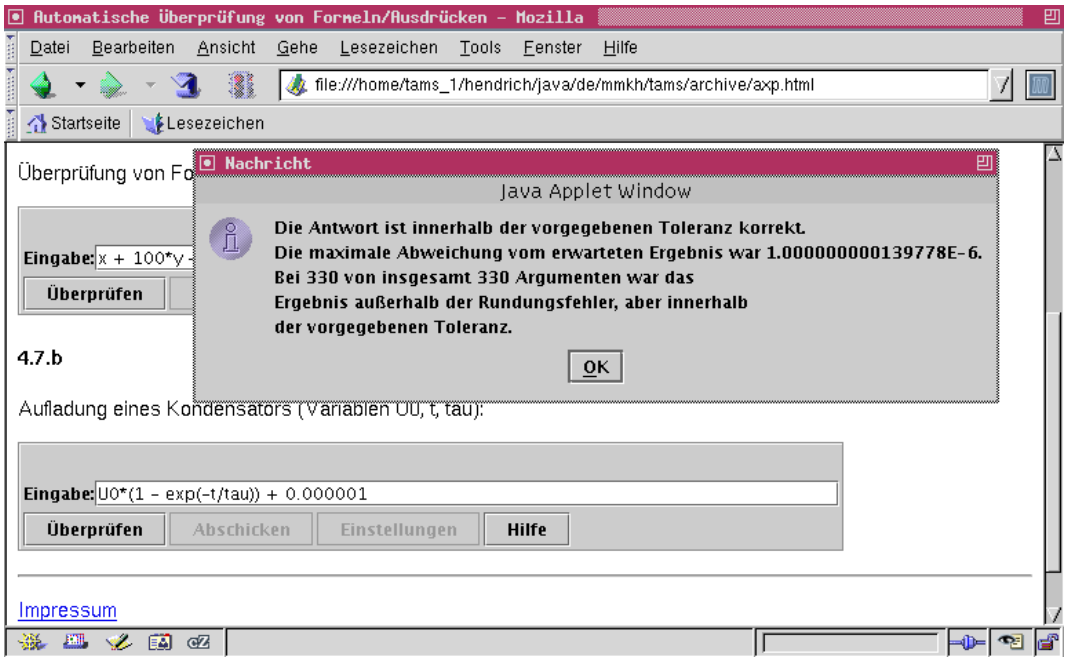

Abbildung 17: *CheckFormulaApplet erlaubt die Überprüfung der von den Studierenden eingegebenen arithmetischen Ausdrücke gegen die zugehörige Musterlösung. Das Beispiel zeigt die Hilfestellung nach Eingabe einer absichtlich (um den Wert 0.000001) ungenauen Formel.*

```
java de.mmkh.tams.StringScrambler 'U0*(1 - exp(-t/tau))'
VTAqKDEgLSBleHAoLXQvdGF1KSk=
```

Alternativ kann die Klasse WriteFormulaApplet eingesetzt werden. Diese berechnet zu den angegebenen `range` und `scrambled` Parametern die zugehörigen `matrix` Werte und schreibt den vollständigen HTML-Code für ein CheckFormulaApplet:

```
java de.mmkh.tams.WriteFormulaApplet 'varnames=x,t,tau' \
  'formula=x*exp(1-t/tau)' 'tolerance=1.0E-6' \
  'samples0=1.0,2.0,4.0' 'samples1=0,0.5,1.0,1.5,2.0,2.5' \
  'range2=0,2,20'
...

<applet
 code=de.mmkh.tams.CheckFormulaApplet.class  codebase="."
 archive="checkapplets.jar" width="600" height="80"
>
<param name="nvars"      value="3" />
<param name="varnames"   value="U0,t,tau" />
<param name="range0"     value="0,10,11" />
<param name="samples1"   value="0,5,10,20,50,100" />
<param name="samples2"   value="0.001,0.01,0.1,1,10" />
<param name="tolerance"  value="0.00001" />
<param name="scrambled"  value="VTAqKDEgLSBleHAoLXQvdGF1KSk=" />
<param name="function"   value="U0*(1 - exp(-t/tau))" />
<param name="default"    value="..." />
</applet>
```

Literatur

[1] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, C-35(8):677–691, 1986

[2] R. E. Bryant, D. R. O'Hallaron, *Computer Systems, A Programmer's Perspective*, Prentice Hall, 2003, ISBN 0-13-034074-X

[3] Fachbereich Informatik der Universität Hamburg, *Studienführer Informatik 2002/2003*, www.informatik.uni-hamburg.de/

[4] K. v. d. Heide, *Vorlesung Technische Informatik 1*, Universität Hamburg, FB Informatik, WS2002,
tams-www.informatik.uni-hamburg.de/lehre/ws2002/t1Vor/

[5] K. v. d. Heide, *Vorlesung Technische Informatik 2*, Universität Hamburg, FB Informatik, SS2003,
tams-www.informatik.uni-hamburg.de/lehre/ss2003/vorlesungen/T2/

[6] K. v. d. Heide, M. Grove, *Übungsaufgaben und Musterlösungen zur Vorlesung Technische Informatik*, Universität Hamburg, FB Informatik, SS2003,
tams-www.informatik.uni-hamburg.de/onlineDoc/

[7] K. v. d. Heide, M. Grove, N. Hendrich, B. Wolfinger, *Praktikum Technische Informatik 1-4*, Universität Hamburg, FB Informatik, tams-www.informatik.uni-hamburg.de/onlineDoc/

[8] N. Hendrich, *Hades Tutorial*, tams-www.informatik.uni-hamburg.de/applets/hades/archive/tutorial.pdf

[9] N. Hendrich, *A Java-based framework for simulation and teaching*, Proc. 3rd European Workshop on Microelectronics Education,
EWME-2000, 285–288, Aix en Provence, 2000

[10] N. Hendrich, *Automatic checking of students' designs using built-in selftest methods*, Proc. 3rd European Workshop on Microelectronics Education, EWME-2002, Baiona, 2002

[11] N. Hendrich, K. v. d. Heide, *Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungsaufgaben*, Projektbericht,
Multimedia-Kontor Hamburg, 2003

[12] N. Hendrich, K. v. d. Heide, *Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungsaufgaben*, Zweiter Projektbericht,
Multimedia-Kontor Hamburg, 2004

[13] N. Hendrich, K. v. d. Heide, *Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungsaufgaben*, Dritter Projektbericht,
Multimedia-Kontor Hamburg, 2004

[14] M. Mayer, *Konzeption und Umsetzung eines Java-Applets zur Logikminimierung mit KV-Diagrammen*, Studienarbeit, Fachbereich Informatik, 1998

[15] J. L. Hennessy, D. A. Patterson, *Computer organization and design: the hardware/software interface*, Morgan Kaufmann, 1998, ISBN 1-558-60491-X

[16] J. Hugunin, *Python and Java — The Best of Both Worlds*, Proc. 6th International Python Conference, San Jose, 1997

[17] D. Jansen (Hrsg.), *Handbuch der Electronic Design Automation* Hanser, 2001 ISBN 3-446-21288-4

[18] *Jython Environment for Students*, Georgia Institute of Technology, 2002 http://coweb.cc.gatech.edu/cs1315/814

[19] Jython project homepage, www.jython.org

[20] J. W. Eaton and others, *GNU Octave Project*, www.octave.org

[21] The MathWorks, Inc., *Matlab Version 5 User's Guide*, 1997 ISBN 0-13-272550-9

[22] The MathWorks, Inc., *Matlab Version 6 User's Guide*, 2002

[23] W. Schiffmann, R. Schmitz, *Technische Informatik 1*, Springer Verlag, 2001, ISBN 3-540-42170-X

[24] W. Schiffmann, R. Schmitz, *Technische Informatik 2*, Springer Verlag, 1999, ISBN 3-540-

[25] W. Schiffmann, R. Schmitz, *Übungsbuch zur Technischen Informatik 1 und 2* (2. Auflage), Springer Verlag, 2001, ISBN 3-540-42171-8

[26] R. W. Schmidt, *Java Network Launching Protocol & API Specification*, JSR-56, Sun Microsystems, Inc., 2001,

[27] R. Schulmeister, *Grundlagen hypermedialer Lernsysteme: Theorie – Didaktik – Design*, Oldenbourg, 1997, ISBN 3-486-24419-1

[28] A. S. Tanenbaum, *Structured Computer Organization, 4th. Edition*, Prentice Hall, 1999 ISBN 0-13-020435-8

[29] imc information multimedia communication AG, *Clix 4 Campus* Lernplattform, http://www.im-c.de/homepage/clix_campus.htm

[30] World wide web consortium, *Extensible Markup Language (XML)*, http://www.w3.org/XML/

[31] P. Hubwieser, *Didaktik der Informatik*, Springer 2000, ISBN 3-540-43510-7

[32] H. Wojtkowiak, *Test und Testbarkeit digitaler Schaltungen*, Teubner 1988, ISBN 3-519-02263-X

[33] N. Freed, N. Borenstein, *Multipurpose Internet Mail Extensions — MIME, Part One: Format of Internet Message Bodies*, Network Working Group, RFC 2045, www.ietf.org/rfc/rfc2045.txt

[34] D. Raggett, A. Le Hors, I. Jacobs, Eds. *World-wide web consortium, HTML 4.01 Specification*, W3C Recommendation 24 December 1999, http://www.w3.org/TR/html401

[35] S. Emmerson, Java Specification Requests, *JSR 108: Units Specification*, http://www.jcp.org/en/jsr/detail?id=108, http://jade.dautelle.com

[36] A. Ruge, *Ein HTML-Browser für interaktive Lehrbücher*, Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 2004

[37] K. Schneider, *Verification of reactive systems: formal methods and algorithms*, Springer, 2004

[38] B. Steffen, G. Levi (eds.), *Verification, Model Checking, and Abstract Interpretation*, Proc. VMCAI 2004, LNCS 2937, Springer 2004, ISBN 3-540-20803-8

[39] W. Menzel, *INCOM – Inputkorrektur durch Constraints und Markups*, E-Learning Consortium Hamburg Projekt, Fachbereich Informatik, Universität Hamburg, 2003

[40] M. Sommer, *Inside CPU*, E-Learning Consortium Hamburg Projekt, Hamburger Universität für Wirtschaft und Politik, 1997

[41] Y. Nahapetian, *Simulation eines von-Neumann-Rechners in der Programmiersprache Java*, Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1996

[42] Lattice Semiconductor Corp., *GAL 16V8 datasheet*, Hillsboro, Oregon, 2004

[43] T. Lindholm, F. Yellin, *Java Virtual Machine Specification, 2nd Edition*, Addison-Wesley, 1999

[44] C. Shannon, *The synthesis of two-terminal switching circuits*, Bell Syst. Techn. Journal 28, 59-98, 1949

[45] I. Wegener, *The Complexity of Boolean Functions*, Wiley-Teubner, 1987, http://ls2-www.cs.uni-dortmund.de/monographs/bluebook

[46] Arash Vahidi, *JDD, a Java Binary Decision Diagram Library*, http://javaddlib.sourceforge.net/jdd/intro.html

[47] *Java compiler compiler - the Java parser generator*, https://javacc.dev.java.net/